

# INTRODUÇÃO AO TESTE DE SOFTWARE

# INTRODUÇÃO AO TESTE DE SOFTWARE

*Márcio Eduardo Delamaro*

*José Carlos Maldonado*

*Mario Jino*

## **Consultoria Editorial**

*Sergio Guedes*

*Núcleo de Computação Eletrônica*

*Universidade Federal do Rio de Janeiro*



4<sup>a</sup> Tiragem



© 2007, Elsevier Editora Ltda.

Todos os direitos reservados e protegidos pela Lei 9.610 de 19/02/1998.

Nenhuma parte deste livro, sem autorização prévia por escrito da editora, poderá ser reproduzida ou transmitida sejam quais forem os meios empregados: eletrônicos, mecânicos, fotográficos, gravação ou quaisquer outros.

Copidesque:

Gypsi Canetti

Editoração Eletrônica:

Márcio Delamaro

Revisão Gráfica:

Marília Pinto de Oliveira

Renato Carvalho

Projeto Gráfico

Elsevier Editora Ltda.

A Qualidade da Informação.

Rua Sete de Setembro, 111 – 16º andar

20050-006 Rio de Janeiro RJ Brasil

Telefone: (21) 3970-9300 FAX: (21) 2507-1991

E-mail: info@elsevier.com.br

Escritório São Paulo:

Rua Quintana, 753/8º andar

04569-011 Brooklin São Paulo SP

Tel.: (11) 5105-8555

ISBN 13: 978-85-352-2634-8

ISBN 10: 85-352-2634-6

**Nota:** Muito zelo e técnica foram empregados na edição desta obra. No entanto, podem ocorrer erros de digitação, impressão ou dúvida conceitual. Em qualquer das hipóteses, solicitamos a comunicação à nossa Central de Atendimento, para que possamos esclarecer ou encaminhar a questão.

Nem a editora nem os autores assumem qualquer responsabilidade por eventuais danos ou perdas a pessoas ou bens, originados do uso desta publicação.

Central de atendimento

Tel.: 0800-265340

Rua Sete de Setembro, 111, 16º andar – Centro – Rio de Janeiro

e-mail: info@elsevier.com.br

site: www.campus.com.br

CIP-BRASIL. CATALOGAÇÃO-NA-FONTE  
SINDICATO NACIONAL DOS EDITORES DE LIVROS, RJ

---

I48

Introdução ao teste de software / organização Márcio  
Eduardo Delamaro, José Carlos Maldonado, Mario Jino  
– Rio de Janeiro : Elsevier, 2007 – 4ª reimpressão  
ISBN 978-85-352-2634-8  
1. Software – Testes. I. Delamaro, Márcio, 1963–.  
II. Maldonado, José Carlos, 1954 –. III. Jino, Mario.

---

07-1623.

CDD 005.14  
CDU 004.415.538

27.04.07 15.05.07

001735

---

# Prefácio

Temos notado um enorme crescimento do interesse, por parte dos desenvolvedores, nas questões relacionadas à qualidade de software. Em particular, a indústria tem despertado para a extrema importância da atividade de teste que, por um lado, pode contribuir para a melhoria da qualidade de um determinado produto e, por outro, representar um custo significativo dentro dos orçamentos empresariais.

Por isso, torna-se indispensável que, no processo de desenvolvimento de software, sejam adotados métodos, técnicas e ferramentas que permitam a realização da atividade de teste de maneira sistematizada e com fundamentação científica, de modo a aumentar a produtividade e a qualidade e a diminuir custos.

Um dos problemas que sentimos é a escassez de mão-de-obra especializada nessa área. Assim, o primeiro objetivo deste livro é servir como livro-texto para disciplinas de cursos relacionados ao desenvolvimento de software como Ciência ou Engenharia da Computação e Sistemas de Informação.

Acreditamos servir, também, como um texto introdutório para profissionais da área que necessitam de uma fonte de consulta e aprendizado. Neste livro tal profissional poderá encontrar as informações básicas relativas às técnicas de teste, bem como formas de aplicá-las nos mais variados domínios e tipos de software.

Podemos dividir o livro em três partes. A primeira, composta pelos Capítulos 1 a 5, apresenta as principais técnicas de teste, mostrando a teoria e os conceitos básicos relacionados a cada uma delas. Sempre que possível, apresenta-se um histórico de como surgiu e como evoluiu cada técnica. Na segunda parte, que vai do Capítulo 6 ao 9, discute-se como essas técnicas têm sido empregadas em diversos tipos de software e diversos domínios de aplicação como: aplicações *Web*, programas baseados em componentes, programas concorrentes e programas baseados em aspectos. Na terceira parte, composta pelos Capítulos 10 a 13, são tratados temas intimamente relacionados ao teste de software e que devem ser, também, de interesse dos desenvolvedores: geração de dados de teste, depuração e confiabilidade.

Em cada um dos capítulos procuramos dar uma visão relativamente aprofundada de cada um dos temas e, também, uma visão de quais são os mais importantes trabalhos realizados e que constituem o estado da arte na área. Uma farta lista de referências bibliográficas foi incluída para permitir aos interessados o aprofundamento nesta atividade tão necessária dentro da indústria de desenvolvimento de software e nesse tão fascinante campo de pesquisa.

## Os autores

A tabela a seguir indica os nomes dos autores e os capítulos para os quais eles contribuíram.

Autores	Capítulos												
	1	2	3	4	5	6	7	8	9	10	11	12	13
Adalberto													✓
Adenilso			✓										
André						✓							
Auri		✓		✓	✓	✓				✓			
Chaim			✓						✓		✓		
Cláudia							✓						
Delamaro	✓			✓	✓	✓				✓			
Ellen				✓	✓				✓				
Jino	✓			✓				✓		✓	✓	✓	✓
Maldonado	✓	✓		✓	✓	✓			✓	✓	✓	✓	
Masiero							✓						
Otávio							✓						
Paulo								✓					
Reginaldo						✓							
Sandra		✓							✓				
Silvia							✓	✓			✓		
Simone								✓	✓				

**Adalberto:** Adalberto Nobiato Crespo, pesquisador do CENPRA, Campinas, e professor adjunto e pesquisador da USF, Itatiba (adalberto@cenpra.gov.br).

**Ades:** Adenilso da Silva Simão, professor adjunto e pesquisador do ICMC/USP, São Carlos (adenilso@icmc.usp.br).

**André:** André Luís dos Santos Domingues, doutorando do ICMC/USP, São Carlos (alsd@icmc.usp.br).

**Auri:** Auri Marcelo Rizzo Vincenzi, professor doutor e pesquisador da UNISANTOS, Santos (auri@pq.cnpq.br).

**Chaim:** Marcos Lordello Chaim, professor doutor do EACH/USP, São Paulo (chaim@usp.br).

**Cláudia:** Maria Cláudia Figueiredo Pereira Emer, doutoranda do DCA/FEEC/UNICAMP, Campinas (mcemer@dca.fee.unicamp.br).

**Delamaro:** Márcio Eduardo Delamaro, professor doutor e pesquisador do UNIVEM, Marília (delamaro@pesquisador.cnpq.br).

**Ellen:** Ellen Francine Barbosa, professora adjunta e pesquisadora do ICMC/USP, São Carlos (francine@icmc.usp.br).

**Jino:** Mario Jino, professor titular e pesquisador do DCA/FEEC/UNICAMP, Campinas (jino@dca.unicamp.br).

**Maldonado:** José Carlos Maldonado, professor titular e pesquisador do ICMC/USP, São Carlos ([jcmaldon@icmc.usp.br](mailto:jcmaldon@icmc.usp.br)).

**Masiero:** Paulo César Masiero, professor titular e pesquisador do ICMC/USP, São Carlos ([masiero@icmc.usp.br](mailto:masiero@icmc.usp.br)).

**Otávio:** Otávio Augusto Lazzarini Lemos, doutorando em Ciência da Computação no ICMC/USP, São Carlos ([oall@icmc.usp.br](mailto:oall@icmc.usp.br)).

**Paulo:** Paulo Sérgio Lopes de Souza, professor adjunto e pesquisador do ICMC/USP, São Carlos ([pssouza@icmc.usp.br](mailto:pssouza@icmc.usp.br)).

**Reginaldo:** Reginaldo Ré, professor de 1º e 2º graus e pesquisador da UTFPR, Campo Mourão ([reginaldo@utfpr.edu.br](mailto:reginaldo@utfpr.edu.br)).

**Sandra:** Sandra C. Pinto Ferraz Fabbri, professora adjunta e pesquisadora da UFSCar, São Carlos ([sfabbri@dc.ufscar.br](mailto:sfabbri@dc.ufscar.br)).

**Silvia:** Silvia Regina Vergilio, professora adjunta e pesquisadora do DInf/UFPR, Curitiba ([silvia@inf.ufpr.br](mailto:silvia@inf.ufpr.br)).

**Simone:** Simone do Rocio Senger de Souza, professora adjunta e pesquisadora do ICMC/USP, São Carlos ([srocio@icmc.usp.br](mailto:srocio@icmc.usp.br)).

# Capítulo 1

## Conceitos Básicos

*Márcio Eduardo Delamaro (UNIVEM)  
José Carlos Maldonado (ICMC/USP)  
Mario Jino (DCA/FEEC/UNICAMP)*

### 1.1 Introdução

Para que possamos tratar de um assunto tão vasto quanto o proposto neste livro é necessário, primeiro, estabelecer uma linguagem própria. Como acontece em muitas das áreas da Computação e das ciências em geral, termos comuns adquirem significados particulares quando usados tecnicamente.

Este capítulo tem o objetivo de estabelecer o escopo no qual o restante do livro se insere, de apresentar os principais termos do jargão da área e introduzir conceitos que serão úteis durante a leitura do restante do texto.

### 1.2 Validação, verificação e teste de software

Definitivamente, a construção de software não é uma tarefa simples. Pelo contrário, pode se tornar bastante complexa, dependendo das características e dimensões do sistema a ser criado. Por isso, está sujeita a diversos tipos de problemas que acabam resultando na obtenção de um produto diferente daquele que se esperava.

Muitos fatores podem ser identificados como causas de tais problemas, mas a maioria deles tem uma única origem: erro humano. Como a maioria das atividades de engenharia, a construção de software depende principalmente da habilidade, da interpretação e da execução das pessoas que o constroem; por isso, erros acabam surgindo, mesmo com a utilização de métodos e ferramentas de engenharia de software.

Para que tais erros não perdurem, ou seja, para serem descobertos antes de o software ser liberado para utilização, existe uma série de atividades, coletivamente chamadas de “Validação, Verificação e Teste”, ou “VV&T”, com a finalidade de garantir que tanto o modo pelo qual o software está sendo construído quanto o produto em si estejam em conformidade com o especificado.

Atividades de VV&T não se restringem ao produto final. Ao contrário, podem e devem ser conduzidas durante todo o processo de desenvolvimento do software, desde a sua concepção, e englobam diferentes técnicas.

Costumamos dividir as atividades de VV&T em estáticas e dinâmicas. As estáticas são aquelas que não requerem a execução ou mesmo a existência de um programa ou modelo executável para serem conduzidas. As dinâmicas são aquelas que se baseiam na execução de um programa ou de um modelo.

O objeto principal deste livro – o teste de software – é uma atividade dinâmica. Seu intuito é executar o programa ou modelo utilizando algumas entradas em particular e verificar se seu comportamento está de acordo com o esperado. Caso a execução apresente resultados não especificados, dizemos que um erro ou defeito (ver Seção 1.3) foi identificado. Além disso, os dados de tal execução podem servir como fonte de informação para a localização e a correção de tais defeitos.

Outras atividades de VV&T não são tratadas neste texto. São atividades importantes e que permitem a eliminação de erros desde as fases iniciais do processo de desenvolvimento, o que, em geral, representa uma economia significativa de recursos. Alguns exemplos de tais atividades são as revisões técnicas e os *walkthroughs*.

### 1.3 Alguns termos do jargão

Nesta seção procuramos identificar alguns termos importantes e que ajudarão o leitor a melhor compreender os textos dos próximos capítulos.

Na seção anterior dissemos que diversos “problemas” podem emergir durante o desenvolvimento de software e, em seguida, nos referimos a esses problemas como “erros”. A literatura tradicional estabelece significados específicos para este e outros termos relacionados como: falha, defeito, erro, engano. Vamos procurar identificar cada um deles.

Definimos “defeito” (do inglês, *fault*) como sendo um passo, processo ou definição de dados incorretos e “engano” (*mistake*) como a ação humana que produz um defeito. Assim, esses dois conceitos são estáticos, pois estão associados a um determinado programa ou modelo e não dependem de uma execução particular.

O estado de um programa ou, mais precisamente, da execução de um programa em determinado instante, é dado pelo valor da memória (ou das variáveis do programa) e do apontador de instruções. A existência de um defeito pode ocasionar a ocorrência de um “erro” (*error*) durante uma execução do programa, que se caracteriza por um estado inconsistente ou inesperado. Tal estado pode levar a uma “falha” (*failure*), ou seja, pode fazer com que o resultado produzido pela execução seja diferente do resultado esperado.

Note-se que essas definições não são seguidas o tempo todo nem são unanimidade entre os pesquisadores e engenheiros de software, principalmente em situações informais do dia-a-dia. Em particular, utiliza-se “erro” de uma maneira bastante flexível, muitas vezes significando defeito, erro ou até falha.

O “domínio” de entrada de um programa  $\mathbf{P}$ , denotado por  $D(P)$ , é o conjunto de todos os possíveis valores que podem ser utilizados para executar  $\mathbf{P}$ . Por exemplo, vamos tomar um programa que recebe como parâmetros de entrada dois números inteiros  $x$  e  $y$ , com  $y \geq 0$ ,

e computa o valor de  $x^y$ , indicando um erro caso os argumentos estejam fora do intervalo especificado. O domínio de entrada deste programa é formado por todos os possíveis pares de números inteiros  $(x, y)$ . Da mesma forma, pode-se estabelecer o domínio de saída do programa, que é o conjunto de todos os possíveis resultados produzidos pelo programa. No nosso exemplo, seria o conjunto de números inteiros e mensagens de erro produzidos pelo programa.

Um “dado de teste” para um programa **P** é um elemento do domínio de entrada de **P**. Um “caso de teste” é um par formado por um dado de teste mais o resultado esperado para a execução do programa com aquele dado de teste. Por exemplo, no programa que computa  $x^y$ , teríamos os seguintes casos de teste:  $\langle(2, 3), 8\rangle$ ,  $\langle(4, 3), 64\rangle$ ,  $\langle(3, -1), \text{“Erro”}\rangle$ . Ao conjunto de todos os casos de teste usados durante uma determinada atividade de teste costuma-se chamar “conjunto de teste” ou “conjunto de casos de teste”.

A Figura 1.1 mostra um cenário típico da atividade de teste. Definido um conjunto de casos de teste **T**, executa-se o programa em teste com **T** e verifica-se qual é o resultado obtido. Se o resultado produzido pela execução de **P** coincide com o resultado esperado, então nenhum erro foi identificado. Se, para algum caso de teste, o resultado obtido difere do resultado esperado, então um defeito foi revelado. Como sugere a Figura 1.1, em geral, fica por conta do testador, baseado na especificação do programa ( $S(P)$ ) ou em qualquer forma de documento que defina seu comportamento, a análise sobre a correção de uma execução. Na verdade, o testador na figura está desempenhando um papel que costumamos chamar de “oráculo”, isto é, aquele instrumento que decide se a saída obtida de uma determinada execução coincide com a saída esperada.

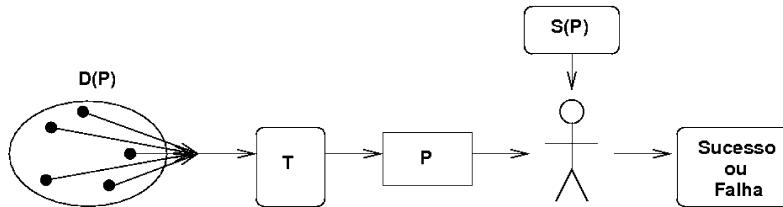


Figura 1.1 – Cenário típico da atividade de teste.

## 1.4 Fases da atividade de teste

A atividade de teste é complexa. São diversos os fatores que podem colaborar para a ocorrência de erros. Por exemplo, a utilização de um algoritmo incorreto para computar o valor das mensalidades a serem pagas para um empréstimo ou a não-utilização de uma política de segurança em alguma funcionalidade do software são dois tipos distintos de engano e, de certa forma, encontram-se em níveis diferentes de abstração. O primeiro tipo de erro provavelmente está confinado a uma função ou rotina que implementa de forma incorreta uma dada funcionalidade. No segundo caso, mesmo que exista uma certa política de segurança implementada de maneira correta, é preciso verificar se todos os pontos nos quais essa política deveria ser aplicada fazem-no de maneira correta.

Por isso, a atividade de teste é dividida em fases com objetivos distintos. De uma forma geral, podemos estabelecer como fases o teste de unidade, o teste de integração e o teste de sistemas. O teste de unidade tem como foco as menores unidades de um programa, que podem ser funções, procedimentos, métodos ou classes. Nesse contexto, espera-se que sejam identificados erros relacionados a algoritmos incorretos ou mal implementados, estruturas de dados incorretas, ou simples erros de programação. Como cada unidade é testada separadamente, o teste de unidade pode ser aplicado à medida que ocorre a implementação das unidades e pelo próprio desenvolvedor, sem a necessidade de dispor-se do sistema totalmente finalizado.

No teste de integração, que deve ser realizado após serem testadas as unidades individualmente, a ênfase é dada na construção da estrutura do sistema. À medida que as diversas partes do software são colocadas para trabalhar juntas, é preciso verificar se a interação entre elas funciona de maneira adequada e não leva a erros. Também nesse caso é necessário um grande conhecimento das estruturas internas e das interações existentes entre as partes do sistema e, por isso, o teste de integração tende a ser executado pela própria equipe de desenvolvimento.

Depois que se tem o sistema completo, com todas as suas partes integradas, inicia-se o teste de sistema. O objetivo é verificar se as funcionalidades especificadas nos documentos de requisitos estão todas corretamente implementadas. Aspectos de correção, completude e coerência devem ser explorados, bem como requisitos não funcionais como segurança, performance e robustez. Muitas organizações adotam a estratégia de designar uma equipe independente para realizar o teste de sistemas.

Além dessas três fases de teste, destacamos, ainda, o que se costuma chamar de “teste de regressão”. Esse tipo de teste não se realiza durante o processo “normal” de desenvolvimento, mas sim durante a manutenção do software. A cada modificação efetuada no sistema, após a sua liberação, corre-se o risco de que novos defeitos sejam introduzidos. Por esse motivo, é necessário, após a manutenção, realizar testes que mostrem que as modificações efetuadas estão corretas, ou seja, que os novos requisitos implementados (se for esse o caso) funcionam como o esperado e que os requisitos anteriormente testados continuam válidos.

Independentemente da fase de teste, existem algumas etapas bem definidas para a execução da atividade de teste. São elas: 1) planejamento; 2) projeto de casos de teste; 3) execução; e 4) análise.

## 1.5 Técnicas e critérios de teste

Podemos notar na Figura 1.1 que, ao se testar um programa  $\mathbf{P}$ , selecionam-se alguns pontos específicos do domínio  $D(P)$  para a execução de  $\mathbf{P}$ . Idealmente, o programa em teste deveria ser executado com todos os elementos de  $D(P)$  para garantir que não contém defeitos. Em geral, tal abordagem é infactível por causa da cardinalidade de  $D(P)$ . Por exemplo, no programa que computa  $x^y$ , o domínio é formado por todos os pares de números inteiros  $(x, y)$ , o que produz um conjunto de cardinalidade  $2^n * 2^n$ , onde  $n$  é o número de bits usado para representar um número inteiro. Em uma arquitetura com 32 bits isso representa  $2^{64} = 18446744073709551616$ . Se cada caso de teste pudesse ser executado em 1 milissegundo, precisaríamos de 5.849.424 séculos para executá-los todos.

Assim, é importante procurarmos formas de utilizar apenas um subconjunto reduzido de  $D(P)$  para a execução de **P**, mas que tenha alta probabilidade de revelar a presença de defeitos caso eles existam. A idéia central para isso é a de “subdomínios” de teste. Um subdomínio de  $D(P)$  é um subconjunto do domínio de entrada que contém dados de teste “semelhantes”. Por exemplo, podemos argumentar que os casos de teste  $\langle(2, -1), \text{“Erro”}\rangle$  e  $\langle(2, -2), \text{“Erro”}\rangle$  são semelhantes, pois, qualquer que seja a implementação, intuitivamente espera-se que **P** se comporte de maneira semelhante para ambos os casos de teste. Nesse caso, não seria necessário executar o programa em teste com os dois casos de teste. Generalizando, qualquer caso de teste da forma  $\langle(x, y), \text{“Erro”}\rangle$  em que  $x > 0$  e  $y < 0$  se comporta do mesmo modo ou, em outras palavras, pertence a um mesmo subdomínio de  $D(P)$ ; por isso, basta executar **P** com apenas um deles. Caso isso seja feito para todos os subdomínios de  $D(P)$ , consegue-se um conjunto de **T** bastante reduzido em relação a  $D(P)$  mas que, de certa maneira, representa cada um dos seus elementos.

Na essência, existem duas formas de se procurar selecionar elementos de cada um dos subdomínios de teste. A primeira é o que se chama de “teste aleatório”, em que um grande número de casos de teste é selecionado aleatoriamente, de modo que, probabilisticamente, se tenha uma boa chance de que todos os subdomínios estejam representados no conjunto de teste **T**. A segunda é o “teste de partição” ou, mais corretamente, “teste de subdomínios”, no qual se procura estabelecer quais são os subdomínios a serem utilizados e, então, selecionar-se os casos de teste em cada subdomínio. Cada uma dessas abordagens tem suas vantagens e desvantagens. Neste livro, as técnicas apresentadas se encaixam na segunda estratégia, não sendo abordado o teste aleatório.

Em relação ao teste de subdomínios, fica a questão de como identificar os subdomínios para, então, fazer a seleção de casos de teste. O que ocorre, na prática, é que são estabelecidas certas “regras” para identificar quando dados de teste devem estar no mesmo subdomínio ou não. Em geral, são definidos “requisitos de teste” como, por exemplo, executar uma determinada estrutura do programa. Os dados de teste que satisfazem esse requisito pertencem ao mesmo subdomínio.

Dessa forma, dependendo do tipo de regra que se utiliza, são obtidos subdomínios diferentes e, portanto, conjuntos diferentes de teste. Tais regras são chamadas “critérios de teste”. Podemos identificar três tipos diferentes de critérios: funcionais, estruturais e baseados em defeitos (ou erros). O que diferencia cada uma dessas técnicas é o tipo de informação utilizada para estabelecer os subdomínios. Um conjunto de teste que satisfaz todos os requisitos de um critério de teste **C**, ou seja, que tem pelo menos um caso de teste para cada subdomínio determinado por **C**, é dito “adequado” a **C** ou “C-adequado”.

Não vamos, aqui, entrar em detalhes sobre as técnicas de teste, pois elas são discutidas na primeira parte deste livro.

## 1.6 Características e limitações

Um único critério de teste é ideal: aquele que estabelece que cada subdomínio de  $D(P)$  é unitário, ou seja, cada dado de teste compõe isoladamente o próprio domínio. Sendo assim, todos os elementos do domínio devem ser selecionados para o teste. É o que chamamos de “teste exaustivo”. Como vimos anteriormente, o teste exaustivo é, em geral, infactível.

Isso significa que, na prática, é impossível mostrar que um programa está correto por meio de teste. Isso porque, aliado a algumas limitações teóricas citadas a seguir, ao se escolher um subconjunto de  $D(P)$  corre-se sempre o risco de deixar de fora algum caso de teste que poderia revelar a presença de um defeito.

Por isso costumamos dizer que o objetivo da atividade de teste não é, como podem pensar muitos, mostrar que um programa está correto. Ao invés disso, o objetivo é mostrar a presença de defeitos caso eles existam. Quando a atividade de teste é realizada de maneira criteriosa e embasada tecnicamente, o que se tem é uma certa “confiança” de que se comporta corretamente para grande parte do seu domínio de entrada.

Algumas limitações teóricas são importantes na definição de critérios de teste, principalmente daqueles baseados no código, ou seja, que utilizam a estrutura do programa para derivar os requisitos de teste. Podemos citar, por exemplo:

- é indecidível se uma determinada seqüência de comandos (um caminho) de um programa é executável ou não. Em outras palavras, não existe um procedimento que indique se existe ou não algum dado de teste que faça com que um caminho do programa seja executado;
- é indecidível se duas rotinas ou dois caminhos computam a mesma função.

Não existe, em princípio, nenhuma restrição sobre o tipo de critérios de teste que pode ser definido e utilizado. Entretanto, a fim de se obter um nível mínimo de qualidade para os conjuntos adequados a um critério **C** para um programa **P**, deve-se requerer que:

- existe um conjunto C-adequado que seja finito e, de preferência, de baixa cardinalidade;
- do ponto de vista do fluxo de execução, um conjunto que seja C-adequado deve garantir que cada comando de **P** seja executado, pelo menos, uma vez;
- do ponto de vista da utilização das variáveis, cada atribuição de valor a uma variável deve ter seu valor verificado por um caso de teste que execute o trecho do programa que vai do ponto em que a variável recebe esse valor até um ponto do programa em que esse valor é utilizado.

## 1.7 Temas relacionados

Vamos abordar, rapidamente, alguns temas importantes, relacionados ao teste de software e que serão tratados na parte final deste livro. Muitos confundem, talvez com certa razão, depuração e teste de software. Embora fortemente relacionadas, são duas atividades diversas. O teste de software tem como objetivo, como vimos anteriormente, mostrar que o software não funciona como o esperado, ou seja, possui defeitos. A depuração, por sua vez, deve localizar as causas que levaram o software a falhar e indicar como ele deve ser corrigido. Por muito tempo, assim como acontecia com a atividade de teste, a depuração baseou-se principalmente na habilidade e na experiência das pessoas encarregadas dessa tarefa, sem que fossem utilizadas técnicas sistemáticas. Recentemente, diversos esforços têm sido observados visando a desenvolver técnicas para melhorar a qualidade e a produtividade nessa atividade. Muitas

delas, inclusive, beneficiam-se de informações geradas durante o teste, como é mostrado no Capítulo 12.

Um ponto importante para o sucesso no teste de um software é a automatização. Diversos tipos de ferramentas de teste têm sido utilizados para buscar aumentar a produtividade nessa atividade, que tende a ser extremamente dispendiosa. Em particular, no projeto de casos de teste torna-se essencial a existência de uma ferramenta de suporte que compute os requisitos de teste de um determinado critério e verifique a adequação de um conjunto de teste. Ainda assim, muito trabalho humano deve ser empregado para criar casos de teste que sejam adequados ao critério utilizado. Nesse sentido, técnicas para geração automática de casos de teste são bastante desejadas para permitir a automatização também desse aspecto do teste. É claro, porém, que não existe um procedimento geral que permita a geração de conjuntos adequados a qualquer critério de teste. Na prática, técnicas específicas são projetadas para cada critério de teste. As técnicas de geração de dados de teste são discutidas com detalhes no Capítulo 11.

Podemos definir confiabilidade como a tendência de um software em operar de maneira correta. Certamente essa é uma das mais importantes – se não a mais importante – características desejadas de um software. Assim, a análise de confiabilidade de software, ou seja, procurar medir de forma objetiva essa característica tornou-se um fator de extrema importância nas organizações que desenvolvem software de maneira profissional. Por outro lado, essa é uma atividade que está longe de ser trivial e demanda técnicas específicas. A análise de confiabilidade baseia-se em conceitos de métricas, medidas e modelos matemáticos que são usados para avaliar o comportamento do software. Por exemplo, medindo-se a freqüência de falhas em um determinado período de tempo e aplicando-se um modelo de falhas pode-se estimar seu nível de confiabilidade e prever seu comportamento futuro. O Capítulo 13 apresenta com detalhes esses conceitos e discute diversos modelos de falhas existentes na literatura.

## 1.8 Considerações finais

Neste capítulo procuramos apresentar alguns conceitos básicos sobre a atividade de teste de software, caracterizando-a dentro do processo de desenvolvimento de software. Foram discutidos brevemente alguns termos importantes sobre o assunto e que deverão ser úteis na leitura dos próximos capítulos deste livro.

# Capítulo 2

## Teste Funcional

*Sandra C. Pinto Ferraz Fabbri (UFSCar)*  
*Auri Marcelo Rizzo Vincenzi (UNISANTOS)*  
*José Carlos Maldonado (ICMC/USP)*

### 2.1 Introdução

Teste funcional é uma técnica utilizada para se projetarem casos de teste na qual o programa ou sistema é considerado uma caixa preta e, para testá-lo, são fornecidas entradas e avaliadas as saídas geradas para verificar se estão em conformidade com os objetivos especificados. Nessa técnica os detalhes de implementação não são considerados e o software é avaliado segundo o ponto de vista do usuário.

Em princípio, o teste funcional pode detectar todos os defeitos, submetendo o programa ou sistema a todas as possíveis entradas, o que é denominado teste exaustivo [294]. No entanto, o domínio de entrada pode ser infinito ou muito grande, de modo a tornar o tempo da atividade de teste inviável, fazendo com que essa alternativa se torne impraticável. Essa limitação da atividade de teste, que não permite afirmar, em geral, que o programa esteja correto, fez com que fossem definidas as técnicas de teste e os diversos critérios pertencentes a cada uma delas. Assim, é possível conduzir essa atividade de maneira mais sistemática, podendo-se inclusive, dependendo do critério utilizado, ter uma avaliação quantitativa da atividade de teste.

O que distingue essencialmente as três técnicas de teste – Funcional, Estrutural e Baseada em Erros – é a fonte utilizada para definir os requisitos de teste. Além disso, cada critério de teste procura explorar determinados tipos de defeitos, estabelecendo requisitos de teste para os quais valores específicos do domínio de entrada do programa devem ser definidos com o intuito de exercitá-los.

Considerando a técnica de teste funcional, um de seus critérios é o Particionamento de Equivalência, como será visto com mais detalhe adiante. O teste de partição tem o objetivo de fazer uma divisão (particionamento) do domínio de entrada do programa de tal modo que, quando o testador seleciona casos de teste dos subconjuntos (partições ou subdomínios), o conjunto de testes resultante é uma boa representação de todo o domínio. Segundo a visão de alguns autores, o teste de partição e o teste randômico são, basicamente, as duas abordagens

para se conduzir a atividade de teste. Em resumo, o teste randômico consiste em definir, de alguma forma, um domínio de entradas para um programa e então testá-lo com dados de teste que são selecionados randomicamente desse domínio.

De acordo com Weyuker e Jeng [431], dependendo de como são olhados, todos os critérios de cobertura, isto é, os critérios de teste estrutural e baseado em erros (apresentados nos Capítulos 4 e 5, respectivamente), que refletem a razão entre o número de requisitos exercitados e o número total de requisitos requeridos, podem ser vistos como uma estratégia de partição. Por exemplo, o critério todos-nós, da técnica estrutural, divide o domínio de entrada em subdomínios não disjuntos formados de todos os casos de teste que fazem com que um determinado comando seja executado. Da mesma forma, todos os outros critérios também poderiam ser vistos dessa maneira, como, por exemplo, os critérios todos-ramos, todos-caminhos, os critérios de fluxo de dados, o teste de mutação, o próprio teste exaustivo (que corresponderia a uma partição em que cada subdomínio é formado de um único elemento) e até mesmo o teste randômico (que seria uma forma degenerada de partição, no qual o subconjunto é o próprio domínio de entrada).

Weyuker e Jeng [431] referem-se ao teste de partição como uma família de estratégias de teste e consideram que, embora matematicamente o termo partição se refira a uma divisão em subconjuntos disjuntos, na prática, isso dificilmente ocorre e os subconjuntos da partição se sobrepõem. O ideal, segundo eles, seria que cada partição contivesse apenas elementos que ou fizessem o programa produzir somente saídas corretas ou somente saídas incorretas, sendo, neste último caso, denominadas partções reveladoras. Assim, segundo os autores, existem várias maneiras de definir a partição de modo a tornar melhor ou pior sua capacidade de detectar defeitos. As técnicas de teste funcional abordam esse problema olhando diretamente o domínio de entrada e procurando estabelecer os subdomínios de interesse.

A seguir, na Seção 2.2 será apresentado um breve histórico sobre o teste funcional, na Seção 2.3 serão comentados os principais critérios que compõem essa técnica de teste e na Seção 2.4 serão apresentadas as considerações finais.

## 2.2 Histórico

Desde os anos 70 encontra-se literatura sobre teste funcional [294, 180, 31]. Nessa época também surgiram vários métodos para apoiar a especificação de sistemas, como a Análise e Projeto Estruturado [147], no qual eram mencionados, embora não diretamente, aspectos de validação do sistema com relação à satisfação dos seus requisitos funcionais. Nesse sentido, algumas técnicas usadas para descrever tais requisitos de forma completa e não ambígua são técnicas que também estão incorporadas em critérios funcionais, como é o caso das Tabelas de Decisão, utilizadas no critério Grafo Causa-Efeito, como será visto neste capítulo.

Roper [344] faz uma discussão sobre a atividade de teste ao longo do ciclo de vida de desenvolvimento, considerando vários aspectos e características dos métodos de análise e projeto em relação ao teste funcional e às demais técnicas abordadas neste livro.

Outro critério do teste funcional, o Particionamento de Equivalência, que a princípio considerava a elaboração de casos de teste baseados apenas no domínio de entrada do programa [294], passou a ser tratado posteriormente de dois pontos de vista, o de entrada e o de saída [344].

Mais recentemente, motivados pela baixa cobertura dos critérios funcionais em relação a outros critérios estruturais e baseados em erros, Linkman et al. [243] propuseram o critério Teste Funcional Sistemático, que tem o objetivo de promover maior cobertura em relação a erros típicos considerados no critério Análise de Mutantes, o qual será abordado no Capítulo 5.

## 2.3 Critérios

Os critérios mais conhecidos da técnica de teste funcional são Particionamento de Equivalência, Análise do Valor Limite, Grafo Causa-Efeito e Error-Guessing. Além desses, também existem outros, como, por exemplo, Teste Funcional Sistemático [243], Syntax Testing, State Transition Testing e Graph Matrix [216].

Como todos os critérios da técnica funcional baseiam-se apenas na especificação do produto testado, a qualidade de tais critérios depende fortemente da existência de uma boa especificação de requisitos. Especificações ausentes ou mesmo incompletas tornarão difícil a aplicação dos critérios funcionais. Além disso, tais critérios também apresentam a limitação de não garantir que partes essenciais ou críticas do produto em teste sejam exercitadas [31].

Por outro lado, os critérios funcionais podem ser aplicados em todas as fases de testes e em produtos desenvolvidos com qualquer paradigma de programação, pois não levam em consideração os detalhes de implementação [92].

A seguir apresenta-se a especificação do programa “Cadeia de Caracteres”, extraída de Roper [344], que será utilizada para exemplificar alguns desses critérios.

Especificação do programa “Cadeia de Caracteres”:

*O programa solicita do usuário um inteiro positivo no intervalo entre 1 e 20 e então solicita uma cadeia de caracteres desse comprimento. Após isso, o programa solicita um caractere e retorna a posição na cadeia em que o caractere é encontrado pela primeira vez ou uma mensagem indicando que o caractere não está presente na cadeia. O usuário tem a opção de procurar vários caracteres.*

### Particionamento em classes de equivalência

Considerando-se que o teste exaustivo é, em geral, impossível de ser aplicado, durante a atividade de teste, o testador fica restrito a usar um subconjunto de todas as possíveis entradas do programa, sendo que esse subconjunto deve ter uma grande probabilidade de encontrar defeitos [294].

Com o objetivo de apoiar a determinação desse subconjunto, tornando a quantidade de dados de entrada finita e mais viável para uma atividade de teste, o particionamento de equivalência divide o domínio de entrada em classes de equivalência que, de acordo com a especificação do programa, são tratadas da mesma maneira. Assim, uma vez definidas as classes de equivalência, pode-se assumir, com alguma segurança, que qualquer elemento da classe pode ser considerado um representante desta, pois todos eles devem se comportar de forma similar, ou seja, se um elemento detectar um defeito, qualquer outro também detecta; se não detectar, os outros também não detectam. Dessa forma, o critério reduz o domínio de entrada a um

tamanho passível de ser tratado durante a atividade de teste. Além do domínio de entrada, alguns autores consideram também o domínio de saída, identificando neste possíveis alternativas que poderiam determinar classes de equivalência no domínio de entrada [344, 92].

Para ajudar na identificação das partições, pode-se observar a especificação procurando termos como “intervalo” e “conjunto” ou palavras similares que indiquem que os dados são processados da mesma forma.

Uma classe de equivalência representa um conjunto de estados válidos ou inválidos para as condições de entrada. As classes podem ser definidas de acordo com as seguintes diretrizes [294]: 1) se a condição de entrada especifica um intervalo de valores, são definidas uma classe válida e duas inválidas; 2) se a condição de entrada especifica uma quantidade de valores, são definidas uma classe válida e duas inválidas; 3) se a condição de entrada especifica um conjunto de valores determinados e o programa pode manipulá-los de forma diferente, é definida uma classe válida para cada um desses valores e uma classe inválida com outro valor qualquer; 4) se a condição de entrada especifica uma situação do tipo “deve ser de tal forma”, são definidas uma classe válida e uma inválida.

Caso seja observado que as classes de equivalência se sobreponem ou que os elementos de uma mesma classe não devem se comportar da mesma maneira, elas devem ser reduzidas a fim de separá-las e torná-las distintas.

Uma vez identificadas as classes de equivalência, devem-se determinar os casos de teste, escolhendo-se um elemento de cada classe, de forma que cada novo caso de teste cubra o maior número de classes válidas possível. Já para as classes inválidas, devem ser gerados casos de testes exclusivos, uma vez que um elemento de uma classe inválida pode mascarar a validação do elemento de outra classe inválida [92].

### **Exemplo de aplicação**

De acordo com a descrição do critério, os dois passos a serem realizados são: i) identificar as classes de equivalência, o que deve ser feito observando-se as entradas e as saídas do programa com o objetivo de particionar o domínio de entrada e; ii) gerar casos de teste selecionando um elemento de cada classe, de forma a ter o menor número de casos de teste possível.

Considerando a especificação dada anteriormente, têm-se quatro entradas:

**T** – tamanho da cadeia de caracteres;

**CC** – uma cadeia de caracteres;

**C** – um caractere a ser procurado;

**O** – a opção por procurar mais caracteres.

De acordo com a especificação, as variáveis CC e C não determinam classes de equivalência, pois os caracteres podem ser quaisquer. Já o tamanho da cadeia deve estar no intervalo entre 1 e 20 (inclusive) e a opção do usuário pode ser “sim” ou “não”.

Considerando o domínio de saída, têm-se duas alternativas:

1. a posição na qual o caractere foi encontrado na cadeia de caracteres;
2. uma mensagem declarando que o caractere não foi encontrado.

Essa informação pode ser usada para fazer outra partição no domínio de entrada: uma contendo caracteres que são encontrados na cadeia e a outra contendo caracteres que não pertencem à cadeia. A Tabela 2.1 apresenta as classes de equivalência identificadas.

Tabela 2.1 – Classes de equivalência da especificação Cadeia de Caracteres

Variável de entrada	Classes de equivalência válidas	Classes de equivalência inválidas
T	$1 \leq T \leq 20$	$T < 1$ e $T > 20$
O	Sim	Não
C	Caractere que pertence à cadeia	Caractere que não pertence à cadeia

Identificadas as classes de equivalência, escolhem-se, arbitrariamente, elementos de cada classe e os casos de teste podem ser definidos como está apresentado na Tabela 2.2.

Tabela 2.2 – Casos de teste para o critério Particionamento de Equivalência

Variáveis de entrada				Saída esperada
T	CC	C	O	
34				entre com um inteiro entre 1 e 20
0				entre com um inteiro entre 1 e 20
3	abc	c		o caractere c aparece na posição 3 da cadeia
			s	
		k		o caractere k não pertence à cadeia
			n	

### Avaliação do critério

A força desse critério está na redução que ele possibilita no tamanho do domínio de entrada e na criação de dados de teste baseados unicamente na especificação. Ele é especialmente adequado para aplicações em que as variáveis de entrada podem ser identificadas com facilidade e assumem valores específicos. No entanto, o critério não é tão facilmente aplicável quando o domínio de entrada é simples, mas o processamento é complexo. Um problema possível de ocorrer é que, embora a especificação possa sugerir que um grupo de dados seja processado de forma idêntica, na prática isso pode não acontecer. Além disso, a técnica não fornece diretrizes para a determinação dos dados de teste e para encontrar combinações entre eles que permitam cobrir as classes de equivalência de maneira mais eficiente [344].

## Análise do valor limite

De acordo com Meyers [294], a experiência mostra que casos de teste que exploram condições limites têm uma maior probabilidade de encontrar defeitos. Tais condições correspondem a valores que estão exatamente sobre ou imediatamente acima ou abaixo dos limitantes das classes de equivalência.

Assim, esse critério é usado em conjunto com o particionamento de equivalência, mas em vez de os dados de teste serem escolhidos aleatoriamente, eles devem ser selecionados de forma que o limitante de cada classe de equivalência seja explorado. Segundo Meyers [294], além da escolha seletiva dos dados de teste, o outro ponto que distingue esse critério do Particionamento de Equivalência é a observação do domínio de saída, diferentemente de outros autores, que já fazem essa consideração no próprio critério de Particionamento de Equivalência.

Embora não existam diretrizes bem definidas que levem à determinação dos dados de teste, Meyers [294] sugere que as seguintes recomendações sejam seguidas:

1. se a condição de entrada especifica um intervalo de valores, devem ser definidos dados de teste para os limites desse intervalo e dados de teste imediatamente subsequentes, que explorem as classes inválidas vizinhas desse intervalo. Por exemplo, se uma classe válida estiver no intervalo  $-1,0$  e  $+1,0$ , devem ser definidos os seguintes dados de teste:  $-1,0; +1,0; -1,001$  e  $+1,001$ ;
2. se a condição de entrada especifica uma quantidade de valores, por exemplo, de 1 a 255 valores, devem ser definidos dados de teste com nenhum valor de entrada, somente um valor, 255 valores e 256 valores de entrada;
3. usar a diretriz 1 para as condições de saída;
4. usar a diretriz 2 para as condições de saída;
5. se a entrada ou saída for um conjunto ordenado, deve ser dada maior atenção aos primeiro e último elementos desse conjunto;
6. usar a intuição para definir outras condições limites.

### Exemplo de aplicação

Considerando as classes de equivalência segundo o critério Particionamento de Equivalência (Tabela 2.1) e lembrando que devem ser explorados os limites tanto do ponto de vista de entrada como de saída, os casos de teste que satisfazem o critério Análise do Valor Limite são apresentados na Tabela 2.3. Observa-se que a linha dupla na tabela denota uma segunda execução do programa, pois só assim todos os casos de teste necessários para satisfazer o critério podem ser exercitados.

Um outro exemplo que caracteriza bem a diferença entre o Particionamento de Equivalência e Análise do Valor Limite é o exemplo do triângulo, dado por Meyers [294]. Para que três valores representem um triângulo, eles devem ser inteiros maiores que zero e a soma de quaisquer desses dois valores deve ser maior que o terceiro valor. Assim, ao definirem-se as classes de equivalência, a válida seria aquela em que essa condição é satisfeita e a classe inválida seria aquela em que a soma de dois dos valores não é maior que o terceiro valor. Para

Tabela 2.3 – Casos de teste para o critério Análise do Valor Limite

Entrada				Saída esperada
T	CC	C	O	
21				entre com um inteiro entre 1 e 20
0				entre com um inteiro entre 1 e 20
1	a	a		o caractere a aparece na posição 1 da cadeia
			s	
		x		o caractere x não pertence à cadeia
			n	
20	abcdefghijklmnopqrstuvwxyz	a		o caractere a aparece na posição 1 da cadeia
			s	
		t		o caractere t aparece na posição 20 da cadeia
			n	

o Particionamento de Equivalência, dois dados de teste seriam 3; 4; 5 e 1; 2; 4 para as classes válida e inválida, respectivamente.

Considerando-se que esse critério deve explorar valores nos limites das classes de equivalência, um dado de teste requerido seria 1; 2; 3, que explora o fato de a soma de dois dos valores ser exatamente igual ao terceiro valor. Dessa forma, se a verificação da condição fosse implementada erroneamente como  $A + B \geq C$  em vez de  $A + B > C$ , somente o dado de teste gerado pela aplicação da análise do valor limite seria capaz de identificar o erro.

### Avaliação do critério

As vantagens e desvantagens do critério Análise do Valor Limite são similares às do critério Particionamento de Equivalência, sendo que existem diretrizes para que os dados de teste sejam estabelecidos, uma vez que estes devem explorar especificamente os limites das classes identificadas.

## Teste funcional sistemático

Este critério combina os critérios Particionamento de Equivalência e Análise do Valor Limite. Uma vez que os domínios de entrada e de saída tenham sido particionados, este critério requer ao menos dois casos de teste de cada partição para minimizar o problema de que defeitos coincidentes mascarem falhas. Além disso, ele também requer a avaliação no limite de cada partição e subsequente a ele [243]. Para facilitar a identificação desses casos de teste, o critério fornece as seguintes diretrizes:

### 1. Valores numéricos

Para o domínio de entrada, selecionar valores de entrada da seguinte forma:

- (a) valores discretos: testar todos os valores; e
- (b) intervalo de valores: testar os extremos e um valor no interior do intervalo.

Para o domínio de saída, selecionar valores de entrada que resultem nos valores de saída que devem ser gerados pelo software. Os tipos das saídas podem não coincidir com os tipos dos valores de entrada; por exemplo, valores de entrada distintos podem produzir um intervalo de valores de saída, dependendo de outros fatores, ou um intervalo de valores de entrada pode produzir somente um ou dois valores de saída como verdadeiro e falso. Assim, devem-se escolher como entrada valores que explorem os valores de saída da seguinte forma:

- (a) valores discretos: gerar cada um deles; e
- (b) intervalo de valores: gerar cada um dos extremos e ao menos um valor no interior do intervalo.

## 2. Tipos de valores diferentes e casos especiais

Tipos diferentes de valores devem ser explorados na entrada e na saída como, por exemplo, espaço em branco, que pode ser interpretado como zero em um campo numérico. Casos especiais como zero sempre devem ser selecionados individualmente, mesmo que ele esteja dentro de um intervalo de valores. O mesmo serve para valores nos limites da representação binária dos dados. Por exemplo, para campos inteiros de 16 bits, os valores -32768 e +32767 deveriam ser selecionados.

## 3. Valores ilegais

Valores que correspondem a entradas ilegais também devem ser incluídos nos casos de teste, para assegurar que o software os rejeita. Deve-se também tentar gerar valores ilegais, os quais não devem ser bem-sucedidos. É importante que sejam selecionados os limites dos intervalos numéricos, tanto inferior quanto superior, valores imediatamente fora dos limites desses intervalos e também os valores imediatamente subsequentes aos limites do intervalo e pertencentes a ele.

## 4. Números reais

Valores reais envolvem mais problemas do que valores inteiros, uma vez que, na entrada, são fornecidos como números decimais, para processamento, são armazenados na forma binária e, como saída, são convertidos em decimais novamente. Verificar o limite para números reais pode não ser exato, mas, ainda assim, essa verificação deve ser incluída como caso de teste. Deve ser definida uma margem de erro de tal forma que, se ultrapassada, então o valor pode ser considerado distinto. Além disso, devem ser selecionados números reais bem pequenos e também o zero.

## 5. Intervalos variáveis

Ocorre quando o intervalo de uma variável depende do valor de outra variável. Por exemplo, suponha que o valor da variável  $x$  pode variar de zero ao valor da variável  $y$  e que  $y$  é um inteiro positivo. Nesse caso, os seguintes dados de entrada devem ser definidos: i)  $x = y = 0$ ; ii)  $x = 0 < y$ ; iii)  $0 < x = y$ ; iv)  $0 < x < y$ . Além disso, devem-se também selecionar os seguintes valores ilegais: i)  $y = 0 < x$ ; ii)  $0 < y < x$ ; iii)  $x < 0$ ; e iv)  $y < 0$ .

## 6. Arranjos

Quando se usa arranjo, tanto como entrada como saída, deve-se considerar o fato de o tamanho do arranjo ser variável bem como os dados serem variáveis. Os elementos

do arranjo devem ser testados como se fossem variáveis comuns, como mencionado no item anterior. Além disso, o tamanho do arranjo deve ser testado com valores intermediários, com os valores mínimo e máximo. Para simplificar o teste, podem-se considerar as linhas e colunas de um arranjo como se fossem subestruturas, a serem testadas em separado. Assim, o arranjo deve ser testado primeiro como uma única estrutura, depois como uma coleção de subestruturas, e cada subestrutura deve ser testada independentemente.

#### 7. Dados tipo texto ou string

Neste caso o dado de entrada deve explorar comprimentos variáveis e também validar os caracteres que o compõem, pois algumas vezes os dados de entrada podem ser apenas alfabéticos, outras vezes, alfanuméricos, e algumas vezes podem possuir caracteres especiais.

Além desses casos, como foi dito inicialmente, o critério Funcional Sistemático requer que dois elementos de uma mesma classe de equivalência sejam explorados para evitar erros do seguinte tipo: suponha que o programa receba um valor de entrada e produza seu quadrado. Se o valor de entrada for 2, e o valor produzido como saída for 4, embora o resultado esteja correto, ainda assim não se pode afirmar qual das operações o programa realizou, pois poderia ser  $2 * 2$  como também  $2 + 2$ .

#### Exemplo de aplicação

Como o critério Funcional Sistemático estende os critérios Particionamento de Equivalência e Análise do Valor Limite, os casos de testes definidos nas Tabelas 2.2 e 2.3 deveriam ser selecionados.

Além desses, os casos de teste da Tabela 2.4 seriam também desenvolvidos.

Tabela 2.4 – Casos de teste para o critério Funcional Sistemático

Entrada				Saída esperada
T	CC	C	O	
a				entre com um inteiro entre 1 e 20
1.0				entre com um inteiro entre 1 e 20
1	!	,	n	o caractere ' ' não pertence à cadeia
1	}	~	n	o caractere ~ não pertence à cadeia
20	!"#\$%&()*+/'01234567	!	s	o caractere ! aparece na posição 1 da cadeia
		"	s	o caractere " aparece na posição 2 da cadeia
		+	s	o caractere + aparece na posição 10 da cadeia
		6	s	o caractere 6 aparece na posição 19 da cadeia
		7	n	o caractere 7 aparece na posição 20 da cadeia
2	ab	b	nao	o caractere b aparece na posição 2 da cadeia
3	a2b	2	0	o caractere 2 aparece na posição 2 da cadeia

### Avaliação do critério

Por ser baseado nos critérios Particionamento de Equivalência e Análise do Valor Limite, o critério Funcional Sistemático apresenta os mesmos problemas que estes. Entretanto, assim como o critério Análise do Valor Limite, este critério fornece diretrizes para facilitar a geração de casos de testes e enfatiza mais fortemente a seleção de mais de um caso de teste por partição e/ou limite, aumentando assim a chance de revelar os defeitos sensíveis a dados e também a probabilidade de obter uma melhor cobertura do código do produto que está sendo testado.

### Grafo Causa-Efeito

Uma das limitações dos critérios anteriores é que eles não exploram combinações dos dados de entrada. Já o critério Grafo Causa-Efeito ajuda na definição de um conjunto de casos de teste que exploram ambigüidades e incompletude nas especificações. O grafo é uma linguagem formal na qual a especificação é traduzida e o processo para derivar casos de teste a partir desse critério pode ser resumido nos seguintes passos [294]:

1. Dividir a especificação do software em partes, pois a construção do grafo para grandes especificações torna-se bastante complexa.
2. Identificar as causas e efeitos na especificação. As causas correspondem às entradas, estímulos, ou qualquer coisa que provoque uma resposta do sistema em teste e os efeitos correspondem às saídas, mudanças no estado do sistema ou qualquer resposta observável. Uma vez identificados, a cada um deve ser atribuído um único número.
3. Analisar a semântica da especificação e transformar em um grafo booleano – o Grafo Causa-Efeito – que liga as causas e os efeitos.
4. Adicionar anotações ao grafo, as quais descrevem combinações das causas e efeitos que são impossíveis devido a restrições sintáticas ou do ambiente.
5. Converter o grafo em uma tabela de decisão, na qual cada coluna representa um caso de teste.
6. Converter as colunas da tabela de decisão em casos de teste.

Considerando que cada nó do grafo pode assumir os valores 0 ou 1, que representam, respectivamente, ausência ou presença do estado, a notação utilizada no Grafo Causa-Efeito é composta dos operadores apresentados na Figura 2.1 e descritos a seguir [294].

- Função identidade: se nó “1” é 1, então nó “2” é 1; senão nó “2” é 0.
- Função not: se nó “1” é 1, então nó “2” é 0; senão nó “2” é 1.
- Função or: se nó “1” ou “2” ou “3” é 1, então nó “4” é 1; senão nó “4” é 0.
- Função and: se ambos nós “1” e “2” são 1, então nó “3” é 1; senão nó “3” é 0.

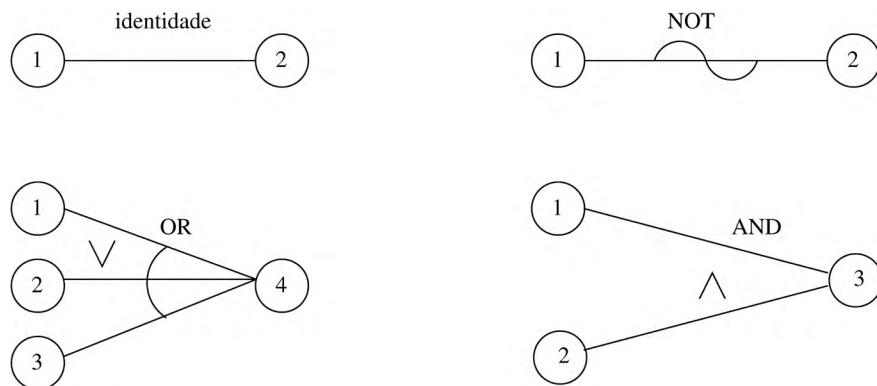


Figura 2.1 – Notação dos Grafos Causa-Efeito.

Para ilustrar a combinação desses operadores em um grafo, considere a especificação do programa “Imprime Mensagens”: o programa lê dois caracteres e, de acordo com eles, mensagens serão impressas da seguinte forma: o primeiro caractere deve ser um “A” ou um “B”. O segundo caractere deve ser um dígito. Nessa situação, o arquivo deve ser atualizado. Se o primeiro caractere é incorreto, enviar a mensagem X. Se o segundo caractere é incorreto, enviar a mensagem Y [294].

As causas são:

- 1 - caractere na coluna 1 é “A”;
- 2 - caractere na coluna 1 é “B”;
- 3 - caractere na coluna 2 é um dígito.

Os efeitos são:

- 70 - a atualização é realizada;
- 71 - a mensagem X é enviada;
- 72 - a mensagem Y é enviada.

O Grafo Causa-Efeito correspondente é apresentado na Figura 2.2. Para verificar se ele corresponde à especificação basta atribuir os possíveis valores 0 e 1 às causas e observar se os efeitos assumem os valores corretos.

Apesar de o grafo da Figura 2.2 corresponder ao exemplo, ele contém uma combinação de causas que é impossível de ocorrer, uma vez que “1” e “2” não podem possuir o valor 1 simultaneamente. Esta e outras situações ocorrem comumente na prática. Para contornar esses problemas a notação da Figura 2.3 pode ser utilizada, sendo que elas representam o seguinte:

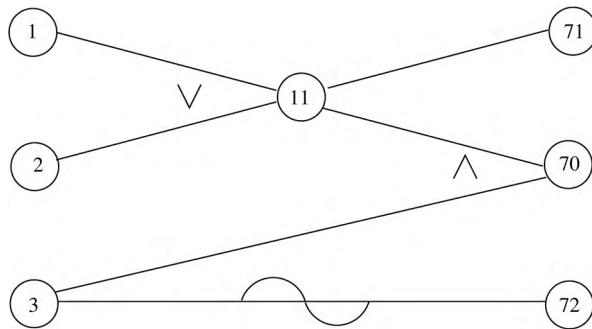


Figura 2.2 – Grafo Causa-Efeito da especificação imprime mensagens.

- Restrição E: no máximo um entre “1” e “2” pode ser igual a 1 (ou seja, “1” e “2” não podem ser 1 simultaneamente).
- Restrição I: no mínimo um entre “1”, “2” e “3” deve ser igual a 1 (ou seja, “1”, “2” e “3” não podem ser 0 simultaneamente).
- Restrição O: um e somente um entre “1” e “2” deve ser igual a 1.
- Restrição R: para que “1” seja igual a 1, “2” deve ser igual a 1 (ou seja, é impossível que “1” seja 1 se “2” for 0).
- Restrição M: se o efeito “1” é 1 o efeito “2” é forçado a ser 0.

Assim, considerando a representação dessas restrições e considerando também ser impossível que as causas “1” e “2” estejam presentes simultaneamente, mas que é possível ambas não estarem presentes, o grafo da Figura 2.2 é refeito para denotar essa situação e é apresentado na Figura 2.4.

O próximo passo é estudar sistematicamente o grafo e construir uma tabela de decisão. Essa tabela mostra os efeitos (ações) que ocorrem para todas as possíveis combinações de causas (condições). Portanto, se podem ocorrer n causas, a tabela de decisão contém  $2^n$  entradas.

Meyers [294] define o seguinte procedimento para elaborar a tabela de decisão:

1. Selecionar um efeito para estar no estado presente, isto é, com valor 1.
2. Rastrear o grafo para trás, encontrando todas as combinações de causas (sujeitas a restrições) que fazem com que esse efeito seja 1.
3. Criar uma coluna na tabela de decisão para cada combinação de causa.
4. Determinar, para cada combinação, os estados de todos os outros efeitos, anotando na tabela.

Ao executar o passo 2, fazer as seguintes considerações:

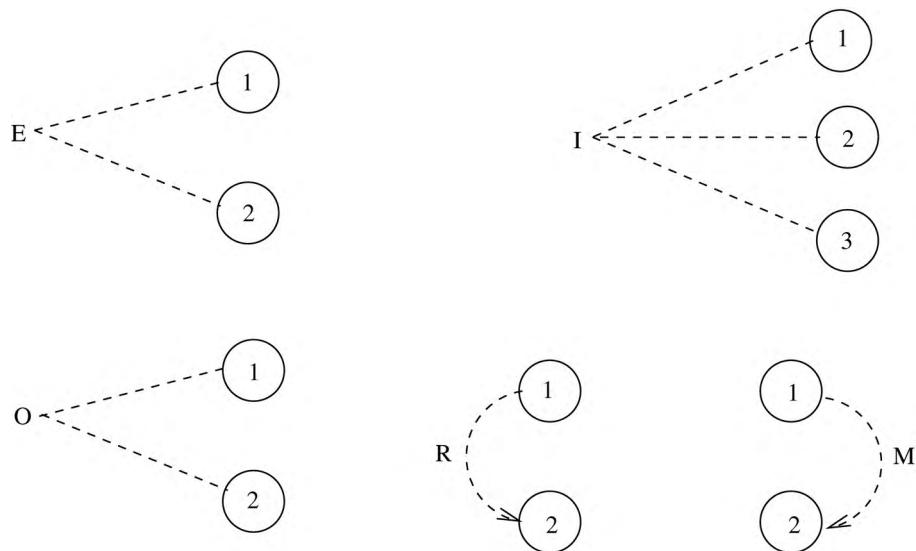


Figura 2.3 – Notação de restrições do Grafo Causa-Efeito.

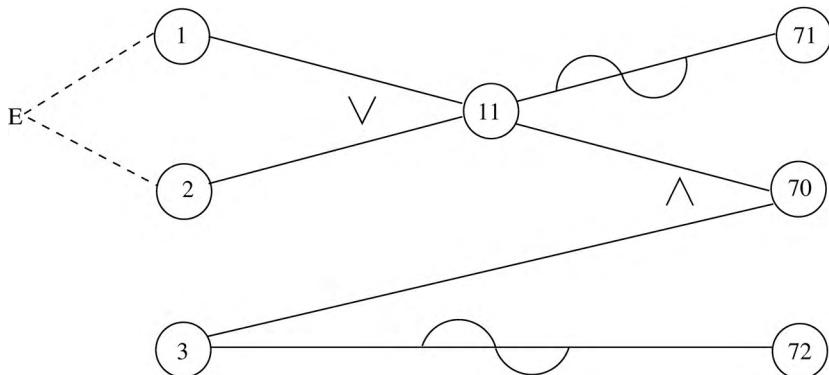


Figura 2.4 – Grafo Causa-Efeito com restrição da especificação imprime mensagens.

1. Quando o nó for do tipo OR e a saída deva ser 1, nunca atribuir mais de uma entrada com valor 1 simultaneamente. O objetivo disso é evitar que alguns erros não sejam detectados pelo fato de uma causa mascarar outra.
2. Quando o nó for do tipo AND e a saída deva ser 0, todas as combinações de entrada que levem à saída 0 devem ser enumeradas. No entanto, se a situação é tal que uma entrada é 0 e uma ou mais das outras entradas é 1, não é necessário enumerar todas as condições em que as outras entradas sejam iguais a 1.

3. Quando o nó for do tipo AND e a saída deva ser 0, somente uma condição em que todas as entradas sejam 0 precisa ser enumerada. (Se esse AND estiver no meio do grafo, de forma que suas entradas estejam vindo de outros nós intermediários, pode ocorrer um número excessivamente grande de situações nas quais todas as entradas sejam 0.)

Seguindo essas diretrizes, a tabela de decisão para o grafo apresentado na Figura 2.4 é apresentada na Figura 2.5.

1	0	0	1	0	1
2	0	0	0	1	0
3	0	1	1	1	0
70	0	0	1	1	0
71	1	1	0	0	0
72	1	0	0	0	1

Figura 2.5 – Tabela de decisão do exemplo da Figura 2.4.

O passo final é converter cada coluna da tabela de decisão em um caso de teste, como será visto no exemplo a seguir.

#### Exemplo de aplicação

Considerando-se a especificação do programa “Cadeia de Caracteres” que vem sendo usada para exemplificar os critérios anteriores, podem ser identificadas as seguintes causas:

1. inteiro positivo no intervalo 1-20;
2. caractere a ser procurado na cadeia;
3. procurar outro caractere.

e os seguintes efeitos:

20. inteiro fora do intervalo;
21. posição do caractere na cadeia;
22. caractere não encontrado;
23. término do programa.

O Grafo Causa-Efeito desse exemplo é apresentado na Figura 2.5, e a respectiva tabela de decisão, na Figura 2.6.

Agora, transformam-se as informações extraídas da tabela de decisão em casos de teste, que estão apresentados na Tabela 2.5. Por exemplo, o 0 na causa 1, na primeira coluna, significa que um inteiro positivo no intervalo 1-20 não está presente, ou seja, o valor de T deve ser diferente de um inteiro nesse intervalo, e assim por diante.

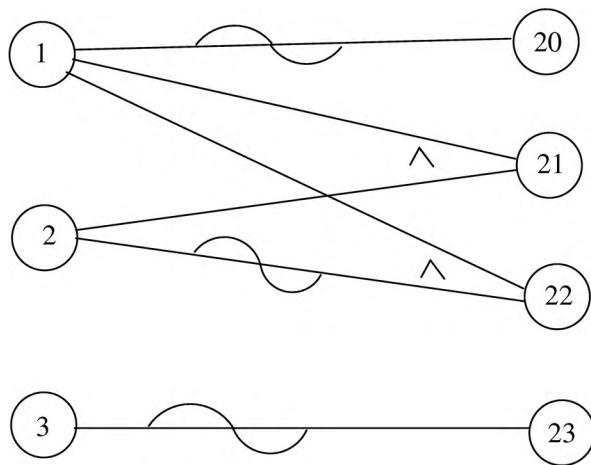


Figura 2.6 – Grafo Causa-Efeito da especificação Cadeia de Caracteres.

	1	0	1	1	-
	2	-	1	0	-
	3	-	1	1	0
20		1	0	0	0
21		0	1	0	0
22		0	0	1	0
23		0	0	0	1

Figura 2.7 – Tabela de decisão do exemplo da Figura 2.6.

### Avaliação do critério

A vantagem do Grafo Causa-Efeito é que esse critério exerce combinações de dados de teste que, possivelmente, não seriam considerados. Além disso, os resultados esperados são produzidos como parte do processo de criação do teste, ou seja, eles fazem parte da própria tabela de decisão. As dificuldades do critério estão na complexidade em se desenvolver o grafo booleano, caso o número de causas e efeitos seja muito grande, e também na conversão do grafo na tabela de decisão, embora esse processo seja algorítmico e existam ferramentas de auxílio. Uma possível solução para isso é tentar identificar subproblemas e desenvolver subgrafos Causa-Efeito para eles. A eficiência desse critério, assim como os outros critérios funcionais, depende da qualidade da especificação, para que sejam identificadas as causas e os efeitos e, também, da habilidade e experiência do testador. Uma especificação muito detalhada pode levar a um grande número de causas e efeitos e, por outro lado, uma especificação muito abstrata pode não gerar dados de teste significativos.

Tabela 2.5 – Dados de teste para o Grafo Causa-Efeito

Entrada				Saída esperada
T	CC	C	O	
23				entre com um inteiro entre 1 e 20
3	abc	c		o caractere c aparece na posição 3
			s	
		x		o caractere x não pertence à cadeia
		k		o caractere k não ocorre na string fornecida
		n		

## Error Guessing

Essa técnica corresponde a uma abordagem *ad-hoc* na qual a pessoa pratica, inconscientemente, uma técnica para projeto de casos de teste, supondo por intuição e experiência alguns tipos prováveis de erros e, a partir disso, definem-se casos de teste que poderiam detectá-los [294]. A idéia do critério é enumerar possíveis erros ou situações propensas a erros e então definir casos de teste para explorá-las. Por exemplo, se o que está sendo testado é um módulo de ordenação, as seguintes situações poderiam ser exploradas: i) uma lista de entrada vazia; ii) uma lista com apenas uma entrada; iii) todas as entradas com o mesmo valor; e iv) uma lista de entrada já ordenada.

## 2.4 Considerações finais

Uma vantagem dos critérios da técnica funcional é que eles requerem somente a especificação do produto para derivar os requisitos de teste, podendo, dessa forma, ser aplicados indistintamente a qualquer programa, seja ele procedural ou orientado a objetos, ou a componentes de software, uma vez que o código fonte não é necessário. Por outro lado, segundo Roper [344], como os critérios funcionais se baseiam apenas na especificação, eles não podem assegurar que partes críticas e essenciais do código tenham sido cobertas.

Além disso, os próprios critérios apresentam algumas limitações, como já foi comentado anteriormente. Embora se espere, por exemplo, que os elementos de uma mesma classe de equivalência se comportem da mesma maneira, na prática isso pode não ocorrer. Dessa forma, é fundamental que as técnicas de teste sejam entendidas como complementares e sejam utilizadas em conjunto para que o software seja explorado de diferentes pontos de vista.

Para ilustrar esse fato, o exemplo utilizado neste capítulo foi implementado em C e os casos de teste gerados para atender os critérios funcionais foram usados para avaliar a cobertura em relação ao critério Análise de Mutantes, o qual será apresentado no Capítulo 5. Essa avaliação foi realizada utilizando-se a ferramenta Proteum [107] e os casos de teste foram explorados de duas maneiras diferentes: isoladamente, isto é, calculou-se o escore de mutação em relação à aplicação de cada conjunto de casos de teste funcional e calculou-se o escore de mutação adicionando-se ao conjunto de casos de teste de um dos critérios os casos de teste

adequados a outro critério funcional, de forma cumulativa. Na Tabela 2.6 apresentam-se esses dados. Foi gerado um total de 863 mutantes, dos quais 66 são equivalentes. Recomenda-se que, após a leitura do Capítulo 5, o leitor volte a avaliar esses resultados.

Tabela 2.6 – Cobertura dos critérios funcionais em relação à Análise de Mutantes

Critério	Cobertura critérios individuais			Cobertura critérios cumulativos		
	Vivos	Escore	Casos de Teste	Vivos	Escore	Casos de Teste
Particionamento	84	0,894	1	84	0,894	1
Análise Valor Limite	54	0,932	2	54	0,932	3
Grafo Causa-Efeito	102	0,872	1	54	0,932	4
Teste Sistemático	14	0,982	10	14	0,982	14

Aplicando-se os conjuntos de casos de teste separadamente, o menos efetivo para distinguir os mutantes foi o conjunto do critério Grafo Causa-Efeito, enquanto de forma cumulativa o menos efetivo foi o do Particionamento de Equivalência. Independentemente da forma de aplicação, o mais efetivo foi o conjunto gerado a partir do Teste Sistemático, como era de se esperar, visto que ele foi proposto justamente com a intenção de melhorar a cobertura conseguida com conjuntos de teste funcionais em relação às outras técnicas de teste.

# Capítulo 3

## Teste Baseado em Modelos

*Adenilso da Silva Simão (ICMC/USP)*

### 3.1 Introdução

Uma especificação é um documento que representa o comportamento e as características que um sistema deve possuir. Ela pode ser definida de diversas formas. Por exemplo, pode-se criar um documento que descreve textualmente o comportamento em uma linguagem natural (português, inglês, etc.). Um risco potencial com a descrição textual é a possibilidade de se incluírem inconsistências na especificação, uma vez que as linguagens naturais geralmente são ambíguas e imprecisas. O uso de outras formas de especificação torna-se importante em contextos nos quais tais imprecisões podem causar problemas.

É importante notar que o uso de linguagem natural não é, por si só, um problema. Afinal, o que é necessário é a precisão e o rigor da especificação. Por exemplo, uma descrição textual feita de forma rigorosa e clara pode ser mais fácil de se compreender e implementar do que uma especificação formal complexa. A especificação formal só é útil se for possível, a partir dela, aumentar a compreensão do sistema, aliando-se a possibilidade de analisar algorítmicamente a especificação. Por algorítmicamente, entenda-se que se trata de uma forma sistemática de verificar as propriedades da especificação.

A modelagem permite que o conhecimento sobre o sistema seja capturado e reutilizado durante diversas fases de desenvolvimento. Boa parte da atividade de teste é gasta buscando-se identificar o que o sistema deveria fazer. Ou seja, antes de se perguntar se o resultado está correto, deve-se saber qual seria o resultado correto. Um modelo é muito importante nessa tarefa, pois, se bem desenvolvido, capture o que é essencial no sistema. No caso de modelos que oferecem a possibilidade de execução (ou simulação, como é muitas vezes o caso), o modelo pode ser utilizado como um oráculo, definindo a linha que separa o comportamento adequado do comportamento errôneo. Contudo, utilizar o modelo como oráculo coloca a seguinte questão: como saber que o modelo está correto? Ou seja, o modelo torna-se um artefato que deve ser testado tanto quanto o próprio sistema. Uma vez verificado o problema de o modelo estar ou não adequado (ou, ao menos, aumentada a confiança de que ele esteja), ele é extremamente valioso para o teste, servindo tanto como oráculo quanto como base para a geração de scripts e cenários de teste.

Durante a realização dos testes, um grande obstáculo é determinar exatamente quais são os objetivos de teste, ou seja, quais itens serão testados e como. Em geral, especificações não rigorosas deixam grande margem a opiniões e especulações. A forma da especificação pode variar de um grafo de fluxos de chamadas intermódulos a um guia de usuário. Uma especificação clara é importante para definir o escopo do trabalho de desenvolvimento e, em especial, o de teste. Se a única definição precisa de o que o sistema deve fazer é o próprio sistema, os testes podem ser improdutivos. Considere um exemplo apresentado por Apfelbaum [15], no qual a seguinte frase é apresentada em um documento de requisitos: “Se um dígito inválido é fornecido, ele deve ser tratado da maneira adequada.” O que vem a ser uma ‘maneira adequada’ não é definida. Um desenvolvedor pode julgar que a maneira adequada é permitir que o usuário tente digitar novamente. Outro desenvolvedor pode julgar que o mais adequado é abortar o comando. Qualquer uma das duas implementações são plausíveis, mas apenas uma delas deve ser utilizada. O testador fica em uma posição na qual recusa tudo que não está de acordo com o próprio julgamento, ou aceita tudo o que pode estar correto no julgamento de alguém. O leitor pode argumentar que a situação é artificial, mas, ainda assim, esse tipo de sentença em especificações não é incomum.

A criação de modelos não é uma nova habilidade que deve ser adquirida. Os testadores sempre constroem um modelo, ainda que informal. (De outra forma, não haveria como determinar se o comportamento foi aceitável.) O ponto é qual é o formato do modelo. Um modelo informal ou que está apenas na cabeça do testador dificulta a automatização dos testes. Para escrever um script de teste ou um plano de teste, o testador deve entender os passos básicos necessários para usar o sistema.

As seqüências de ações que podem ocorrer durante o uso do sistema são definidas. Em geral, existe mais de uma opção de “próximas ações” que podem ocorrer em um ponto específico do processo. Algumas técnicas permitem descrever quais são as próximas ações em um grafo nos quais os nós representam as configurações (ou estados) e as arestas representam as transições entre as configurações. Essas técnicas são coletivamente denominadas “Máquinas de Transições de Estados”. 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 incluem o uso de variáveis e condicionais, fazendo com que as transições dependam de variáveis ou do contexto atual do sistema.

Existem diversas técnicas baseadas em Máquinas de Transições de Estados. Elas diferem entre si em características relativas à forma como certos elementos são explícita ou implicitamente representados. O modelo mais simples são as Máquinas de Estados Finitos (MEFs), que serão usadas como base para a apresentação dos conceitos deste capítulo. Em uma MEF, as configurações e as transições são representadas explicitamente. Entretanto, para sistemas que possuem um grande número de possíveis configurações distintas, uma MEF talvez não seja adequada, pois o número de estados pode ser excessivamente grande. Outras técnicas foram propostas para resolver esse problema (conhecido como problema da explosão de estados). Por exemplo, uma MEF estendida adiciona às MEFs tradicionais conceitos de variáveis de contexto e transições parametrizadas. Uma configuração particular do sistema não é mais explicitamente representada no modelo. Em vez disso, uma configuração é composta pelo estado atual da MEF e os valores das variáveis de contexto.

O propósito deste capítulo é apresentar algumas técnicas para a condução de testes baseados em modelos. O capítulo será centrado em Máquinas de Transições de Estados e apre-

sentará algumas técnicas desenvolvidas para se gerarem casos de teste a partir de um modelo formal. Tradicionalmente, essas técnicas são utilizadas no teste de protocolos. Contudo, elas podem ser utilizadas em outros contextos, fazendo-se a devida avaliação de custo/benefício. Por exemplo, a especificação do comportamento esperado de um sistema pode ser considerada um protocolo que, se a implementação estiver em conformidade, permitirá ao sistema ser utilizado corretamente e ao usuário se comunicar adequadamente com o sistema.

Este capítulo está organizado da seguinte forma. Na Seção 3.2, são apresentados os principais conceitos das MEFs, assim como suas propriedades. Na Seção 3.3, definem-se as principais seqüências básicas utilizadas pelos métodos de geração de seqüências de teste. Na Seção 3.4, apresentam-se alguns métodos de geração de seqüências de teste, buscando ilustrar as diferenças entre eles e enfatizando o papel das seqüências básicas. Na Seção 3.5, os métodos apresentados são comparados, permitindo identificar o custo/benefício de cada um. Por fim, na Seção 3.6, são feitas as considerações finais deste capítulo.

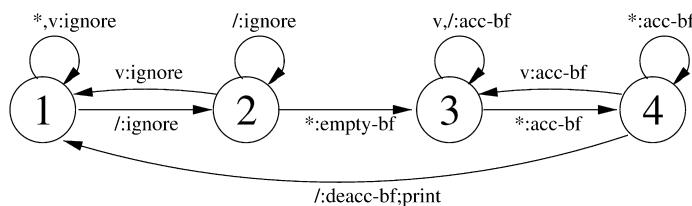
## 3.2 Máquinas de Estados Finitos

Uma Máquina de Estados Finitos [152] é uma máquina hipotética composta por estados e transições. Cada transição liga um estado  $a$  a um estado  $b$  ( $a$  e  $b$  podem ser o mesmo estado). A cada instante, uma máquina pode estar em apenas um de seus estados. Em resposta a um evento de entrada, a máquina gera um evento de saída e muda de estado. Tanto o evento de saída gerado quanto o novo estado são definidos unicamente em função do estado atual e do evento de entrada [105]. Uma Máquina de Estados Finitos pode ser representada tanto por um diagrama de transição de estados quanto por uma tabela de transição. Em um diagrama de transição de estados, os estados são representados por círculos e as transições são representadas por arcos direcionados entre estados. Cada arco é rotulado com a entrada que gera a transição e a saída que é produzida, usando o formato *entrada:saída*. Em uma tabela de transição, os estados são representados por linhas, e as entradas, por colunas [105].

**Definição 1.** Formalmente, uma Máquina de Estados Finitos é uma tupla  $M = (X, Z, S, s_0, f_z, f_s)$ , sendo que

- $X$  é um conjunto finito não-vazio de símbolos de entrada;
- $Z$  é um conjunto finito de símbolos de saída;
- $S$  é um conjunto finito não-vazio de estados;
- $s_0 \in S$  é o estado inicial;
- $f_z : (S \times X) \rightarrow Z$  é a função de saída;
- $f_s : (S \times X) \rightarrow S$  é a função de próximo estado.

A Figura 3.1 apresenta um exemplo, extraído de [79], de uma Máquina de Estados Finitos que especifica o comportamento de um extrator de comentários (*Comment Printer*). O conjunto de entrada consiste em uma seqüência composta pelos símbolos ‘\*’, ‘/’ e ‘v’, sendo que ‘v’ representa qualquer caractere diferente de ‘\*’ e ‘/’. A entrada é uma cadeia de



Estado \ Entrada	*	/	v	*	/	v
1	ignore	ignore	ignore	1	2	1
2	empty-bf	ignore	ignore	3	2	1
3	acc-bf	acc-bf	acc-bf	4	3	3
4	acc-bf	deacc-bf; print	acc-bf	4	1	3

Figura 3.1 – Máquina de Estados Finitos: diagrama e tabela de transição de estados [79].

Tabela 3.1 – Operações da Máquina de Estados Finitos da Figura 3.1

Operação	Ação
<b>ignore</b>	ação nula
<b>empty-bf</b>	buffer := <>
<b>acc-bf</b>	buffer := buffer concatenado com o caractere corrente
<b>deacc-bf</b>	buffer := buffer com o caractere mais à direita truncado
<b>print</b>	imprime o conteúdo do buffer

caracteres e apenas os comentários são impressos (usando a sintaxe da linguagem C). Um comentário é qualquer cadeia de caracteres entre ‘/\*’ e ‘\*/’. As operações usadas no exemplo são apresentadas na Tabela 3.1.

Uma vez que, pela própria definição, o conjunto de estados atingíveis em uma Máquina de Estados Finitos é finito, é possível responder a quase todas as questões sobre o modelo, o que permite a automatização do processo de validação. No entanto, a classe de sistemas que podem ser modelados por uma máquina de estados finitos é consideravelmente limitada. Mesmo para os casos nos quais a modelagem é possível, o modelo resultante pode ser excessivamente grande. Por exemplo, a impossibilidade da representação explícita da concorrência leva à explosão combinatória do número de estados. Com o propósito de aumentar o poder de modelagem, foram propostas várias extensões às Máquinas de Estados Finitos, tais como Statecharts [163] e Estelle [250, 52].

Para o restante de capítulo, é importante definir algumas propriedades que uma MEF pode satisfazer. Essas propriedades são apresentadas a seguir.

**Definição 2.** Diz-se que a MEF é completamente especificada se ela trata todas as entradas em todos os estados. Caso contrário, ela é parcialmente especificada. Para máquinas parcialmente especificadas o teste de conformidade é fraco. Caso contrário, ele é forte. Quando

as transições não especificadas de uma MEF são completadas com fins de realização dos testes, diz-se que assume uma suposição de completude.

**Definição 3.** Uma MEF é fortemente conectada se para cada par de estados  $(s_i, s_j)$  existe um caminho por transições que vai de  $s_i$  a  $s_j$ . Ela é inicialmente conectada se a partir do estado inicial é possível atingir todos os demais estados da MEF.

**Definição 4.** Uma MEF é minimal (ou reduzida) se na MEF não existem quaisquer dois estados equivalentes. Dois estados são equivalentes se possuem as mesmas entradas e produzem as mesmas saídas.

**Definição 5.** Uma MEF é determinística quando, para qualquer estado e para uma dada entrada, a MEF permite uma única transição para um próximo estado. Caso contrário, ela é não-determinística.

## 3.3 Seqüências básicas

Os métodos de geração de casos de teste são fortemente baseados em algumas seqüências básicas, as quais são utilizadas para se obter um resultado parcial importante. Nesta seção, serão apresentadas as seqüências mais importantes para o restante do capítulo.

### 3.3.1 Seqüências de sincronização

Uma seqüência de sincronização para um estado  $s$  (expressa por  $SS(s)$ ) é uma seqüência de símbolos de entrada que leva a MEF ao estado  $s$ , independentemente do estado original da MEF. Por exemplo, considerando a MEF da Figura 3.1, uma seqüência de sincronização para o estado 1 é  $\langle *, *, /, v \rangle$ . Para verificar isso, observe que, não importa qual estado seja escolhido como origem, a aplicação dessa seqüência leva sempre ao estado 1, como apresentado na Tabela 3.2. Observe que a tabela indica os estados que são atingidos após a aplicação de cada entrada da seqüência.

Tabela 3.2 – Aplicação da seqüência de sincronização para o estado 1

	*	*	/	v
1	1	1	2	<b>1</b>
2	3	4	1	<b>1</b>
3	4	4	1	<b>1</b>
4	4	4	4	<b>1</b>

A seqüência de sincronização é útil em situações nas quais se deseja garantir que a MEF vá para um estado em particular. Por exemplo, uma seqüência de sincronização pode ser utilizada para posicionar a MEF no estado inicial sem a necessidade de uma operação de *reset*. Contudo, deve-se ter em mente que a seqüência de sincronização da MEF não é necessariamente uma seqüência de sincronização para a implementação, uma vez que a implementação pode conter falhas.

O cálculo das seqüências de sincronização pode ser feito com o conceito de *conjunto de estados* que representa um dos possíveis estados em que a MEF pode estar. Por exemplo, um conjunto  $\{3, 4\}$  indica que a MEF está no estado 3 ou no estado 4. Pode-se definir uma função  $f'_s : \mathcal{PS} \times X \rightarrow \mathcal{PS}$ , que, dados um conjunto de estados e uma entrada, calcula o conjunto de estados resultante, da seguinte forma:

$$f'_s(I, x) = \{s' \mid s \in I \wedge f_s(s, x) = s'\}$$

Por exemplo, pode-se verificar que  $f'_s(\{3, 4\}, *) = \{4\}$ .

É possível redefinir uma seqüência de sincronização para um estado  $s$  como

$$f'_s(S, SS(s)) = \{s\}$$

Assim, a seqüência  $\langle *, *, /, v \rangle$  é uma seqüência de sincronização para 4 porque  $f'_s(\{1, 2, 3, 4\}, \langle *, *, /, v \rangle) = \{4\}$ .

As seqüências de sincronização podem ser calculadas por meio da construção de uma árvore especial, denominada *árvore de sincronização*. Em uma árvore de sincronização cada nó representa um conjunto de estados. A raiz da árvore é o conjunto de estados formado por todos os estados  $S$ . Para cada símbolo de entrada  $x$ , um nó que representa o conjunto de estados  $A$  terá um filho que representa o conjunto de estados  $B = f'_s(A, x)$ . Uma seqüência de sincronização para um estado  $s$  é formada pelos símbolos de entrada no caminho que vai da raiz a um nó com um conjunto de estados  $\{s\}$ . A Figura 3.2 apresenta a árvore de sincronização para a MEF da Figura 3.1. Pode-se verificar, na árvore de sincronização, como a seqüência de sincronização para o estado 1 foi calculada.

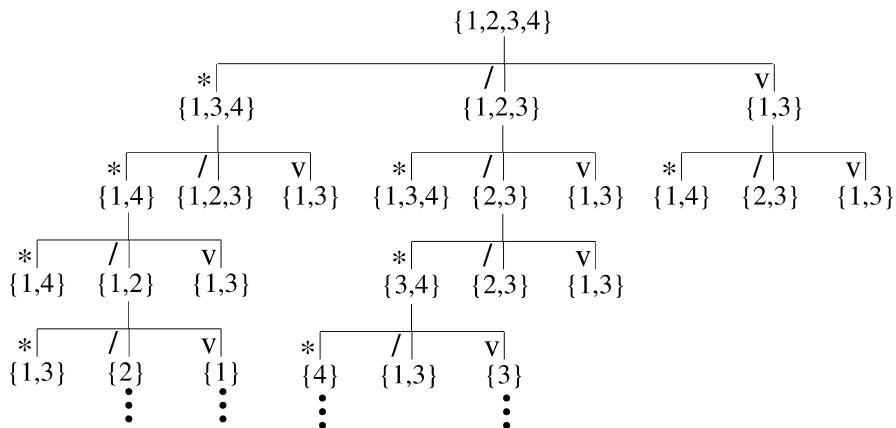


Figura 3.2 – Árvore de sincronização para a MEF da Figura 3.1.

### 3.3.2 Seqüências de distinção

Um seqüência de distinção (DS) é uma seqüência de símbolos de entrada que, quando aplicada aos estados da MEF, produzem saídas distintas para cada um dos estados. Ou seja, observando-se a saída produzida pela MEF como resposta à DS, pode-se determinar em qual estado a MEF estava originalmente.

Para a MEF da Figura 3.1, que foi utilizada até aqui para ilustrar os conceitos apresentados, não existe uma seqüência de distinção. Será utilizado outro exemplo para ilustrar os conceitos e apresentar o algoritmo para o cálculo de uma seqüência de distinção. Considere a MEF da Figura 3.3. Uma DS para essa MEF é uma seqüência  $\langle 0, 0, 1 \rangle$ , como pode ser verificado pela Tabela 3.3. Cada estado é representado por uma linha e cada coluna representa uma entrada da seqüência de distinção. As células representam as saídas produzidas. Observe que cada um dos estados produz uma saída distinta para essa seqüência. Por exemplo, se a saída for  $\langle 1, 0, 1 \rangle$ , pode-se ter certeza de que o estado original era 3 (apesar de a MEF não estar mais nesse estado ao final da aplicação da DS). Obviamente, qualquer seqüência que tenha uma DS como prefixo também é uma DS (já que todos os estados foram distintos, posso concatenar qualquer seqüência de símbolos de entrada, e a seqüência continuará fazendo a distinção). Uma DS é própria se nenhum de seus prefixos é também uma DS. Em geral, tem-se interesse em uma DS própria.

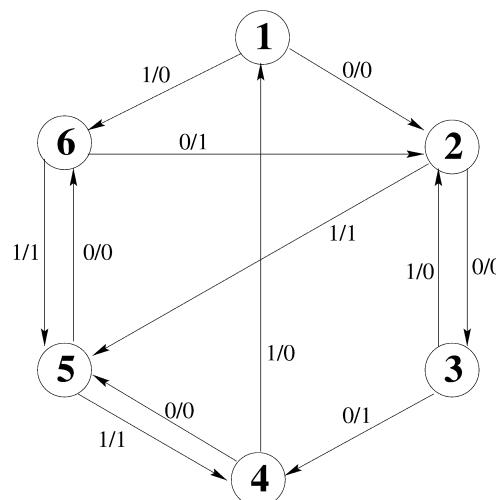


Figura 3.3 – Máquina de Estado Finito.

Tabela 3.3 – Saídas geradas pela MEF da Figura 3.3 em resposta à seqüência de distinção  $\langle 0, 0, 1 \rangle$

	0	0	1
1	0	0	0
2	0	1	0
3	1	0	1
4	0	0	1
5	0	1	1
6	1	0	0

Como pôde ser observado com a MEF da Figura 3.1, uma DS pode não existir. Isto é, não existe uma seqüência capaz de distinguir todos os estados. Existem outras seqüências

que podem ser utilizadas nesse caso, como as UIO e o conjunto W, apresentados nas seções seguintes.

Para o cálculo de uma DS, pode-se utilizar o conceito de grupos de incerteza. Um grupo de incerteza é um conjunto de estados que ainda não foram distinguidos e para os quais ainda é necessário que alguma entrada seja anexada à seqüência para ela se tornar uma DS. Por exemplo, um grupo  $\{\{2, 3, 5, 6\}, \{2, 4\}\}$  indica que os estados 2, 3, 5 e 6 precisam ser distinguidos, assim como os estados 2 e 4. Um grupo é trivial se todos os conjuntos de estados são unitários ou vazios.

Um grupo  $g$  pode ser particionado em relação a um símbolo de entrada  $x$ , criando um grupo  $g'$  no qual todos os conjuntos de estados são subconjuntos dos conjuntos em  $g$ . No particionamento, são mantidos juntos os estados que produzem a mesma saída em relação a  $x$ . Por exemplo, o grupo  $\{\{2, 3, 5, 6\}, \{2, 4\}\}$  pode ser particionado em relação a 0, criando o grupo  $\{\{2, 5\}, \{3, 6\}, \{2, 4\}\}$ .

Após ser particionado considerando-se a saída, devem-se calcular os próximos estados correspondentes ao grupo obtido em relação à entrada em questão (uma vez que, após a aplicação da entrada, a MEF estará no respectivo próximo estado). Por exemplo, após ser particionado e atualizado em relação a 0, o grupo  $\{\{2, 3, 5, 6\}, \{2, 4\}\}$  gerará o grupo  $\{\{3, 6\}, \{4, 2\}, \{5, 3\}\}$ . Ao se atualizar um grupo, pode ocorrer de o grupo ficar inconsistente. Um grupo é inconsistente caso, ao calcular os próximos estados, se perceba que dois estados que ainda não foram distinguidos levam ao mesmo estado (e, portanto, não poderão ser mais distinguidos!). É possível verificar que, ao se particionar o grupo  $\{\{2, 3, 5, 6\}, \{2, 4\}\}$  em relação a 1, se obtém o grupo  $\{\{2, 5, 6\}, \{3\}, \{4, 2\}\}$ , que é inconsistente, pois tanto o estado 2 quanto o estado 6 levam ao estado 5 quando a entrada 1 é aplicada.

Agora, de posse das operações de particionamento e atualização, pode-se construir uma árvore de distinção. Em uma árvore de distinção cada nó é formado por um grupo de incerteza. O nó raiz é formado pelo grupo no qual todos os estados ainda não foram distinguidos (ou seja,  $[S]$ ). Cada nó possui um filho para cada símbolo de entrada. O grupo do nó filho é calculado a partir do grupo de nó pai, particionando-o e, caso o grupo resultante não seja inconsistente, atualizando-o. A árvore é construída por nível a partir da raiz. Um nó é folha (i) se for inconsistente ou (ii) se for trivial. Uma DS é a seqüência de símbolos de entrada presentes no caminho que vai da raiz da árvore a um nó cujo grupo seja trivial.

A Figura 3.4 mostra a árvore de distinção para a MEF da Figura 3.3. Pode-se observar como os grupos são particionados e atualizados. Note que os conjuntos unitários são descartados da representação porque eles não têm influência alguma sobre o cálculo do DS. O mesmo é feito com os conjuntos vazios. Os grupos inconsistentes são marcados com retângulos. Os grupos triviais são marcados com elipses. Observe que a MEF em questão possui outras três DS próprias (no caso,  $\langle 0, 0, 1 \rangle$ ,  $\langle 0, 0, 0, 1 \rangle$ ,  $\langle 0, 0, 0, 0, 0 \rangle$ ,  $\langle 0, 0, 0, 0, 1 \rangle$ ).

### 3.3.3 Seqüências UIO

Como pôde ser observado na seção anterior, nem todas as MEFs possuem DSs. Entretanto, mesmo nessas situações, pode-se fazer a distinção entre um estado e os demais. Para isso, usa-se uma Seqüência Única de Entrada e Saída (UIO). Uma UIO é utilizada para verificar se a MEF está em um estado particular. Assim, para cada estado da MEF, pode-se ter uma UIO distinta. Isso é verdade apenas se forem consideradas tanto as entradas como as saídas. A

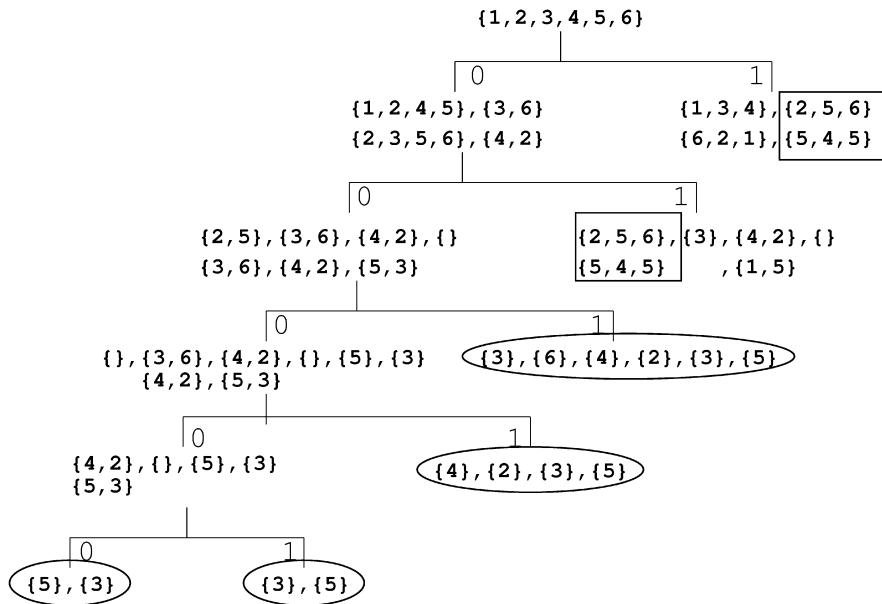


Figura 3.4 – Árvore de distinção.

seqüência de entradas da UIO de um estado pode ser igual à seqüência de entradas da UIO de outro estado. Assim, pode-se concluir que se uma MEF possui uma DS, ela também possui UIOs para todos os seus estados. Contudo, em geral, pode-se obter UIOs mais curtas.

Considere a MEF da Figura 3.3. A seqüência  $\langle 1/0, 1/0 \rangle$  é uma UIO para o estado 4 (anotaremos a saída logo após a entrada, separada por um '/'). Isso pode ser comprovado pelas saídas produzidas por todos os estados apresentada na Tabela 3.4. A UIO(4) deve ser usada como a seguir. Aplica-se a seqüência de entrada  $\langle 1, 1 \rangle$ . Se a seqüência de saída for  $\langle 0, 0 \rangle$ , então pode-se garantir que o estado original era o estado 4. No entanto, diferentemente da DS, nem sempre é possível dizer qual era o estado se a saída não for a esperada. Por exemplo, se a saída for  $\langle 0, 1 \rangle$ , não há como saber se o estado original era o estado 1 ou o estado 3. Pode-se notar também que a  $\langle 1, 1 \rangle$  também pode formar uma UIO para o estado 5 (que, nesse caso, seria a seqüência  $\langle 1/1, 1/0 \rangle$ ).

 Tabela 3.4 – Saída geradas para cada estado pelo  $UIO(1)$ 

	1	1
1	0	1
2	1	1
3	0	1
4	0	0
5	1	0
6	1	1

### 3.3.4 Conjunto de caracterização

Um conjunto de caracterização é um conjunto de seqüências de entrada que podem, em conjunto, identificar qual era o estado original. Por exemplo, o conjunto  $W = \{\langle 0 \rangle, \langle 1, 1 \rangle\}$  é um conjunto de caracterização para a MEF da Figura 3.3. Deve-se observar que, para se utilizar um conjunto de caracterização, cada seqüência deve ser aplicada ao estado original (que se deseja determinar). Ao se aplicar uma das seqüências, porém, a MEF é levada a um outro estado. Por isso, para se utilizar um conjunto de caracterização não unitário, deve-se possuir uma forma de restaurar a MEF ao estado original. Na Seção 3.4.4, que descreve o método W, é apresentada uma forma de se realizar essa restauração.

Para verificar que o conjunto  $W$  é um conjunto de caracterização, observe que cada linha da Tabela 3.5 é distinta, apesar de não o ser para uma seqüência em particular. Por exemplo, apesar de os estados 2 e 6 produzirem as mesmas saídas para as seqüências  $\langle 1, 1 \rangle$ , eles produzem saídas diferentes para a seqüência  $\langle 0 \rangle$ .

Tabela 3.5 – Saída gerada para cada estado pelas seqüências do conjunto de caracterização

	0	1	1
1	0	0	1
2	0	1	1
3	1	0	1
4	0	0	0
5	0	1	0
6	1	1	1

Para calcular o conjunto de caracterização, usamos o conceito de estados  $k$ -distinguíveis, para  $k \geq 0$ . Dois estados  $s_i$  e  $s_j$  são  $k$ -distinguíveis se existe uma seqüência de símbolos de entrada de tamanho  $k$  para a qual os estados  $s_i$  e  $s_j$  produzam saídas distintas. Em outras palavras, é possível decidir se o estado original era  $s_i$  ou  $s_j$  com uma seqüência com no máximo  $k$  símbolos. Define-se um conjunto  $D_k$  como o conjunto de pares não-ordenados de estados distintos que são  $k$ -distinguíveis.<sup>1</sup> O conjunto  $D_1$  pode ser calculado diretamente a partir da função de saída da MEF e é formado por todos os pares de entrada que são distinguidos por um único símbolo de entrada. Assim,

$$D_1 = \{(s_i, s_j) \in S \times S \mid \exists x \in X \wedge f_z(s_i, x) \neq f_z(s_j, x)\}$$

O conjunto  $D_2$  pode ser calculado a partir da função de próximo estado e do conjunto  $D_1$ . Se um par de estados  $(s_i, s_j)$  está em  $D_1$ , certamente estará em  $D_2$  também. Por outro lado, se  $(s_i, s_j)$  não está em  $D_1$ , mas  $\exists x \in X, (f_s(s_i, x), f_s(s_j, x)) \in D_1$ , pode-se verificar que  $(s_i, s_j) \in D_2$ . De uma forma geral, para  $k \geq 1$ , tem-se que

$$D_{k+1} = D_k \cup \{(s_i, s_j) \in S \times S \mid \exists x \in X, (f_s(s_i, x), f_s(s_j, x)) \in D_k\}$$

Algoriticamente, o cálculo pode parar quando  $D_{k+1} = D_k$ , ou seja, nenhum par é adicionado.

<sup>1</sup>O algoritmo apresentado é uma variação do algoritmo das tabelas de pares, apresentado por Gill [152].

Considere a MEF da Figura 3.3. Os conjuntos  $D_1$  e  $D_2$  seriam:

$$D_1 = \{(1, 2), (1, 3), (1, 5), (1, 6), (2, 3), (2, 4), (2, 6), \\ (3, 4), (3, 5), (3, 6), (4, 5), (4, 6), (5, 6)\}$$

e

$$D_2 = D_1 \cup \{(1, 4), (2, 5)\}$$

Pode-se observar que, para  $k \geq 2$ ,  $D_k = D_2$ .

De posse dos conjuntos  $D_k$ , podemos calcular uma seqüência capaz de distinguir entre qualquer par de estados. Suponha que se queira calcular uma seqüência para distinguir  $s_i$  de  $s_j$ . Procura-se o menor  $k$  tal que  $(s_i, s_j) \notin D'_{k-1}$  e  $(s_i, s_j) \in D'_k$ . (Para simplificar o algoritmo, pode-se assumir que  $D_0 = \{\}$ . A generalização é segura, pois nenhum par de estados é distingível com uma seqüência de tamanho 0.) Se não existir tal  $k$ , então os estados não podem ser distinguidos e são, portanto, equivalentes. Nesse caso, a MEF não é minimal.

Agora sabe-se que uma seqüência de tamanho  $k$  distingue  $s_i$  de  $s_j$ . Vamos, então, calculá-la, recursivamente. Se  $k = 1$ , a seqüência procurada é formada por um símbolo de entrada que produza saídas diferentes para  $s_i$  e  $s_j$ , ou seja, é  $\langle x \rangle$  para algum  $x$  tal que  $f_z(s_i, x) \neq f_z(s_j, x)$ . Para  $k > 1$ , encontre um  $x \in X$  tal que  $(f_s(s_i, x), f_s(s_j, x)) \in D_{k-1}$ . Pode existir mais de um  $x$  que satisfaça essa exigência. Pode-se escolher qualquer um. No entanto, como veremos, essa escolha pode influenciar na cardinalidade do conjunto de caracterização. Agora, calcula-se uma seqüência de tamanho  $k - 1$  que distinga  $f_s(s_i, x)$  e  $f_s(s_j, x)$ . A seqüência que distingue  $s_i$  e  $s_j$  é o símbolo  $x$  concatenado à seqüência de tamanho  $k - 1$  calculada.

Calculemos uma seqüência que distinga os estados 1 e 4. O menor  $k$  nesse caso é igual a 2, pois  $(1, 4) \notin D_1$  e  $(1, 4) \in D_2$ . O primeiro símbolo da seqüência é 1, uma vez que com esse símbolo a máquina vai para os estados 6 e 1, respectivamente. Agora, deve-se distinguir 6 e 1. A seqüência  $\langle 0 \rangle$  pode ser utilizada. Assim, a seqüência que distingue 1 e 4 é  $\langle 1, 0 \rangle$ .

Para calcular o conjunto de caracterização, calculamos seqüências que distingam cada par de estados. Nesse momento, a consistência na escolha entre dois ou mais possíveis símbolos de entrada pode ser importante para reduzir a cardinalidade do conjunto de caracterização. Por exemplo, se tanto  $\langle 0, 0 \rangle$  quanto  $\langle 1, 1 \rangle$  distinguem dois estados, mas apenas  $\langle 1, 1 \rangle$  distinguem outros dois, dependendo da forma como as seqüências são escolhidas, pode-se ter um conjunto de caracterização apenas com  $\langle 1, 1 \rangle$  ou com ambas as seqüências. Contudo, o problema de se calcular um conjunto de caracterização com o número mínimo de seqüências é NP-completo.

Para ilustrar o processo de geração de conjuntos de caracterização, vamos calculá-lo para a MEF da Figura 3.1. O  $D_1 = (1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)$  contém todos os pares de estados. A seqüência  $\langle * \rangle$  distingue os pares  $(1, 2), (1, 3), (1, 4), (2, 3)$  e  $(2, 4)$ . Por outro lado, a seqüência  $\langle / \rangle$  distingue o par  $(3, 4)$  (além de outros que também são distinguidos por  $\langle * \rangle$  e que não nos interessa nesse caso). Portanto, um conjunto de caracterização é  $\{\langle * \rangle, \langle / \rangle\}$ .

## 3.4 Geração de seqüências de teste

Uma seqüência de teste, nesse contexto, é uma seqüência de símbolos de entrada derivada da MEF que são submetidos à implementação correspondente para verificar a conformidade

da saída gerada pela implementação com a saída especificada na seqüência de teste. Relações formais de equivalência entre especificações e implementação foram trabalhadas em [398], mas não serão discutidas aqui. Os métodos mais clássicos para a geração de seqüências de teste a partir de MEFs são:

**Método TT (Transition Tour)** Para uma dada MEF, o transition tour é uma seqüência que parte do estado inicial atravessa todas as transições pelo menos uma vez e retorna ao estado inicial. Permite a detecção de erros de saída, mas não garante erros de transferência, ou seja, erros em que a transição leva a MEF a um estado diferente do qual ela deveria levar.

**Método UIO (Unique input/output)** Produz uma seqüência de identificação de estado. Não garante cobertura total dos erros, pois a seqüência de entrada leva a uma única saída na especificação correta, mas isso não é garantido para as implementações com erro.

**Método DS (Seqüência de distinção)** Uma seqüência de entrada é uma DS se a seqüência de saída produzida é diferente quando a seqüência de entrada é aplicada a cada estado diferente. A desvantagem desse método é que a DS nem sempre pode ser encontrada para uma dada MEF.

**Método W** É um conjunto de seqüências que permite distinguir todos os estados e garante detectar os erros estruturais, sempre que a MEF for completa, mínima e fortemente conexa.

Outros métodos como Método Wp, UIO-v, método-E, etc. são discutidos em (Lai, 2002). A seguir, são apresentados os métodos TT, UIO, DS e W.

### 3.4.1 Método TT

O método TT é relativamente simples, quando comparado com os outros três métodos. Ele gera um conjunto de seqüências que passam por todas as transições da MEF, ou seja, as seqüências geradas cobrem todas as transições.

Em alguns casos, o conjunto de seqüências é unitário, isto é, uma única seqüência passa por todas as transições. Contudo, para algumas MEFs, pode não ser possível calcular uma única seqüência. Por exemplo, a MEF pode possuir dois conjuntos de estados tais que, uma vez entrando em um deles, não se consegue chegar a um estado do outro conjunto, o que pode ser considerado um caso particular de MEF não completamente conexa. Nesse caso, pode-se utilizar um conjunto com mais seqüências.

O algoritmo para calcular as seqüências do método TT depende das propriedades que a MEF satisfaz. Em geral, algoritmos para grafos podem ser adaptados para esse propósito [152].

### 3.4.2 Método DS

O método DS [159] utiliza a seqüência de distinção DS para gerar as seqüências de teste. São utilizadas também as seqüências de sincronização. Assim, sua aplicabilidade fica condicionada à existência dessas seqüências para a MEF em questão.

Para aplicar o método DS, deve-se primeiramente selecionar uma seqüência DS. Como essa seqüência é utilizada diversas vezes ao longo do método, é importante que se selecione a menor seqüência. Chamaremos essa seqüência DS de  $X_d$ . Por exemplo, para a MEF da Figura 3.3, a seqüência  $X_d$  é  $\langle 0, 0, 1 \rangle$ .

Em seguida, gera-se um grafo, denominado *grafo*  $X_d$ , no qual cada nó representa um estado da MEF. Para cada estado  $s_i$ , existe no grafo um estado correspondente e existe uma aresta que liga  $s_i$  ao estado resultante da aplicação de  $X_d$  a  $s_i$ . O exemplo do grafo  $X_d$  para a MEF da Figura 3.3 e a seqüência  $X_d$  definida anteriormente são apresentados na Figura 3.5.

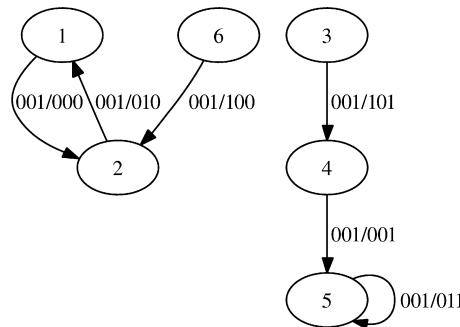


Figura 3.5 – Grafo  $X_d$ .

Gera-se a seqüência de verificação, a qual é formada por duas subseqüências, que são:

**seqüência- $\alpha$** , que verifica todos os estados da MEF; e

**seqüência- $\beta$** , que verifica todas as transições.

As seqüências- $\alpha$  são geradas percorrendo-se o grafo  $X_d$  sem que se repitam as arestas. Escolhe-se um estado como sendo o inicial. Percorre-se então o grafo marcando os estados como “reconhecido” conforme são atingidos. Quando um estado que já foi reconhecido é novamente visitado, precisa-se escolher outro estado inicial. No entanto, para garantir que o último estado foi atingido corretamente, aplica-se mais uma vez a seqüência  $X_d$ . Ao se escolher um estado inicial, deve-se primeiramente escolher o estado origem. Um estado origem é um estado que não é destino de nenhuma aresta no grafo  $X_d$ . Quando não houver mais estados origens, escolhe-se qualquer estado ainda não reconhecido. Agora, deve-se gerar a seqüência inicial do método, que é a seqüência de sincronização para o estado inicial da seqüência- $\alpha$ . Se não houver seqüências de sincronização para tal estado, seleciona-se uma seqüência inicial que deve ser encontrada por outros meios, tais como os apresentados no trabalho de Booth [40].

Vejamos como é a construção da seqüência- $\alpha$  a partir do grafo da Figura 3.5. Para começar, deve-se escolher entre os estados origens disponíveis 3 e 6. Escolhemos arbitrariamente o estado 3, marcando-o como “reconhecido”. Agora, aplica-se a seqüência  $X_d$ . O estado atingido é o estado 4, que é marcado como “reconhecido”. Aplica-se a seqüência  $X_d$ , atingindo-se o estado 5. Aplicando-se novamente a seqüência  $X_d$ , atinge-se novamente o estado 5. Assim, um novo estado deve ser escolhido. No entanto, antes de começar o processo para outro estado, aplica-se mais uma vez a seqüência  $X_d$ . Essa última aplicação é necessária

para verificar se o estado atingido foi realmente o estado 5. O novo estado inicial só pode ser o estado 6 (que é o único estado origem restante). Deve-se aplicar uma seqüência que leve a MEF do estado 5 ao estado 6. Nesse caso, a seqüência  $\langle 0 \rangle$  pode ser utilizada. Uma vez estando no estado 6, repete-se o procedimento anterior. A seqüência- $\alpha$  obtida é:

$$\langle 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1 \rangle$$

No caso da MEF da Figura 3.3, a seqüência inicial é a seqüência de sincronização para o estado 3, que é  $\langle 1, 1, 0, 1, 1, 1, 0, 0 \rangle$ .

Deve-se agora calcular a seqüência- $\beta$ . A seqüência- $\beta$  vai verificar se todas as transições da MEF estão corretas. Para isso, usa-se uma estratégia semelhante à utilizada para a seqüência- $\alpha$ . Todavia, em vez de se utilizar a seqüência  $X_d$ , utiliza-se uma seqüência da forma  $x_i X_d$ , para todo  $x_i \in X$ . Por exemplo, estando-se em um estado  $s_i$  que possui uma transição para o estado  $s_j$  com a entrada  $x_i$ , aplica-se primeiramente a entrada  $x_i$  e, em seguida, aplica-se a  $X_d$  para verificar se o estado atingido foi realmente o  $s_j$ . Em outras palavras, pode-se criar um grafo, chamado grafo- $\beta$ , tal que para cada estado exista um nó e para cada  $x_i \in X$  exista uma transição que leva um estado  $s_i$  a  $s_j = f_s(s_i, \langle x \rangle \cdot X_d)$ . Deve-se identificar uma seqüência que cubra todo o grafo. Por exemplo, partindo-se do estado 1 e aplicando-se a seqüência  $\langle 1 \rangle \cdot X_d = \langle 1, 0, 0, 1 \rangle$ , atinge-se o estado 2. A Figura 3.6 apresenta o grafo- $\beta$  para a MEF da Figura 3.3.

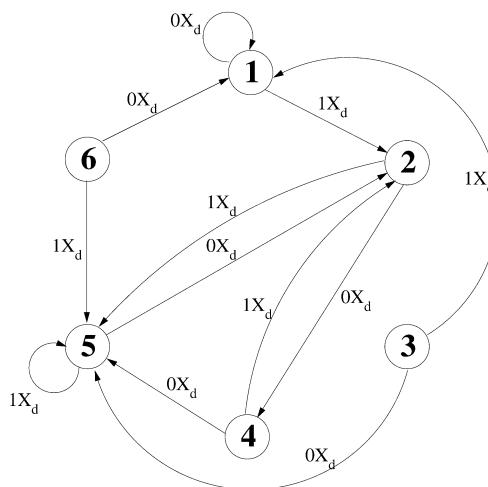


Figura 3.6 – Diagrama- $\beta$ .

Uma redução do grafo- $\beta$  e, consequentemente, das seqüências usadas para cobertura pode ser feita, considerando-se que a seqüência- $\alpha$  já foi aplicada. O que se pode observar é que, na seqüência- $\alpha$ , todos os estados já foram verificados. Assim, a última transição da aplicação de cada  $X_d$  já foi verificada. Por exemplo, aplicando a seqüência  $X_d (\langle 0, 0, 1 \rangle)$  no estado 2, a MEF vai para o estado 1, passando pelos estados 3 e 4. Se a transição que vai de 2 a 3 com a entrada 0 e a transição que vai de 3 a 4 com entrada 0 tiverem sido verificadas, não há necessidade de se verificar a transição que vai de 4 a 1 com entrada 1, visto que essa verificação já foi feita pela seqüência- $\alpha$  (caso contrário, se outro estado houvesse sido atingido, a seqüência- $\alpha$

teria identificado). Assim, podemos descartar do grafo- $\beta$  todas as arestas que correspondem ao último passo da aplicação do  $X_d$ . Observe na Tabela 3.6, passo a passo, a aplicação da seqüência  $X_d$  a cada um dos estados da MEF. As transições representadas pela aplicação da última entrada da seqüência  $X_d$  na penúltima coluna da tabela não precisam ser verificadas. Assim, as arestas partindo dos estados 2, 3, 4, 5 e 6 com a entrada 1 podem ser retiradas do grafo- $\beta$ . Do mesmo modo, também pode ser retirada a última transição da aplicação de uma seqüência incluída para ligar as componentes desconexas da geração da seqüência- $\alpha$ . Nesse caso, a seqüência  $\langle 0 \rangle$ , por exemplo, foi aplicada para levar a MEF do estado 5 ao estado 6. Assim, a aresta partindo do estado 5 com a entrada 0 pode ser retirada do grafo- $\beta$ . O grafo resultante é apresentado na Figura 3.7.

Tabela 3.6 – Saída resultante da aplicação da  $X_d$  a cada estado

	0	0	1
1	2	3	2
2	3	4	1
3	4	5	4
4	5	6	5
5	6	2	5
6	2	3	2

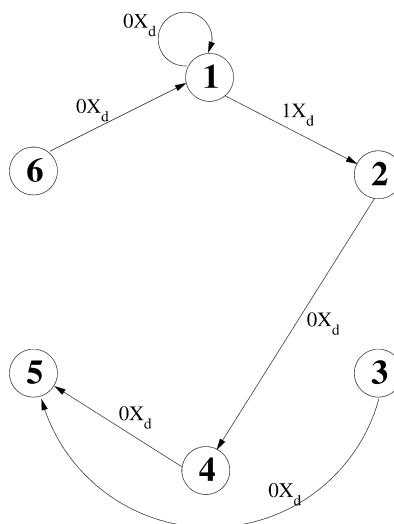


Figura 3.7 – Diagrama- $X_d$  reduzido.

Agora, pode-se calcular a seqüência- $\beta$  obtendo-se uma cobertura mínima das arestas do grafo- $\beta$ . Partindo-se do estado 1, a seqüência  $\langle 0X_d, 1X_d, 0X_d, 0X_d \rangle$  cobre quatro arestas, e pára no estado 5. Para cobrir as demais arestas, é necessário incluir uma seqüência que leva a outro estado que é origem de alguma aresta não coberta. Nesse caso, pode-se utilizar a seqüência  $\langle 0 \rangle$  que leva ao estado 6. Então, cobre-se essa aresta com  $\langle 0X_d \rangle$ , parando no estado

1. Utiliza-se a seqüência  $\langle 0, 0 \rangle$  para levar a MEF ao estado 3 (que ainda possui uma aresta não coberta). Com a seqüência  $\langle 0X_d \rangle$ ,obre-se a última aresta. A seqüência- $\beta$ , portanto, é

$$\langle 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1 \rangle$$

### 3.4.3 Método UIO

O método UIO utiliza as seqüências UIO para a geração de um conjunto de seqüências de entrada para testar a MEF. Para cada transição de um estado  $s_i$  para  $s_j = f_s(s_i, x)$ , com algum  $x$ , define-se uma seqüência que leva a máquina do estado inicial até  $s_i$ , aplica-se o símbolo de entrada  $x$ , em seguida, aplica-se a seqüência UIO do estado que deveria ser atingido. Ou seja, cada seqüência é da forma:

$$P(s_i) \cdot x \cdot UIO(s_j),$$

sendo que  $P(s_i)$  é uma seqüência que leva a MEF do estado inicial ao estado  $s_i$  e  $UIO(s_j)$  é uma seqüência para  $s_j$ .

Consideremos a MEF da Figura 3.3. Precisamos inicialmente calcular as seqüências  $P$  que levarão a MEF a cada um dos estados. A Tabela 3.7 apresenta tais seqüências. De posse das seqüências UIO para essa MEF (apresentadas na Tabela 3.4), podem-se calcular as seqüências do método UIO. Por exemplo, para a transição que liga o estado 4 ao estado 1 com a entrada 1, a seqüência correspondente é formada por  $P(4) \cdot 1 \cdot UIO(1) = \langle 0, 0, 1 \rangle \cdot \langle 1 \rangle \cdot \langle 1, 0 \rangle$ . O conjunto formado por todas as seqüências do método UIO é apresentado na Figura 3.8.

Tabela 3.7 – Seqüências que levam a MEF a cada um dos estados

P(1)	$\langle \rangle$
P(2)	$\langle 0 \rangle$
P(3)	$\langle 0, 0 \rangle$
P(4)	$\langle 0, 0, 0 \rangle$
P(5)	$\langle 0, 1 \rangle$
P(6)	$\langle 1 \rangle$

### 3.4.4 Método W

O método W utiliza o conjunto de caracterização para verificar se todos os estados e transições estão corretamente implementados. Cada seqüência é composta por duas subseqüências. A primeira subseqüência (denominada, seqüência- $T$ ) é formada a partir da árvore de teste, e serve para garantir que cada transição foi atingida. A segunda subseqüência (denominada, seqüência- $P$ ) é derivada a partir do conjunto de caracterização W, e é usada para verificar se o estado atingido após a transição está correto.

A árvore de teste é construída de forma que os nós correspondem aos estados e as arestas correspondem as transições. A raiz da árvore é o estado inicial e cada transição da MEF aparece exatamente uma vez na árvore. A Figura 3.9 apresenta a árvore de teste da máquina da Figura 3.1. As seqüências- $T$  são todos os caminhos parciais da árvore de teste.

$\langle 0, 0, 1 \rangle$	para o estado 1 entrada 0
$\langle 1, 0, 1 \rangle$	para o estado 1 entrada 1
$\langle 0, 0, 0, 1 \rangle$	para o estado 2 entrada 0
$\langle 0, 1, 1, 1 \rangle$	para o estado 2 entrada 1
$\langle 0, 0, 0, 1, 1 \rangle$	para o estado 3 entrada 0
$\langle 0, 0, 1, 0, 1 \rangle$	para o estado 3 entrada 1
$\langle 0, 0, 0, 0, 1, 1 \rangle$	para o estado 4 entrada 0
$\langle 0, 0, 0, 1, 1, 0 \rangle$	para o estado 4 entrada 1
$\langle 0, 1, 0, 0, 1 \rangle$	para o estado 5 entrada 0
$\langle 0, 1, 1, 1, 1 \rangle$	para o estado 5 entrada 1
$\langle 1, 0, 0, 1 \rangle$	para o estado 6 entrada 0
$\langle 1, 1, 1, 1 \rangle$	para o estado 6 entrada 1

Figura 3.8 – Seqüências geradas pelo método UIO.

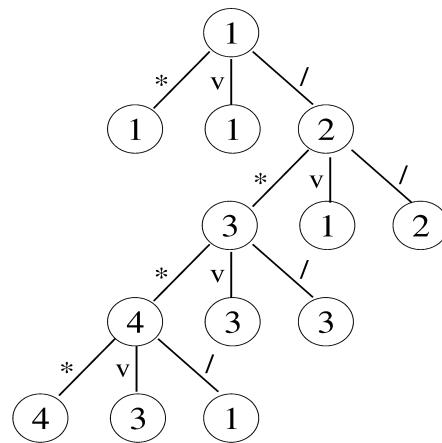


Figura 3.9 – Árvore de teste para a máquina da Figura 3.1.

Uma vez calculadas as seqüências  $T$ , calculam-se as seqüências  $P$  para verificar se as transições levaram aos estados corretos. Nesse ponto, é necessário fazer-se uma estimativa da quantidade máxima de estados que a implementação pode ter. Essa estimativa é essencial para o método, pois, garante-se que, se a implementação possuir no máximo essa quantidade de estados e passar por todas as seqüências de teste com o resultado correto, ela estará correta. A estimativa deve ser tão acurada quanto possível. Caso contrário, a quantidade de seqüências geradas é muito grande. Seja  $g$  a diferença entre a estimativa de estados adotada e quantidade de estados presentes na MEF, as seqüências  $P$  serão calculadas prefixando as seqüências do conjunto de caracterização com todas as possíveis seqüências de entrada com tamanho menor ou igual a  $g$ . Ou seja,

$$P = \left( \bigcup_{i=0}^g X^i \right) \cdot W$$

Por exemplo, se o número de estados da MEF é 5 e a estimativa é 7, tem-se que  $g = 2$  e  $P = (\langle \rangle \cup X \cup X \cdot X) \cdot W$ .

Para a máquina da Figura 3.1, o conjunto  $W$  é formado pelas seqüências  $\langle / \rangle$  e  $\langle * \rangle$ . Assim, o conjunto de seqüências de teste é formado pela concatenação de cada um dos caminhos parciais da árvore com cada uma das seqüências do conjunto de caracterização. A Figura 3.10 apresenta as seqüências do método W com a estimativa de estado igual ao número de estados da MEF, ou seja,  $g = 0$ .

$\langle / \rangle$	$\langle * \rangle$
$\langle */ \rangle$	$\langle ** \rangle$
$\langle v/ \rangle$	$\langle v* \rangle$
$\langle // \rangle$	$\langle /* \rangle$
$\langle / * / \rangle$	$\langle / ** \rangle$
$\langle / v/ \rangle$	$\langle / v* \rangle$
$\langle /// \rangle$	$\langle // * \rangle$
$\langle / * */ \rangle$	$\langle / * ** \rangle$
$\langle / * v/ \rangle$	$\langle / * v* \rangle$
$\langle / * / / \rangle$	$\langle / * / * \rangle$
$\langle / * * * / \rangle$	$\langle / * * * * \rangle$
$\langle / * *v/ \rangle$	$\langle / * *v* \rangle$
$\langle / * */ / \rangle$	$\langle / * */ * \rangle$

Figura 3.10 – Seqüências de teste geradas pelo método W.

### 3.5 Comparação entre os métodos

Alguns aspectos tornam-se bastante importantes para a seleção de um determinado método de geração de seqüências de teste, tais como as propriedades que o método exige que a MEF satisfaça, a aplicabilidade e o comprimento das seqüências geradas.

A Tabela 3.8 apresenta um resumo comparativo de todas as propriedades necessárias à aplicação de cada método, assim como as seqüências necessárias. Podemos observar que o método TT é o menos exigente.

Quanto à aplicabilidade, o método W é o que pode ser aplicado a qualquer MEF, uma vez que toda MEF minimal possui um conjunto de caracterização. Caso a MEF não seja minimal, ela pode ser reduzida para uma MEF minimal equivalente [152]. Os demais métodos exigem que determinadas seqüências existam, tais como a seqüência de distinção, o que nem sempre é verdadeiro, mesmo para MEFs minimais.

Para MEFs completamente especificadas, os métodos UIO, DS e W são capazes de detectar falhas nas saídas de todas as transições [296]. Dessa forma, eles são capazes de detectar erros de transferência (próximo estado incorreto), de operação (símbolo de entrada ou saída da transição incorretos) e de estados em excesso ou falta [79].

Tabela 3.8 – Comparação entre os métodos TT, UIO, DS e W

	TT	UIO	DS	W
Minimalidade		✓	✓	✓
Completamente especificada		✓	✓	✓
Fortemente conectada		✓	✓	
Máquina de Mealy		✓	✓	✓
Determinismo	✓	✓	✓	✓
Seqüências de sincronização			✓	
Seqüências de distinção			✓	
Seqüências únicas de entrada/saída		✓		
Conjunto de caracterização				✓

De acordo com Sidhu [357], o principal fato que contribui para o comprimento das seqüências de teste reside na escolha das seqüências de identificação. Assim, se o comprimento de uma seqüência de distinção é maior que o das seqüências UIO, então o método DS gerará subseqüências mais longas que o método UIO. Discussões semelhantes podem ser feitas para os demais métodos. Ainda de acordo com Sidhu [357], por meio de experimentação, verificou-se que o método UIO produz as seqüências de teste mais curtas que os métodos DS e W, ao passo que o método W produz as seqüências mais longas.

## 3.6 Considerações finais

A utilização de modelos para auxiliar a condução de testes traz diversas vantagens. A possibilidade de se eliminar a ambigüidade, ou ao menos reduzi-la, auxilia na elaboração mais rigorosa das expectativas que o sistema deve satisfazer. Pode-se, portanto, obter maior rigor.

A possibilidade de automação é outra vantagem (talvez, a mais importante). Diversos métodos podem ser utilizados para sistematicamente derivar casos de teste a partir do modelo. Nesse capítulo, apresentamos uma visão geral sobre o teste baseado em modelos, assim como alguns métodos de geração de casos de teste. Os métodos apresentados foram escolhidos entre os mais importantes. Contudo, essa é uma linha de pesquisa ainda muito intensa, na qual se tentam resolver vários problemas tanto de ordem teórica quanto de ordem prática que surgem da aplicação desses métodos em problemas reais. Por exemplo, em muitos casos, os modelos não apresentam as propriedades necessárias. Assim, os métodos e/ou os modelos devem ser adaptados ou estendidos.

# Capítulo 4

## Teste Estrutural

*Ellen Francine Barbosa (ICMC/USP)*  
*Marcos Lordello Chaim (EACH/USP)*  
*Auri Marcelo Rizzo Vincenzi (UNISANTOS)*  
*Márcio Eduardo Delamaro (UNIVEM)*  
*Mario Jino (DCA/FEEC/UNICAMP)*  
*José Carlos Maldonado (ICMC/USP)*

### 4.1 Introdução

Conforme discutido nos capítulos anteriores, técnicas e critérios de teste fornecem ao projetista de software uma abordagem sistemática e teoricamente fundamentada para a condução da atividade de teste. Além disso, constituem um mecanismo que pode auxiliar na garantia da qualidade dos testes e na maior probabilidade em revelar defeitos no software. Várias técnicas podem ser adotadas para se conduzir e avaliar a qualidade da atividade de teste, sendo diferenciadas de acordo com a origem das informações utilizadas para estabelecer os requisitos de teste [102].

A técnica estrutural (ou caixa branca) estabelece os requisitos de teste com base em uma dada implementação, requerendo a execução de partes ou de componentes elementares do programa [294, 329]. Os caminhos lógicos do software são testados, fornecendo-se casos de teste que põem à prova tanto conjuntos específicos de condições e/ou laços bem como pares de definições e usos de variáveis. Os critérios pertencentes à técnica estrutural são classificados com base na complexidade, no fluxo de controle e no fluxo de dados [261, 329, 429], e serão discutidos mais detalhadamente no decorrer deste capítulo.

A técnica estrutural apresenta uma série de limitações e desvantagens decorrentes das limitações inerentes à atividade de teste de programa como estratégia de validação [141, 191, 302, 337]. Tais aspectos introduzem sérios problemas na automatização do processo de validação de software [261]:

- não existe um procedimento de teste de propósito geral que possa ser usado para provar a correção de um programa;
- dados dois programas, é indecidível se eles computam a mesma função;

- é indecidível, em geral, se dois caminhos de um programa, ou de programas diferentes, computam a mesma função;
- é indecidível, em geral, se um dado caminho é executável, ou seja, se existe um conjunto de dados de entrada que leve à execução desse caminho.

É importante ressaltar, ainda, as limitações inerentes à técnica estrutural:

- caminhos ausentes: se o programa não implementa algumas funções, não existirá um caminho que corresponda àquela função; consequentemente, nenhum dado de teste será requerido para exercitá-la;
- correção coincidente: o programa pode apresentar, coincidentemente, um resultado correto para um dado em particular de entrada, satisfazendo um requisito de teste e não revelando a presença de um defeito; entretanto, se escolhido outro dado de entrada, o resultado obtido poderia ser incorreto.<sup>1</sup>

Independentemente dessas desvantagens, a técnica estrutural é vista como complementar às demais técnicas de teste existentes, uma vez que cobre classes distintas de defeitos [294, 329, 102]. De fato, os conceitos e experiência obtidos com o teste estrutural podem ser abstraídos e incorporados às demais técnicas. Ainda, as informações obtidas pela aplicação de critérios estruturais têm sido consideradas relevantes para as atividades de manutenção, depuração e avaliação da confiabilidade de software [172, 320, 329, 402, 403].

Este capítulo está organizado da maneira a seguir. Na Seção 4.2 é apresentada uma revisão histórica do surgimento e do desenvolvimento do teste estrutural. Em seguida, na Seção 4.3, são discutidas definições básicas e modelos subjacentes à técnica. Alguns dos principais critérios de teste estrutural em nível de unidade são abordados na Seção 4.4. Na Seção 4.5 são discutidas as características de algumas ferramentas de suporte ao teste estrutural, em particular da ferramenta POKE-TOOL. Por fim, na Seção 4.6 são apresentados os comentários finais sobre o tema.

## 4.2 Histórico

Os primeiros critérios estruturais para o teste de programas eram baseados essencialmente no fluxo de controle dos programas. O critério de McCabe [276], utilizando uma medida de complexidade de software baseada na representação de fluxo de controle de um programa – a complexidade ciclomática –, foi um dos primeiros critérios estruturais definidos. Já os critérios Todos-Nós, Todas-Arestas e Todos-Caminhos são os mais conhecidos da classe de critérios baseados em fluxo de controle e exigem, respectivamente, que cada comando, cada desvio e cada caminho do programa em teste seja executado pelo menos uma vez [294, 329].

De modo geral, o teste baseado unicamente nos critérios Todos-Nós e Todas-Arestas tem se mostrado pouco eficaz para revelar a presença de defeitos simples e triviais, mesmo para programas pequenos. Por outro lado, apesar de desejável, executar todos os caminhos de um programa é, na maioria das vezes, uma tarefa impraticável. De fato, para muitos programas

<sup>1</sup>Esta pode ser considerada uma limitação fundamental para qualquer estratégia de teste.

o número de caminhos é infinito ou extremamente grande devido à ocorrência de estruturas de iteração. Isso torna a aplicação prática do critério Todos-Caminhos bastante restrita.

Vários critérios foram propostos para a seleção de caminhos com o objetivo de introduzir critérios mais rigorosos que os critérios Todos-Nós e Todas-Arestas e menos restritivos e dispendiosos do que o critério Todos-Caminhos. Myers [294], por exemplo, discute a utilização dos critérios Cobertura de Condições (*condition coverage*), Cobertura de Decisões/Condições (*decision/condition coverage*) e Cobertura de Condições Múltiplas (*multiple-condition coverage*) como complemento aos critérios Todos-Nós (*statement coverage*) e Todas-Arestas (*decision coverage*). O critério Cobertura de Condições requer, basicamente, que cada condição em uma decisão assuma todos os valores de saída possíveis pelo menos uma vez. Já o critério Cobertura de Decisões/Condições requer que cada condição e cada decisão assumam todos os valores de saída possíveis pelo menos uma vez e, além disso, que cada ponto de entrada seja invocado pelo menos uma vez. Por fim, o critério Cobertura de Condições Múltiplas requer que todas as possíveis combinações de saída de uma condição em cada decisão, bem como todos os pontos de entrada, sejam exercitados ao menos uma vez.

Outras iniciativas limitam-se a sugerir restrições no número de vezes que um laço é executado [215]. Howden [186] propôs o critério Limite-Interior (*boundary-interior*) para classificar os caminhos em diferentes classes, dependendo da maneira como os laços são executados; ao menos um caminho que cubra cada classe é requerido. Woodward [444], motivado pela indicação de estudos experimentais de que a presença de um número significativo de defeitos pode ser revelada pela simples concatenação de arcos, propôs uma hierarquia de critérios que consiste essencialmente em concatenar seqüências de códigos terminadas por uma transferência de controle: LCSAJ – seqüência de código linear e desvio (*linear code sequence and jump*). É importante observar que tais abordagens ainda consideram, em essência, a estrutura de controle para derivar os requisitos de teste. Além disso, Howden [191] ressalta que, ainda que se limite o número de iterações de um laço a duas ou três vezes, a quantidade de caminhos a serem testados será freqüentemente muito grande.

Em meados da década de 1970 surgiram os critérios baseados na análise do fluxo de dados dos programas. A análise de fluxo de dados [176], por meio de análise estática do programa, tem sido utilizada para otimização de código por compiladores e para detecção de anomalias em programas. Em geral, cada ocorrência de uma variável no programa é classificada como uma definição ou como um uso. Assim, tais critérios requerem que sejam testadas as interações que envolvem definições de variáveis e subsequentes referências a essas definições, ou seja, exigem a execução de caminhos do ponto em que uma variável foi definida, até o ponto em que ela foi utilizada [178, 231, 261, 301, 337, 400].

A introdução dos critérios baseados em fluxo de dados teve como principal objetivo o estabelecimento de uma hierarquia entre os critérios Todas-Arestas e Todos-Caminhos, visando a tornar o teste estrutural mais rigoroso. Segundo Howden [191], critérios baseados em fluxo de dados são mais seletivos, além de promover a concatenação de arcos, e conduzem à seleção de caminhos com uma maior relação com os aspectos funcionais do programa.

Herman [178] pode ser considerado um dos precursores do uso de informações de fluxo de dados para o estabelecimento de critérios de teste. Essencialmente, o “critério de Herman” requer que toda referência (uso) a uma variável seja exercitada, pelo menos uma vez, a partir de pontos do programa em que essa variável é definida, ou seja, Herman estabelece associações entre definições e subsequentes usos (sem distinção do tipo de uso) de variáveis, e requer que essas associações sejam exercitadas pelos casos de teste.

Introduzindo uma distinção entre tipos de referências a variáveis – uso computacional (c-uso) e uso predicativo (p-uso) –, Rapps e Weyuker [336, 337] propuseram na década de 1980 uma família de critérios de fluxo de dados. Os critérios centrais são: Todas-Definições, Todos-Usos e Todos-Du-Caminhos. Os demais critérios definidos pelas autoras – Todos-p-Usos, Todos-p-Usos/Alguns-c-Usos e Todos-c-Usos/Alguns-p-Usos – são variações do critério Todos-Usos.

Além dos critérios de Rapps e Weyuker, outros critérios de teste que utilizam informações de fluxo de dados foram propostos. Ntafos [301] introduziu uma família de critérios denominada K-tuplas requeridas, a qual requer que todas as seqüências de (K-1), ou menos, interações definição-uso sejam testadas. Variando-se  $K$ , obtém-se a família de critérios K-tuplas requeridas. Laski e Korel [231] propuseram os critérios Ambiente de Dados, Contexto Elementar de Dados e Contexto Ordenado de Dados. A introdução desses critérios é baseada na intuição de que valores atribuídos a uma variável em um ponto particular  $i$  do programa podem depender de valores de outras variáveis, em que cada uma tenha várias definições distintas que atinjam o comando  $s_i$  por caminhos livres de definição.<sup>2</sup> Ural e Yang [400] introduziram o critério Todos-Oi-Caminhos-Simples, o qual procura explorar efeitos entre entradas e saídas do programa. A idéia básica é estabelecer concatenações de associações entre definições e usos de variáveis que levem à determinação de caminhos completos. Para isso são estabelecidas seqüências (concatenações) de associações, definindo quando o uso de uma variável  $x$  é afetado pela definição de uma variável  $y$ .

Os critérios de Rapps e Weyuker, assim como os demais critérios baseados em fluxo de dados, exigem a ocorrência explícita de um uso de variável ao estabelecer um determinado elemento (caminho, associação, etc.). Esse aspecto determina as principais limitações dessa classe de critérios. Nessa perspectiva, no final dos anos 80 e início dos anos 90, Maldonado [261] introduziu a família de critérios Potenciais-Usos e a correspondente família de critérios executáveis. Os critérios básicos que fazem parte dessa família são: Todos-Potenciais-Usos, Todos-Potenciais-Usos/Du e Todos-Potenciais-Du-Caminhos. De modo geral, os critérios Potenciais-Usos baseiam-se nas associações entre uma definição de variável e seus possíveis subsequentes usos para a derivação de casos de teste.

O conceito de “potencial-uso” traz pequenas, mas fundamentais, modificações nos conceitos apresentados anteriormente. Os elementos a serem requeridos são caracterizados independentemente da ocorrência explícita de uma referência a uma determinada definição. Se um uso dessa definição pode existir (um potencial-uso) a “potencial-associação” entre a definição e o potencial-uso é caracterizada, e eventualmente requerida. De fato, a introdução do conceito potencial-uso procura explorar todos os possíveis efeitos a partir de uma mudança de estado do programa em teste, decorrente de definição de variáveis em um determinado nó  $i$ . Segundo Maldonado [261], tais critérios exploram idéias em concordância com a filosofia discutida por Myers [294]: um defeito está claramente presente se um programa não faz o que se supõe que ele faça, mas defeitos estão também presentes se um programa faz o que se supõe que não faça.

É importante observar que os requisitos de teste exigidos pelos critérios estruturais citados limitam-se ao escopo da unidade. De fato, tais critérios foram inicialmente propostos para o teste de unidade de programas procedimentais. No entanto, esforços na tentativa de estender o uso de critérios estruturais para o teste de integração também podem ser identificados [160, 171, 200, 244, 411]. Além disso, na tentativa de adequar sua utilização em

<sup>2</sup>A formalização de caminhos livres de definição é feita na Seção 4.3 deste capítulo.

diferentes contextos, extensões dos critérios estruturais também têm sido propostas para o teste de programas OO e de componentes, e para o teste de aspectos. Tais extensões serão discutidas em detalhes nos Capítulos 6 e 7.

## 4.3 Definições e conceitos básicos

A seguir são apresentados a terminologia e os conceitos básicos pertinentes ao teste estrutural. São discutidos os modelos subjacentes à técnica e detalhados os elementos estruturais que determinam os requisitos de teste e a análise de cobertura associados.

O teste estrutural baseia-se no conhecimento da estrutura interna do programa, sendo os aspectos de implementação fundamentais para a geração/seleção dos casos de teste associados. Em geral, a maioria dos critérios da técnica estrutural utiliza uma representação de programa conhecida como “Grafo de Fluxo de Controle” (GFC) ou “Grafo de Programa”. Um programa  $P$  pode ser decomposto em um conjunto de blocos disjuntos de comandos. A execução do primeiro comando de um bloco acarreta a execução de todos os outros comandos desse bloco, na ordem dada. Todos os comandos de um bloco, possivelmente com exceção do primeiro, têm um único predecessor e exatamente um único sucessor, exceto possivelmente o último comando.

Usualmente, a representação de um programa  $P$  como um GFC ( $G = (N, E, s)$ ) consiste em estabelecer uma correspondência entre vértices (nós) e blocos e em indicar possíveis fluxos de controle entre blocos por meio das arestas (arcos). Assume-se que um GFC é um grafo orientado, com um único nó de entrada  $s \in N$  e um único nó de saída  $o \in N$ , no qual cada vértice representa um bloco indivisível de comandos e cada aresta representa um possível desvio de um bloco para outro. Quando essas condições não forem satisfeitas, esse fato será indicado explicitamente. Cada bloco de comandos tem as seguintes características:

- uma vez que o primeiro comando do bloco é executado, todos os demais são executados seqüencialmente;
- não existe desvio de execução para nenhum comando dentro do bloco.

A partir do GFC podem ser escolhidos os elementos que devem ser executados, caracterizando assim o teste estrutural.

Seja um GFC  $G = (N, E, s)$  em que  $N$  representa o conjunto de nós,  $E$  o conjunto de arcos, e  $s$  o nó de entrada. Um “caminho” é uma seqüência finita de nós  $(n_1, n_2, \dots, n_k)$ ,  $k \geq 2$ , tal que existe um arco de  $n_i$  para  $n_i + 1$  para  $i = 1, 2, \dots, k - 1$ . Um caminho é um “caminho simples” se todos os nós que compõem esse caminho, exceto possivelmente o primeiro e o último, são distintos; se todos os nós são distintos, diz-se que esse caminho é um “caminho livre de laço”. Um “caminho completo” é aquele em que o primeiro nó é o nó de entrada e o último nó é um nó de saída do grafo  $G$ .

Seja  $IN(x)$  e  $OUT(x)$  o número de arcos que entram e que saem do nó  $x$ , respectivamente. Assumimos  $IN(s) = 0$ , tal que  $s$  é o nó de entrada, e  $OUT(o) = 0$ , tal que  $o$  é o nó de saída.

As ocorrências de uma variável em um programa podem ser uma “definição”, uma “indefinição” ou um “uso”. Usualmente, os tipos de ocorrências de variáveis são definidos por um modelo de fluxo de dados.

Conforme o modelo de fluxo de dados definido por Maldonado [261], uma definição de variável ocorre quando um valor é armazenado em uma posição de memória. Em geral, em um programa, uma ocorrência de variável é uma definição se ela está: (i) no lado esquerdo de um comando de atribuição; (ii) em um comando de entrada; ou (iii) em chamadas de procedimentos como parâmetro de saída. A passagem de valores entre procedimentos por meio de parâmetros pode ser por valor, referência ou por nome [150]. Se a variável for passada por referência ou por nome, considera-se que seja um parâmetro de saída. As definições decorrentes de possíveis definições em chamadas de procedimentos são diferenciadas das demais e são ditas definidas por referência. Por outro lado, diz-se que uma variável está indefinida quando não se tem acesso ao seu valor ou sua localização deixa de estar definida na memória.

A ocorrência de uma variável é um uso quando a referência a essa variável não a estiver definindo. Dois tipos de usos são caracterizados: “c-uso” e “p-uso”. O primeiro tipo afeta diretamente uma computação realizada ou permite que o resultado de uma definição anterior possa ser observado; o segundo tipo afeta diretamente o fluxo de controle do programa. Observa-se que enquanto c-usos estão associados aos nós do GFC, p-usos estão associados a seus arcos.

Considere uma variável  $x$  definida em um nó  $i$ , com uso em um nó  $j$  ou em um arco que chega em  $j$ . Um caminho  $(i, n_1, \dots, n_m, j)$ ,  $m \geq 0$  que não contenha definição de  $x$  nos nós  $n_1, \dots, n_m$ , é chamado de “caminho livre de definição” c.r.a  $x$  do nó  $i$  ao nó  $j$  e do nó  $i$  ao arco  $(n_m, j)$ .

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

A título de ilustração, considere o programa `identifier` (Programa 4.1). O programa é responsável por determinar se um identificador é válido ou não – um identificador válido deve começar com uma letra e conter apenas letras ou dígitos; além disso, deve ter no mínimo 1 e no máximo 6 caracteres de comprimento. É importante observar que o programa `identifier` contém um defeito.

---

Programa 4.1

---

```

1      main ()
2  /* 1 */
3  /* 1 */     char achar;
4  /* 1 */     int length, valid_id;
5  /* 1 */     length = 0;
6  /* 1 */     valid_id = 1;
7  /* 1 */     printf ("Identificador: ");
8  /* 1 */     achar = fgetc (stdin);
9  /* 1 */     valid_id = valid_s(achar);
10 /* 1 */     if(valid_id)
11 /* 2 */
12 /* 2 */     length = 1;
13 /* 2 */
14 /* 3 */     achar = fgetc (stdin);

```

```

15  /* 4 */      while(achar != '\n')
16  /* 5 */      {
17  /* 5 */          if(!valid_f(achar))
18  /* 6 */          {
19  /* 6 */              valid_id = 0;
20  /* 6 */          }
21  /* 7 */          length++;
22  /* 7 */          achar = fgetc (stdin);
23  /* 7 */      }
24  /* 8 */      if(valid_id && (length >= 1) && (length < 6))
25  /* 9 */      {
26  /* 9 */          printf ("Valido\n");
27  /* 9 */      }
28  /* 10 */     else
29  /* 10 */     {
30  /* 10 */         printf ("Invalido\n");
31  /* 10 */     }
32  /* 11 */ }
33
34         int valid_s(char ch)
35  /* 1 */ {
36  /* 1 */     if(((ch >= 'A') && (ch <= 'Z')) || ((ch >= 'a') && (ch <= 'z')))
37  /* 2 */     {
38  /* 2 */         return (1);
39  /* 2 */     }
40  /* 3 */     else
41  /* 3 */     {
42  /* 3 */         return (0);
43  /* 3 */     }
44  /* 4 */ }
45
46         int valid_f(char ch)
47  /* 1 */ {
48  /* 1 */     if(((ch >= 'A') && (ch <= 'Z')) || ((ch >= 'a') && (ch <= 'z')) ||
49  /* 2 */         ((ch >= '0') && (ch <= '9')))
50  /* 2 */     {
51  /* 2 */         return (1);
52  /* 2 */     }
53  /* 3 */     else
54  /* 3 */     {
55  /* 3 */         return (0);
56  /* 3 */     }
57  /* 4 */ }
58

```

---

O primeiro bloco de comandos (nó) é caracterizado da linha 2 à linha 10. O segundo bloco é formado pelas linhas de 11 a 13. O terceiro bloco refere-se à linha 14, e assim por diante. Ao todo, 11 nós constituem o GFC referente à função `main` do programa `identifier`, 4 nós constituem o grafo referente à função `valid_s` e 4 nós constituem o grafo da função `valid_f`. Na Figura 4.1 é ilustrado o grafo obtido referente à função `main`, gerado pela ferramenta *ViewGraph*<sup>3</sup> [412].

O comando `if(valid_id)` (linha 10) ilustra um desvio de execução entre os nós do programa. Caso sejam exercitados os comandos internos ao `if`, tem-se um desvio de execução do nó 1 para o nó 2, representado no grafo pelo arco (1,2). Do contrário, se os comandos

<sup>3</sup>A *ViewGraph* é uma ferramenta para visualização de Grafos de Fluxo de Controle e informações de teste. Detalhes a respeito da ferramenta podem ser encontrados em Vilela et al. [412].

internos ao `if` não forem executados, tem-se um desvio do nó 1 para o nó 3, representado pelo arco (1,3). De maneira similar, obtém-se os arcos (2,3),(3,4),(4,5), e assim por diante.

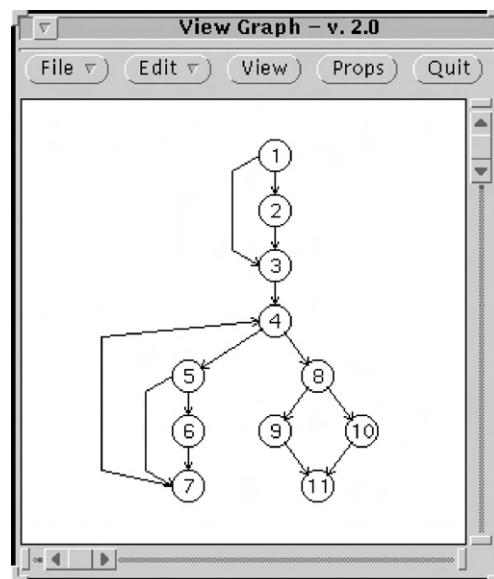


Figura 4.1 – Grafo de Fluxo de Controle do programa `identifier` gerado pela *ViewGraph*.

O caminho (2,3,4,5,6,7) é um caminho simples e livre de laços. O caminho (1,2,3,4,5,7,4,8,9,11) é um caminho completo. O caminho (1,3,4,8,9) é dito um “caminho não executável”, ou seja, não existe um dado de entrada que leve à execução desse caminho. Para executar o caminho (1,3,4,8,9) a variável `valid_id` tem de ser avaliada como 0 no comando `if(valid_id)` (nó 1), fazendo com que haja um desvio de execução do nó 1 para o nó 3. Observe que a variável `length` permanece com seu valor inicial, ou seja, 0. Além disso, para que haja um desvio de execução do nó 4 para o nó 8, os comandos internos à estrutura de repetição `while (achar != '\n')` (nó 4) não podem ser executados. Nesse caso, a variável `achar` deve receber um `'\n'` no nó 3. Como os valores de `valid_id` e `length` são iguais a 0, o comando condicional no nó 8 é avaliado como falso e, consequentemente, não ocorre desvio do nó 8 para o nó 9. Nesse caso, os comandos do nó 9 não são executados. Caracteriza-se, assim, que o caminho (1,3,4,8,9) é não executável. De fato, qualquer caminho completo que inclua tal caminho também é considerado não executável. A definição formal a respeito de caminhos não executáveis é discutida na Seção 4.4.3 deste capítulo.

O comando `length = 0` (linha 5) consiste em uma definição de variável, enquanto os comandos `achar != '\n'` (linha 15) e `length++` (linha 21) correspondem, respectivamente, a um uso predutivo e a um uso computacional de variável (neste caso, seguido de uma redefinição, visto que `length++` é equivalente a `length = length + 1`).

Os critérios de teste estrutural baseiam-se em diferentes tipos de conceitos e elementos de programas para determinar os requisitos de teste. Na Tabela 4.1 ilustram-se alguns desses elementos e critérios associados, os quais serão apresentados e discutidos nas próximas seções.

Tabela 4.1 – Elementos e critérios associados em relação ao programa `identifier`

Elemento	Exemplo ( <code>identifier</code> )	Critério
Nó	6	Todos-Nós
Arco	(5,6)	Todas-Arestas
Caminho	(1,2,3,4,8,9,11)	Todos-Caminhos
Definição de variáveis	<code>length=0</code>	Todas-Defs
Uso predicativo de variável	<code>achar != '\n'</code>	Todos-p-Usos
Uso computacional de variável	<code>length++</code>	Todos-c-Usos

## 4.4 Critérios de teste estrutural

Os critérios estruturais são, em geral, classificados em: (i) critérios baseados na complexidade, (ii) critérios baseados em fluxo de controle e (iii) critérios baseados em fluxo de dados. A seguir são apresentados e discutidos os principais critérios de teste associados a cada uma dessas classes, em nível de unidade. Atenção especial é dada aos critérios baseados em fluxo de dados, particularmente aos critérios de Rapps e Weyuker [336, 337] e aos critérios Potenciais-Usos [261].

### 4.4.1 Critérios baseados na complexidade

Os critérios baseados na complexidade utilizam informações sobre a complexidade do programa para derivar os requisitos de teste. Um critério bastante conhecido dessa classe é o critério de McCabe (ou teste de caminho básico) [276], que utiliza a complexidade ciclomática do GFC para derivar os requisitos de teste.

A complexidade ciclomática é uma métrica de software que proporciona uma medida quantitativa da complexidade lógica de um programa. Utilizado no contexto do teste de caminho básico, o valor da complexidade ciclomática estabelece o número de caminhos linearmente independentes do conjunto básico de um programa, oferecendo um limite máximo para o número de casos de teste que devem ser derivados a fim de garantir que todas as instruções sejam executadas pelo menos uma vez [329].

Um caminho linearmente independente é qualquer caminho do programa que introduza pelo menos um novo conjunto de instruções de processamento ou uma nova condição. Quando estabelecido em termos de um GFC, um caminho linearmente independente deve incluir pelo menos um arco que não tenha sido atravessado antes que o caminho seja definido. Assim, cada novo caminho introduz um arco.

Caminhos linearmente independentes estabelecem um conjunto básico para o GFC [329]. Ou seja, se casos de teste puderem ser projetados a fim de forçar a execução desses caminhos (um conjunto básico), cada instrução do programa terá a garantia de ser executada pelo menos uma vez e cada condição terá sido executada com verdadeiro e falso. É importante observar que o conjunto básico não é único; de fato, diferentes conjuntos básicos podem ser derivados para um determinado GFC.

Para saber quantos caminhos devem se procurados, é necessário calcular a complexidade ciclomática  $V(G)$ , que pode ser computada de três maneiras [329]:

1. o número de regiões em um GFC. Uma região pode ser informalmente descrita como uma área incluída no plano do grafo. O número de regiões é computado contando-se todas as áreas delimitadas e a área não delimitada fora do grafo; ou
2.  $V(G) = E - N + 2$ , tal que  $E$  é o número de arcos e  $N$  é o número de nós do GFC  $G$ ; ou
3.  $V(G) = P + 1$ , tal que  $P$  é o número de nós predicativos contido no GFC  $G$ .

O valor da complexidade ciclomática  $V(G)$  oferece um limite máximo para o número de caminhos linearmente independentes que constitui o conjunto básico e, consequentemente, um limite máximo do número de casos de teste que deve ser projetado e executado para garantir a cobertura de todas as instruções de programa [329].

Segundo Pressman [329], visto que o número de regiões aumenta com o número de caminhos de decisão e laços, a métrica de McCabe oferece uma medida quantitativa da dificuldade de conduzir os testes e uma indicação da confiabilidade final.

#### 4.4.2 Critérios baseados em fluxo de controle

Os critérios baseados em fluxo de controle utilizam apenas características de controle da execução do programa, como comandos ou desvios, para determinar quais estruturas são necessárias. Os critérios mais conhecidos dessa classe são [294, 329]:

- Todos-Nós: exige que a execução do programa passe, ao menos uma vez, em cada vértice do GFC; ou seja, que cada comando do programa seja executado pelo menos uma vez;
- Todas-Arestas (ou Todos-Arcos): requer que cada aresta do grafo, isto é, cada desvio de fluxo de controle do programa, seja exercitada pelo menos uma vez;
- Todos-Caminhos: requer que todos os caminhos possíveis do programa sejam executados.

É importante ressaltar que a cobertura do critério Todos-Nós é o mínimo esperado de uma boa atividade de teste, pois, do contrário, o programa testado é entregue sem a certeza de que todos os comandos presentes foram executados ao menos uma vez.

Além disso, conforme observado anteriormente, apesar de desejável, executar todos os caminhos de um programa é, na maioria dos casos, uma tarefa impraticável. Isso se deve ao fato de que, na presença de laços, o número de caminhos de um programa pode ser muito grande ou até mesmo infinito. Tal aspecto foi uma das motivações para a introdução dos critérios baseados em fluxo de dados, discutidos a seguir.

#### 4.4.3 Critérios baseados em fluxo de dados

Os critérios baseados em fluxo de dados utilizam a análise de fluxo de dados [176] como fonte de informação para derivar os requisitos de teste. Uma característica comum aos critérios dessa categoria é que eles requerem o teste das interações que envolvam definições de variáveis e subsequentes referências a essas definições [178, 231, 301, 337, 400]. Portanto, para a derivação de casos de teste, tais critérios baseiam-se nas associações entre a definição de uma variável e seus possíveis usos subsequentes.

Uma motivação para a introdução dos critérios baseados em fluxo de dados foi a indicação de que, mesmo para programas pequenos, o teste baseado unicamente no fluxo de controle não era eficaz para revelar a presença mesmo de defeitos simples e triviais. Nesse sentido, a introdução dessa classe de critérios procurou estabelecer uma hierarquia entre os critérios Todas-Arestas e Todos-Caminhos, visando a tornar o teste estrutural mais rigoroso. De acordo com Ural [400], critérios baseados em fluxo de dados são mais adequados para certas classes de defeitos, tais como defeitos computacionais, uma vez que dependências de dados são identificadas e, portanto, segmentos funcionais são requeridos como requisitos de teste.

A seguir são apresentadas duas famílias de critérios baseados em fluxo de dados: a família de critérios proposta por Rapps e Weyuker [336, 337] e a família de critérios Potenciais-Usos, proposta por Maldonado [261].

#### Critérios de Rapps e Weyuker

Entre os critérios de fluxo de dados, destacam-se os critérios de Rapps e Weyuker [336, 337], introduzidos nos anos 80. Para derivar os requisitos de teste exigidos por tais critérios, Rapps e Weyuker propuseram alguns conceitos e definições. Um deles foi o “Grafo Def-Usos” (*def-use graph*), o qual consiste em uma extensão do GFC. Nesse grafo são adicionadas informações a respeito do fluxo de dados do programa, caracterizando associações entre pontos do programa nos quais é atribuído um valor a uma variável (definição da variável) e pontos nos quais esse valor é utilizado (referência ou uso da variável). Os requisitos de teste são determinados com base em tais associações.

O Grafo Def-Usos é obtido a partir do GFC associando-se a cada nó  $i$  os conjuntos  $c\text{-uso}(i) = \{\text{variáveis com } c\text{-uso global no bloco } i\}$  e  $\text{def}(i) = \{\text{variáveis com definições globais no bloco } i\}$ , e a cada arco  $(i, j)$  o conjunto  $p\text{-uso}(i, j) = \{\text{variáveis com } p\text{-usos no arco } (i, j)\}$ . Dois conjuntos foram definidos:  $\text{dcu}(x, i) = \{\text{nós } j, \text{ tal que } x \in c\text{-uso}(j) \text{ e existe um caminho livre de definição c.r.a } x \text{ do nó } i \text{ para o nó } j\}$  e  $\text{dpu}(x, i) = \{\text{arcos } (j, k), \text{ tal que } x \in p\text{-uso}(j, k) \text{ e existe um caminho livre de definição c.r.a } x \text{ do nó } i \text{ para o arco } (j, k)\}$ .

Adicionalmente, foram definidos os conceitos de “du-caminho”, “associação definição-c-uso”, “associação definição-p-uso” e “associação”. Um caminho  $(n_1, n_2, \dots, n_j, n_k)$  é um “du-caminho” c.r.a variável  $x$  se  $n_1$  tiver uma definição global de  $x$  e: (1)  $n_k$  tem um c-uso de  $x$  e  $(n_1, n_2, \dots, n_j, n_k)$  é um caminho simples livre de definição c.r.a  $x$ ; ou (2)  $(n_j, n_k)$  tem um p-uso de  $x$  e  $(n_1, n_2, \dots, n_j, n_k)$  é um caminho livre de definição c.r.a  $x$  e  $n_1, n_2, \dots, n_j$  é um caminho livre de laço.

Uma “associação definição-c-uso” é uma tripla  $\langle i, j, x \rangle$  em que  $i$  é um nó que contém uma definição global de  $x$  e  $j \in \text{dcu}(x, i)$ . Uma “associação definição-p-uso” é uma tripla  $\langle i, (j, k), x \rangle$  em que  $i$  é um nó que contém uma definição global de  $x$  e  $(j, k) \in \text{dpu}(x, i)$ . Uma “associação” é uma associação definição-c-uso, uma associação definição-p-uso ou um du-caminho.

Com base nos conceitos mencionados e nas definições apresentadas na Seção 4.3, Rapps e Weyuker propuseram uma família de critérios de fluxo de dados [336, 337]. Os principais critérios dessa família são:

- Todas-Definições: requer que cada definição de variável seja exercitada pelo menos uma vez, não importa se por um c-uso ou por um p-uso;
- Todos-Usos: requer que todas as associações entre uma definição de variável e seus subseqüentes usos (c-usos e p-usos) sejam exercitadas pelos casos de teste, por pelo menos um caminho livre de definição, ou seja, um caminho em que a variável não é redefinida. Os critérios Todos-p-Usos, Todos-p-Usos/Alguns-c-Usos e Todos-c-Usos/Alguns-p-Usos representam variações do critério Todos-Usos;
- Todos-Du-Caminhos: requer que toda associação entre uma definição de variável e subseqüentes p-usos ou c-usos dessa variável seja exercitada por todos os caminhos livres de definição e livres de laço que cubram essa associação.

É importante ressaltar que, para requerer um determinado elemento (caminho, associação, etc.), a maior parte dos critérios baseados em fluxo de dados exige a ocorrência explícita de um uso de variável e não garante, necessariamente, a inclusão do critério Todas-Arestas na presença de caminhos não executáveis, presentes na maioria dos programas. Tais aspectos motivaram o estudo e a introdução de novos conceitos para a definição de critérios que objetivam, sobretudo, eliminar essas deficiências. A família de critérios Potenciais-Usos, discutida a seguir, foi definida nessa perspectiva.

## Critérios Potenciais-Usos

Conforme será visto no Capítulo 10, a relação de inclusão e a complexidade dos critérios são propriedades importantes associadas aos critérios de teste, sendo utilizadas para avaliá-los, do ponto de vista teórico.

Em linhas gerais, a “complexidade de um critério  $C$ ” é definida como o número máximo de casos de teste requerido pelo critério no pior caso. Ou seja, dado um programa qualquer  $P$ , se existir um conjunto de casos de teste  $T$  que seja  $C$ -adequado para  $P$ , então a cardinalidade de  $T$  é menor ou igual à complexidade do critério  $C$ .

Já a relação de inclusão estabelece uma ordem parcial entre os critérios, caracterizando uma hierarquia entre eles. Dados dois critérios  $C_1$  e  $C_2$ , diz-se que  $C_1$  inclui  $C_2$  se, para qualquer programa  $P$ , todo conjunto de teste  $C_1$ -adequado é também  $C_2$ -adequado. O critério  $C_1$  inclui estritamente o critério  $C_2$ , denotado por  $C_1 \Rightarrow C_2$ , se  $C_1$  inclui  $C_2$  e  $C_2$  não inclui  $C_1$ . Quando nem  $C_1$  inclui  $C_2$  nem  $C_2$  inclui  $C_1$ , diz-se que os critérios  $C_1$  e  $C_2$  são incomparáveis [337].

A complexidade e a relação de inclusão refletem, em geral, as propriedades básicas que devem ser consideradas no processo de definição de um critério de teste  $C$ , a saber [261]:

1. incluir o critério Todas-Arestas, ou seja, um conjunto de casos de teste que exerce os elementos requeridos pelo critério  $C$  deve exercitar todos os desvios de execução do programa;
2. requerer, do ponto de vista de fluxo de dados, ao menos um uso de todo resultado computacional; isto equivale ao critério  $C$  incluir o critério Todas-Definições;
3. requerer um conjunto de casos de teste finito.

A principal desvantagem dos critérios baseados em análise de fluxo de dados é que na presença de caminhos não executáveis estes não garantem a inclusão do critério Todas-Arestas [141, 144]. Diz-se que tais critérios não estabelecem uma “ponte” (*bridge the gap*) entre os critérios Todas-Arestas e Todos-Caminhos. É importante ressaltar, ainda, que a maioria dos programas reais contém caminhos não executáveis.

Na prática, ao aplicar-se um critério de teste estrutural aplica-se, na realidade, o correspondente critério estrutural executável. Assim, é de fundamental importância a definição e o estudo das propriedades dos critérios na presença de caminhos não executáveis. Nesse contexto, no início dos anos 90, Maldonado [261] definiu a família de critérios Potenciais-Usos e a correspondente família de critérios executáveis.

Em linhas gerais, os critérios Potenciais-Usos requerem associações independentemente da ocorrência explícita de uma referência (um uso) a uma definição de variável. Ou seja, se um uso dessa definição pode existir (há um caminho livre de definição até um certo nó ou arco, isto é, um potencial-uso), a potencial-associação entre a definição e o potencial-uso é caracterizada e eventualmente requerida. Dito de outra maneira, tais critérios requerem que caminhos livres de definição em relação a qualquer nó  $i$  que possua definição de variável e a qualquer variável  $x$  definida em  $i$  sejam executados, independentemente de ocorrer uso dessa variável nesses caminhos. Desse modo, é possível verificar, por exemplo, que o valor de  $x$  não foi alterado nesses caminhos (possivelmente devido a efeitos colaterais), ganhando-se maior confiança de que a computação correta é realizada.

Para o estabelecimento da família de critérios Potenciais-Usos definem-se [261]:

- $\text{defg}(i) = \{\text{variáveis } v \mid v \text{ é definida no nó } i\}$ ;
- $\text{pdcu}(x, i) = \{\text{nós } j \mid \text{existe um caminho livre de definição c.r.a. } x \text{ do nó } i \text{ para o nó } j\}$ ;
- $\text{pdpu}(x, i) = \{\text{arcos } (j, k) \mid \text{existe um caminho livre de definição c.r.a. } x \text{ do nó } i \text{ para o arco } (j, k)\}$ ;
- “potencial-du-caminho” c.r.a variável  $x$  como um caminho livre de definição  $(n_1, n_2, \dots, n_j, n_k)$  c.r.a  $x$  do nó  $n_1$  para o nó  $n_k$  e para o arco  $(n_j, n_k)$ , tal que o caminho  $(n_1, n_2, \dots, n_j)$  é um caminho livre de laço e no nó  $n_1$  ocorre uma definição de  $x$ ;
- “associação potencial-definição-c-uso” como a tripla  $\langle i, j, x \rangle$ , em que  $x \in \text{defg}(i)$  e  $j \in \text{pdcu}(x, i)$ ;
- “associação potencial-definição-p-uso” como a tripla  $\langle i, (j, k), x \rangle$ , em que  $x \in \text{defg}(i)$  e  $(j, k) \in \text{pdpu}(x, i)$ ;

- “potencial associação” como um potencial-du-caminho, uma associação potencial-de-definição-c-uso ou uma associação potencial-definição-p-uso. É importante observar que, por definição, toda associação é uma potencial-associação e todo uso é também um potencial-uso.

Da mesma forma como os demais critérios baseados em fluxo de dados, os critérios Potenciais-Usos podem utilizar o Grafo Def-Uso como base para o estabelecimento dos requisitos de teste. Na verdade, basta ter a extensão do GFC associando a cada nó  $i$  o conjunto  $\text{defg}(i)$ . O grafo assim estendido é denominado de “Grafo Def” [261].

A cada nó  $i$  tal que  $\text{defg}(i) \neq \phi$  é associado um grafo denominado  $\text{grafo}(i)$ , o qual contém todos os potenciais-du-caminhos com início no nó  $i$ . Cada nó  $k$  do  $\text{grafo}(i)$  é completamente identificado pelo número do nó do GFC que lhe deu origem e pelo conjunto  $\text{deff}(k)$  – o conjunto de variáveis definidas no nó  $i$ , mas que não são redefinidas em um caminho do nó  $i$  para o nó  $k$ . Observa-se que  $\text{deff}(k) \subseteq \text{defg}(i)$  e que  $\text{deff}(i) = \text{defg}(i)$ .

Introduz-se a notação  $\langle i, (j, k), \{v_1, \dots, v_n\} \rangle$  para representar o conjunto de associações  $\langle i, (j, k), v_1 \rangle, \dots, \langle i, (j, k), v_n \rangle$ ; ou seja,  $\langle i, (j, k), \{v_1, \dots, v_n\} \rangle$  indica que existe, no  $\text{grafo}(i)$ , pelo menos um caminho livre de definição c.r.a  $v_1, \dots, v_n$  do nó  $i$  ao arco  $(j, k)$ . Ressalta-se que podem existir no  $\text{grafo}(i)$  outros caminhos livres de definição c.r.a algumas das variáveis  $v_1, \dots, v_n$ , mas que não sejam, simultaneamente, livres de definição para todas elas.

Diz-se que um caminho  $\pi_1 = (i_1, \dots, i_k)$  está incluído em um conjunto  $\Pi$  de caminhos se e somente se  $\Pi$  contém um caminho  $\pi_2 = (n_1, \dots, n_m)$  tal que  $i_1 = n_j, i_2 = n_{j+1}, \dots, i_k = i_{j+k-1}$ , para algum  $j, 1 \leq j \leq m - k + 1$ . Diz-se que  $\pi_1$  é incluído em  $\pi_2$  ou que  $\pi_1$  é um subcaminho de  $\pi_2$ .

Um caminho completo  $\pi$  cobre uma associação potencial-definição-c-uso  $\langle i, j, x \rangle$  (respectivamente, uma associação potencial-definição-p-uso  $\langle i, (j, k), x \rangle$ ), se ele incluir um caminho livre de definição c.r.a  $x$  do nó  $i$  para o nó  $j$  (respectivamente, de  $i$  para o arco  $(j, k)$ ). O caminho  $\pi$  cobre um potencial-du-caminho  $\pi_1$  se  $\pi_1$  estiver incluído em  $\pi$ . Um conjunto  $\Pi$  de caminhos cobre uma potencial-associação se algum elemento do conjunto o fizer. Observa-se que, se um conjunto de caminhos  $\Pi$  cobrir uma associação ou uma potencial associação do nó  $i$  para o nó  $j$  (respectivamente, para o arco  $(j, k)$ ), então existe um caminho livre de definição c.r.a variável  $x \in \text{defg}(i)$  do nó  $i$  para o nó  $j$  (respectivamente, para o arco  $(j, k)$ ), que é incluído em  $\Pi$ .

Um “caminho completo é executável ou factível” se existir um conjunto de valores que possa ser atribuído às variáveis de entrada do programa e que resulte na execução desse caminho; caso contrário, diz-se que ele é “não executável” [141]. Um “caminho é executável” se ele for um subcaminho de um caminho completo executável, isto é, se ele for incluído em um caminho completo executável. Uma “potencial-associação é executável” se existir um caminho completo executável que cubra essa associação; caso contrário, é “não executável”. Em função desses conceitos outros conjuntos são definidos [261]:

- $\text{fdcu}(x, i) = \{j \in \text{dcu}(x, i) \mid a \text{ associação } \langle i, j, x \rangle \text{ é executável}\};$
- $\text{fdpu}(x, i) = \{(j, k) \in \text{dpu}(x, i) \mid a \text{ associação } \langle i, (j, k), x \rangle \text{ é executável}\};$
- $\text{fpdcu}(x, i) = \{j \in \text{pdcu}(x, i) \mid a \text{ potencial-associação } \langle i, j, x \rangle \text{ é executável}\};$  e
- $\text{fpdpu}(x, i) = \{(j, k) \in \text{pdpu}(x, i) \mid a \text{ potencial-associação } \langle i, (j, k), x \rangle \text{ é executável}\}.$

A partir dessas definições, é possível estabelecer os critérios básicos que fazem parte da família de critérios Potenciais-Usos [261]:

- Todos-Potenciais-Usos: requer que pelo menos um caminho livre de definição de uma variável definida em um nó  $i$  para todo nó e todo arco possível de ser alcançado a partir de  $i$  seja exercitado;
- Todos-Potenciais-Usos/Du: requer que pelo menos um potencial-du-caminho com relação a uma variável  $x$  definida em  $i$  para todo nó e para todo arco possível de ser alcançado a partir de  $i$  seja exercitado;
- Todos-Potenciais-Du-Caminhos: requer que todos os potenciais-du-caminhos com relação a todas as variáveis  $x$  definidas e todos os nós e arcos possíveis de serem alcançados a partir dessa definição sejam exercitados.

Os critérios Potenciais-Usos são os únicos critérios baseados em fluxo de dados que satisfazem as três propriedades mínimas esperadas de um critério de teste  $C$  discutidas anteriormente; ou seja, estabelecem uma hierarquia entre os critérios Todas-Arestas e Todos-Caminhos, mesmo na presença de caminhos não executáveis. Além disso, nenhum outro critério de teste baseado em fluxo de dados inclui os critérios Potenciais-Usos. Outro ponto importante é que, apesar de terem complexidade de ordem exponencial, o que poderia ser um limitante para a aplicação efetiva desses critérios (assim como dos demais critérios baseados em fluxo de dados), na prática tais critérios requerem um número pequeno de casos de teste. Esses aspectos serão retomados e discutidos no Capítulo 10.

#### 4.4.4 Exemplo de aplicação

Para ilustrar a aplicação de alguns dos principais critérios estruturais serão utilizados o programa `identifier` (Programa 4.1) e o GFC correspondente (Figura 4.1), ambos apresentados na Seção 4.3. Com relação ao programa `identifier` é importante observar que os subcaminhos  $(1,3,4,8,9)$ ,  $(2,3,4,8,10)$  e  $(6,7,4,8,9)$  são não executáveis e, desse modo, quaisquer caminhos completos que os incluam também serão considerados não executáveis.

Considere inicialmente o critério de McCabe, baseado na complexidade. De acordo com o GFC, determina-se a complexidade ciclomática  $V(G)$  conforme uma das seguintes maneiras:

- $V(G) = 5$  regiões; ou
- $V(G) = 14$  arcos – 11 nós + 2 = 5; ou
- $V(G) = 4$  nós predicativos + 1 = 5.

O valor de  $V(G)$  fornece o número de caminhos linearmente independentes ao longo da estrutura de controle do programa. Assim, no caso do programa `identifier` espera-se especificar cinco caminhos, ilustrados na Tabela 4.2, que correspondem ao conjunto básico de caminhos linearmente independentes para o programa. As reticências (...) indicam que qualquer caminho ao longo do restante da estrutura de controle é aceitável. Observe que o caminho  $(1,2,3,4,8,10,11)$  é não executável, devendo ser descartado. Ressalta-se que o conjunto básico não é único e, assim, diferentes conjuntos básicos poderiam ser derivados.

Tabela 4.2 – Elementos requeridos pelo critério de McCabe

Caminhos Requeridos	Não executável
(1,2,3,4,8,9,11)	
(1,2,3,4,8,10,11)	×
(1,2,3,4,5,7,4, ...)	
(1,2,3,4,5,6,7,4, ...)	
(1,3,4, ...)	

Considere agora os critérios baseados em fluxo de controle – Todos-Nós e Todas-Arestas. Na Tabela 4.3 ilustram-se os elementos requeridos por esses critérios. É importante observar que no caso do critério Todas-Arestas apenas os arcos primitivos foram considerados. O conceito de arco primitivo foi introduzido por Chusho [82], tendo como base o fato de existirem arcos dentro de um GFC que são sempre executados quando outro arco é executado. Tais arcos são ditos não essenciais para a análise de cobertura. A utilização do conceito de arcos primitivos dentro do contexto do teste estrutural teve como objetivo viabilizar a descrição dos elementos requeridos pelos critérios de teste dessa natureza [261].

Tabela 4.3 – Elementos requeridos pelos critérios Todos-Nós e Todas-Arestas

Elementos Requeridos											
Nós	1	2	3	4	5	6	7	8	9	10	11
Arcos	(1,2)		(1,3)		(5,6)		(5,7)		(8,9)		(8,10)

Partindo-se para a classe de critérios baseados em fluxo de dados, considere os critérios Todas-Definições e Todos-Usos, propostos por Rapps e Weyuker [336, 337]. Na Figura 4.2 é ilustrado o Grafo Def-Usa do programa `identifier`.

Para exercitar a definição da variável `length` (nó 1) de acordo com o critério Todas-Definições, basta ser executado um dos seguintes subcaminhos: (1,3,4,5,7); (1,3,4,8,9); (1,3,4,8,10); e (1,3,4,5,6,7). Observe que o caminho (1,3,4,8,9) é não executável. Se qualquer um dos demais caminhos for exercitado, o requisito de teste será satisfeito. Para satisfazer o critério Todas-Definições, esta mesma análise deve ser realizada para todas as definições que ocorrem no programa. Na Tabela 4.4 são ilustrados os subcaminhos que podem ser executados para cada uma das definições de variável no programa. A seleção de pelo menos um subcaminho para cada definição caracteriza o conjunto de elementos requeridos pelo critério.

A título de ilustração, considere que para testar o programa `identifier` tenha sido utilizado um conjunto de casos de teste inicial ( $T_0$ ), gerado com o objetivo de satisfazer os requisitos de teste de um dado critério funcional como, por exemplo, o critério particionamento em classes de equivalência (Capítulo 2). Considerando  $T_0 = \{(a1, \text{Válido}), (2B3, \text{Inválido}), (Z-12, \text{Inválido}), (A1b2C3d, \text{Inválido})\}$ , observa-se que  $T_0$  também satisfaz os requisitos de teste dos critérios de McCabe, Todos-Nós e Todas-Arestas.

Em relação ao critério Todos-Usos, também com respeito à definição da variável `length` no nó 1, são requeridas as seguintes associações:  $\langle 1, 7, \text{length} \rangle$ ;  $\langle 1, (8, 9), \text{length} \rangle$  e

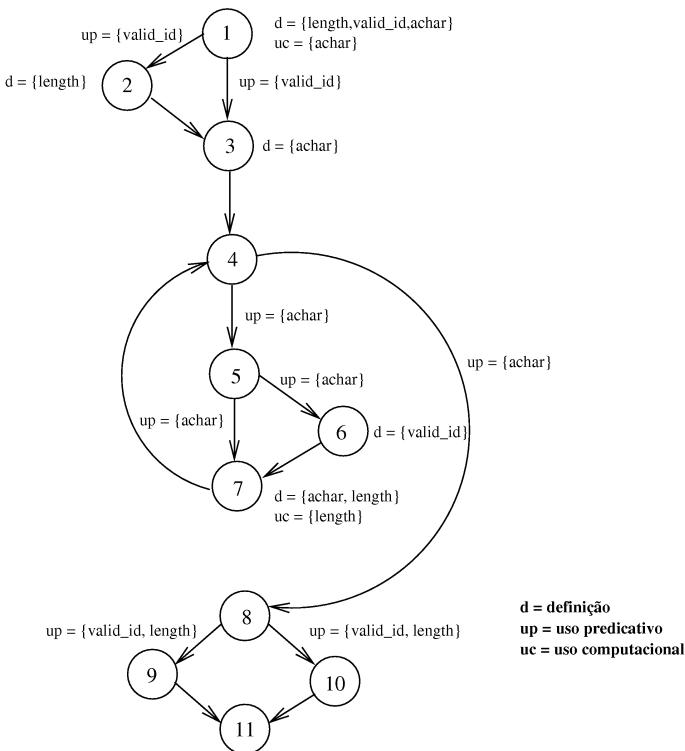


Figura 4.2 – Grafo Def-Uso do programa `identifier`.

$\langle 1, (8, 10), \text{length} \rangle$ .<sup>4</sup> Observe que a associação  $\langle 1, (8, 9), \text{length} \rangle$  é não executável – o único caminho livre de definição possível de exercitá-la seria um caminho que incluísse o subcaminho  $(1, 3, 4, 8, 9)$ . Já para a associação  $\langle 1, 7, \text{length} \rangle$  qualquer caminho completo executável que inclua um dos subcaminhos  $(1, 3, 4, 5, 6, 7)$  ou  $(1, 3, 4, 5, 7)$  é suficiente para exercitá-la. Essa mesma análise deve ser feita para todas as demais variáveis e associações pertinentes, a fim de satisfazer o critério. Na Tabela 4.5 é ilustrado o conjunto completo de associações requeridas pelo critério Todos-Usos.

Finalmente, considere o critério Todos-Potenciais-Usos, pertencente à família de critérios Potenciais-Usos [261]. Na Figura 4.3 é ilustrado o Grafo Def do programa `identifier`.

Utilizando-se o conceito de potencial-uso, tem-se que a variável `length` definida no nó 1 poderia ter um uso predutivo nos arcos  $(5, 6)$  e  $(5, 7)$ , fazendo com que as potenciais-associações  $\langle 1, (5, 6), \text{length} \rangle$  e  $\langle 1, (5, 7), \text{length} \rangle$  sejam requeridas pelo critério Todos-Potenciais-Usos. É importante observar que as potenciais-associações  $\langle 1, (5, 6), \text{length} \rangle$  e  $\langle 1, (5, 7), \text{length} \rangle$  não seriam requeridas pelos demais critérios de fluxo de dados que não fazem uso do conceito de potencial-uso. Note-se, ainda, que, conforme observado anterior-

<sup>4</sup>As notações  $\langle i, j, \text{var} \rangle$  e  $\langle i, (j, k), \text{var} \rangle$  indicam que a variável `var` é definida no nó  $i$  e existe um uso computacional de `var` no nó  $j$  ou um uso predutivo de `var` no arco  $(j, k)$ , respectivamente, bem como pelo menos um caminho livre de definição do nó  $i$  ao nó  $j$  ou ao arco  $(j, k)$ .

Tabela 4.4 – Elementos requeridos pelo critério Todas-Definições

Nó	Variável	Caminhos Requeridos	Não executável
1	length	(1,3,4,5,7)	
		(1,3,4,5,6,7)	
		(1,3,4,8,9)	×
		(1,3,4,8,10)	
	valid_id	(1,2,3,4,8,9)	
		(1,3,4,8,10)	
		(1,3,4,5,7)	
		(1,3,4,5,6)	
	achar	(1,3,4,8)	
		(2,3,4,5,7)	
		(2,3,4,5,6,7)	
		(2,3,4,8,9)	
2	length	(2,3,4,8,10)	×
		(3,4,5,7)	
		(3,4,5,6)	
		(3,4,8)	
3	achar	(6,7,4,8,9)	×
		(6,7,4,8,10)	
6	length	(7,4,5,7)	
		(7,4,5,6,7)	
		(7,4,8,9)	
		(7,4,8,10)	
	achar	(7,4,5,7)	
		(7,4,5,6)	
		(7,4,8)	

Tabela 4.5 – Elementos requeridos pelo critério Todos-Uso

Associações Requeridas	NE	Associações Requeridas	NE
$\langle 1, 7, \{length\} \rangle$		$\langle 3, (4, 5), \{achar\} \rangle$	
$\langle 1, (8, 9), \{length, valid_id\} \rangle$	×	$\langle 3, (4, 8), \{achar\} \rangle$	
$\langle 1, (8, 10), \{length, valid_id\} \rangle$		$\langle 3, (5, 6), \{achar\} \rangle$	
$\langle 1, (1, 2), \{valid_id\} \rangle$		$\langle 3, (5, 7), \{achar\} \rangle$	
$\langle 1, (1, 3), \{valid_id\} \rangle$		$\langle 6, (8, 9), \{valid_id\} \rangle$	×
$\langle 1, 1, \{achar\} \rangle$		$\langle 6, (8, 10), \{valid_id\} \rangle$	
$\langle 1, (4, 5), \{achar\} \rangle$		$\langle 7, 7, \{length\} \rangle$	
$\langle 1, (4, 8), \{achar\} \rangle$		$\langle 7, (8, 9), \{length\} \rangle$	
$\langle 1, (5, 6), \{achar\} \rangle$		$\langle 7, (8, 10), \{length\} \rangle$	
$\langle 1, (5, 7), \{achar\} \rangle$		$\langle 7, (4, 5), \{achar\} \rangle$	
$\langle 2, 7, \{length\} \rangle$		$\langle 7, (4, 8), \{achar\} \rangle$	
$\langle 2, (8, 9), \{length\} \rangle$		$\langle 7, (5, 6), \{achar\} \rangle$	
$\langle 2, (8, 10), \{length\} \rangle$	×	$\langle 7, (5, 7), \{achar\} \rangle$	

NE – não-executável

mente, toda associação é uma potencial-associação e, desse modo, as associações requeridas pelo critério Todos-Uso são um subconjunto das potenciais-associações requeridas pelo critério Todos-Potenciais-Uso.

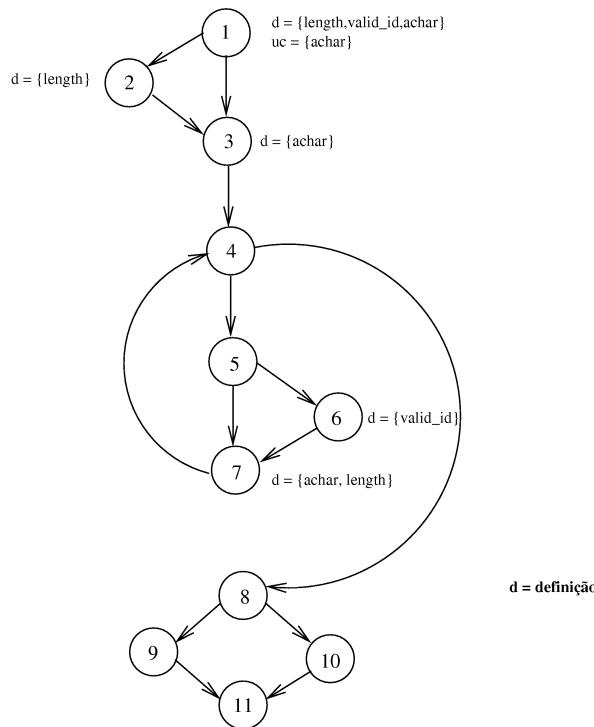


Figura 4.3 – Grafo Def do programa `identifier`.

O conjunto completo de associações requeridas pelo critério Todos-Potenciais-Usos está representado na Tabela 4.6.

Partindo-se do conjunto  $T_0$ , ilustrado anteriormente, seria necessária a inclusão de quatro novos casos de teste de modo a satisfazer os requisitos de teste dos critérios Todas-Definições, Todos-Usos e Todos-Potenciais-Usos:  $T_2 = T_0 \cup \{(1\#, Inválido), (\%, Inválido), (c, Válido), (\#-%, Inválido)\}$ .

## 4.5 Ferramentas

A aplicação de critérios de teste sem o apoio de ferramentas automatizadas tende a ser uma atividade propensa a erros e limitada a programas muito simples. Tal aspecto motiva o desenvolvimento de ferramentas automáticas para auxiliar na condução de testes efetivos e na análise dos resultados obtidos.

As ferramentas de apoio ao teste estrutural possibilitam, em sua maioria, a análise de cobertura de um conjunto de casos de teste segundo algum critério selecionado. Algumas delas oferecem suporte à obtenção dos dados de entrada necessários para satisfazer um particular requisito de teste. Em geral, tal suporte é baseado na execução simbólica do programa em teste. Apesar de não auxiliarem diretamente na determinação das entradas de um programa,

Tabela 4.6 – Elementos requeridos pelo critério Todos-Potenciais-Usos

Associações Requeridas	NE	Associações Requeridas	NE
$\langle 1, (6, 7), \{\text{length}\} \rangle$		$\langle 2, (6, 7), \{\text{length}\} \rangle$	
$\langle 1, (1, 3), \{\text{achar}, \text{length}, \text{valid\_id}\} \rangle$		$\langle 2, (5, 6), \{\text{length}\} \rangle$	
$\langle 1, (8, 10), \{\text{length}, \text{valid\_id}\} \rangle$		$\langle 3, (8, 10), \{\text{achar}\} \rangle$	
$\langle 1, (8, 10), \{\text{valid\_id}\} \rangle$		$\langle 3, (8, 9), \{\text{achar}\} \rangle$	
$\langle 1, (8, 9), \{\text{length}, \text{valid\_id}\} \rangle$	$\times$	$\langle 3, (5, 7), \{\text{achar}\} \rangle$	
$\langle 1, (8, 9), \{\text{valid\_id}\} \rangle$		$\langle 3, (6, 7), \{\text{achar}\} \rangle$	
$\langle 1, (7, 4), \{\text{valid\_id}\} \rangle$		$\langle 3, (5, 6), \{\text{achar}\} \rangle$	
$\langle 1, (5, 7), \{\text{length}, \text{valid\_id}\} \rangle$		$\langle 6, (8, 10), \{\text{valid\_id}\} \rangle$	
$\langle 1, (5, 7), \{\text{valid\_id}\} \rangle$		$\langle 6, (8, 9), \{\text{valid\_id}\} \rangle$	$\times$
$\langle 1, (5, 6), \{\text{length}, \text{valid\_id}\} \rangle$		$\langle 6, (5, 7), \{\text{valid\_id}\} \rangle$	
$\langle 1, (5, 6), \{\text{valid\_id}\} \rangle$		$\langle 6, (5, 6), \{\text{valid\_id}\} \rangle$	
$\langle 1, (2, 3), \{\text{achar}, \text{valid\_id}\} \rangle$		$\langle 7, (8, 10), \{\text{achar}, \text{length}\} \rangle$	
$\langle 1, (1, 2), \{\text{achar}, \text{length}, \text{valid\_id}\} \rangle$		$\langle 7, (8, 9), \{\text{achar}, \text{length}\} \rangle$	
$\langle 2, (8, 10), \{\text{length}\} \rangle$	$\times$	$\langle 7, (5, 7), \{\text{achar}, \text{length}\} \rangle$	
$\langle 2, (8, 9), \{\text{length}\} \rangle$		$\langle 7, (6, 7), \{\text{achar}, \text{length}\} \rangle$	
$\langle 2, (5, 7), \{\text{length}\} \rangle$		$\langle 7, (5, 6), \{\text{achar}, \text{length}\} \rangle$	

NE – não-executável

necessárias para a execução de caminhos específicos, grande parte das ferramentas de teste apresenta ao usuário os requisitos de teste exigidos para que os critérios sejam satisfeitos. Assim, de certa maneira, orientam e auxiliam o usuário na elaboração dos casos de teste [261].

É importante observar que a existência de caminhos não executáveis é um aspecto bastante restritivo para a automatização dos critérios estruturais, assim como da atividade de teste de um modo geral. Heurísticas têm sido introduzidas para lidar com esse aspecto [141].

As primeiras ferramentas comerciais de apoio ao teste estrutural foram a *RXVP80* [118] e a *TCAT* [341], as quais apoiavam a aplicação de critérios estruturais baseados somente no Grafo de Fluxo de Controle. A ferramenta *RXVP80*, distribuída pela General Research Corporation, é um sistema que realiza basicamente a análise de cobertura do teste de arcos em programas escritos em Fortran. Além do suporte ao teste dinâmico, via instrumentação, essa ferramenta fornece ainda: análise estática do código-fonte com a geração do Grafo de Chamada dos módulos (unidades) que compõem o sistema em teste; geração do GFC dos módulos; verificação de anomalias no código-fonte; verificação de tipos nas chamadas de procedimentos; e geração de alguns relatórios que fornecem referências cruzadas entre código-fonte, variáveis e módulos.

A ferramenta *TCAT* (Test-Coverage Analysis Tool), distribuída pela Software Research Corporation, realiza o teste de unidades segundo o critério Teste de Ramos Lógicos (arcos). Esse tipo de teste de ramos divide os predicados encontrados em uma condição em vários “if’s”, no qual cada “if” só contém um dos predicados. Essa ferramenta insere código adicional (além do de monitoração) para mapear os ramos lógicos.

A ferramenta *SCORE* [82] apóia o teste de arcos de programas escritos em Pascal, tendo sido desenvolvida na Hitachi. Implementa duas medidas de cobertura distintas. A primeira medida é baseada na forma tradicional que consiste no quociente entre o número de arcos executados e o número total de arcos. A segunda medida baseia-se na aplicação do conceito

de arco primitivo e consiste no quociente entre o número de arcos primitivos executados e o número total de arcos primitivos. Essas medidas possibilitam verificar a efetividade do uso do conceito de arcos primitivos no contexto de teste estrutural, pois evitam uma superestimativa da cobertura de arcos.

Na década de 1980, com a introdução dos critérios baseados em análise de fluxo de dados, começaram a surgir ferramentas de apoio à aplicação de tais critérios. As ferramentas desenvolvidas eram protótipos de pesquisa construídos em universidades [178, 141, 222, 434], não estando disponíveis comercialmente. De modo geral, tais ferramentas diferenciavam-se na abordagem utilizada para determinação dos requisitos de teste, nas características quanto à interface com o usuário, nas facilidades para determinação dos dados de entrada, nas facilidades para tratamento de várias linguagens e no gerenciamento da sessão de teste.

Uma das primeiras iniciativas de automatização foi a ferramenta *Asset (A System to Select and Evaluate Tests)* [142], desenvolvida na New York University. A ferramenta apóia a aplicação dos critérios baseados em fluxo de dados definidos por Rapps e Weyuker [336, 337] no teste de programas Pascal.

A *Proteste* [330], desenvolvida na Universidade Federal do Rio Grande do Sul, consiste em um protótipo de apoio ao teste estrutural de programas escritos em Pascal, incluindo tanto critérios baseados em fluxo de controle (Todos-Nós e Todas-Arestas) quanto critérios baseados em fluxo de dados (os definidos por Rapps e Weyuker [336, 337] e os Potenciais-Usos [261]). Além de Pascal, o ambiente pode ser configurado para outras linguagens por meio da utilização de uma ferramenta que gera analisadores de código-fonte específicos para cada linguagem.

Desenvolvida na FEEC/UNICAMP em colaboração com o ICMC/USP, a ferramenta *POKE-TOOL (Potential Uses Criteria Tool for Program Testing)* [61, 263] apóia a aplicação dos critérios Potenciais-Usos e também de outros critérios estruturais baseados em fluxo de controle e fluxo de dados. Inicialmente a ferramenta foi desenvolvida para o teste de unidade de programas escritos em C [61] e, posteriormente, devido à sua característica de multilinguagem, também foram propostas configurações para o teste de programas em Cobol e Fortran [140, 234]. A *POKE-TOOL* será apresentada em detalhes na próxima seção.

No contexto das pesquisas mais recentes destaca-se o ambiente  $\chi$ Suds, desenvolvido pela *Telcordia Technologies* [6, 389]. A  $\chi$ Suds consiste em um conjunto de sete ferramentas ( $\chi$ ATAC,  $\chi$ Rexx,  $\chi$ Vue,  $\chi$ Slice,  $\chi$ Prof,  $\chi$ Find e  $\chi$ Diff) destinadas ao entendimento, à análise e ao teste de software para códigos C/C++. Entre as sete ferramentas, a responsável pela análise de cobertura é a  $\chi$ ATAC. Tal ferramenta mede o quanto a aplicação está sendo exercitada pelo conjunto de teste, identificando códigos que não estão suficientemente testados e determinando a cobertura da aplicação com os casos de teste. Essas medidas são usadas para indicar o progresso durante o teste do software e servem de critério de aceitação para subseqüentes estágios do desenvolvimento e da atividade de teste. Para o teste de programas C, a  $\chi$ ATAC apóia a aplicação dos critérios de fluxo de controle (Todos-Nós e Todas-Arestas) e de fluxo de dados (Todos-c-Usos, Todos-p-Usos e Todos-Usos). Para o teste de programas C++, somente os critérios de fluxo de controle estão disponíveis.

Por fim, conforme observado anteriormente, os critérios baseados em fluxo de controle e em fluxo de dados também têm sido investigados no teste de programas OO e de componentes [419]. Como consequência, uma série de ferramentas comerciais e não comerciais encontra-se disponível, sobretudo para o teste de programas escritos em C++ e Java. A fer-

ramento JaBUTi (*Java Bytecode Understanding and Testing*) [422], por exemplo, apóia a aplicação do teste estrutural intramétodo em programas e componentes Java. Basicamente, a ferramenta fornece suporte a diferentes critérios de teste estruturais para a análise de cobertura, um conjunto de métricas estáticas para avaliar a complexidade das classes que compõem o programa/componente e, ainda, implementa algumas heurísticas de particionamento de programas que buscam auxiliar a localização de defeitos. Outras versões da ferramenta JaBUTi também estão disponíveis: (1) JaBUTi/AJ (*Java Bytecode Understanding and Testing/AspectJ*) [235], para o teste estrutural de unidade de programas orientados a aspectos (OA), baseados na linguagem AspectJ; (2) JaBUTi/MA (*Java Bytecode Understanding and Testing/Mobile Agents*) [112], para o teste estrutural de agentes móveis; e (3) JaBUTi/DB (*Java Bytecode Understanding and Testing/Database*) [297], para o teste estrutural de aplicações de banco de dados. Uma síntese das principais ferramentas de apoio ao teste estrutural de programas OO e de componentes bem como detalhes referentes à ferramenta JaBUTi podem ser encontrados no Capítulo 6.

#### 4.5.1 POKE-TOOL

Nesta seção são apresentadas as principais características da ferramenta de teste POKE-TOOL (*Potential Uses Criteria Tool for Program Testing*) [61, 263]. A ferramenta apóia a aplicação dos critérios Todos-Nós, Todas-Arestas e os critérios básicos da família Potenciais-Usos (Todos-Potenciais-Usos, Todos-Potenciais-Usos/Du e Todos-Potenciais-Du-Caminhos) no teste de unidade de programas escritos na linguagem C.

A ferramenta POKE-TOOL encontra-se disponível em ambiente UNIX, possuindo módulos funcionais cuja utilização se dá por meio de interface gráfica ou linha de comando (*shell scripts*). A interface gráfica permite ao usuário iniciante explorar e aprender os conceitos relacionados ao critério em uso e à própria ferramenta. Além disso, oferece melhores recursos para a visualização dos casos de teste e dos requisitos de teste. Embora conduzir uma sessão de teste pela interface gráfica seja provavelmente mais fácil, esta é menos flexível do que quando se utiliza a chamada direta aos programas que compõem as ferramentas. A interface gráfica depende de constante interação do testador, ao passo que a utilização de scripts possibilita a execução de longas sessões de teste em *batch*. O usuário pode construir um programa especificando o teste a ser realizado e a ferramenta simplesmente executa esse programa, permitindo que se economize tempo na atividade de teste devido à redução do número de interações com a ferramenta. Por outro lado, a elaboração de scripts exige um esforço de programação e completo domínio tanto dos conceitos sobre o teste quanto dos próprios programas que compõem a ferramenta, devendo ser utilizado pelo testador mais experiente. Scripts de teste têm se mostrado de grande utilidade na condução de estudos experimentais, em que uma mesma seqüência de passos deve ser executada várias vezes até que os resultados obtidos sejam significativos do ponto de vista estatístico.

A POKE-TOOL foi projetada como uma ferramenta interativa cuja operação é orientada à sessão de teste. O termo “sessão de teste” (ou sessão de trabalho) é utilizado para designar as atividades envolvendo um teste, a saber [261]: análise estática da unidade; preparação para o teste; submissão de casos de teste; avaliação de casos de teste; e gerenciamento dos resultados de teste.

A sessão de teste é dividida em duas fases: uma estática e outra dinâmica. Na fase estática, a ferramenta analisa o código-fonte, obtém as informações necessárias para a aplicação dos

critérios e instrumenta o código com inserção de pontas de prova (instruções de escrita), as quais produzem o rastreamento (*trace*) do caminho executado, gerando uma nova versão da unidade em teste – versão instrumentada –, que viabiliza a posterior avaliação da adequação de um dado conjunto de casos de teste. Terminada a fase estática, a POKE-TOOL apresenta, sob solicitação do usuário, o conjunto de elementos requeridos (nós, arcos, caminhos ou associações) para satisfazer o critério selecionado. Com base em tais informações, o usuário pode projetar os casos de teste a fim de que eles executem os elementos exigidos.

A fase dinâmica consiste no processo de execução e avaliação de casos de teste. No entanto, antes de executar os casos de teste, é necessário que se gere o programa executável a partir da versão instrumentada da unidade em teste. Depois de executados os casos de teste, estes são avaliados de acordo com o critério selecionado. O resultado da avaliação é o conjunto de elementos requeridos que restam ser executados para satisfazer o critério e o percentual de cobertura obtido pelo conjunto de casos de teste. Ainda nessa fase, a ferramenta fornece o conjunto de elementos que foram executados, as entradas e saídas, bem como os caminhos percorridos pelos casos de teste. O processo de execução/avaliação deve continuar até que todos os elementos restantes tenham sido satisfeitos, executados ou detectada a sua não executabilidade.

Eventualmente, o usuário pode desejar interromper a sessão de teste. Nesse caso, a POKE-TOOL fornece meios para interrupção da sessão e armazenamento dos dados gerados e do estado do teste até o momento. Posteriormente, pode-se recuperar a sessão de teste e recomeçar o teste a partir do ponto em que este foi interrompido.

A versão mais recente da POKE-TOOL também apóia a aplicação de alguns dos critérios de Rapps e Weyuker [336, 337] – Todos-Usos, Todos-c-Usos e Todos-p-Usos. Entretanto, essa versão ainda não possui interface gráfica, sendo utilizada via linha de comando. Tal versão está disponível na Incubadora Virtual da FAPESP (<http://incubadora.fapesp.br/projects/poketool/>).

A seguir, é ilustrado o desenvolvimento de uma sessão de teste simples utilizando a POKE-TOOL. Considere-se, para isso, o programa `identifier` (Programa 4.1).

Na Figura 4.4 é apresentada a tela principal da ferramenta e as funções fornecidas. Para testar o programa deve-se primeiramente criar uma sessão de teste, fornecendo alguns parâmetros. Basicamente, o usuário entra com o programa a ser testado e seleciona todos ou alguns dos critérios disponíveis. Na Figura 4.5 é ilustrada a tela de criação de uma sessão de teste para o programa `identifier`, utilizando todos os critérios apoiados pela ferramenta.

Após sua criação, é possível adicionar casos de teste à sessão. A POKE-TOOL possibilita que casos de teste sejam inseridos/removidos dinamicamente ou, ainda, importados (de arquivos texto, de outras sessões de teste da própria POKE-TOOL, ou de sessões de teste da *Proteum* [107] – uma ferramenta de suporte ao teste baseado em mutação que será apresentada no Capítulo 5). Na hipótese de o conjunto de casos de teste fornecido ser vazio, a ferramenta apresenta o conjunto de elementos necessários para satisfazer o critério de teste, orientando, assim, a seleção de casos de teste. Se o conjunto de casos de teste fornecido não for vazio, é produzida uma relação de elementos requeridos pelo critério mas ainda não executados. Na Figura 4.6 é ilustrada parte das associações requeridas pelos critérios Todos-Potenciais-Usos e Todos-Potenciais-Usos/Du quando fornecido um conjunto de casos de teste vazio.

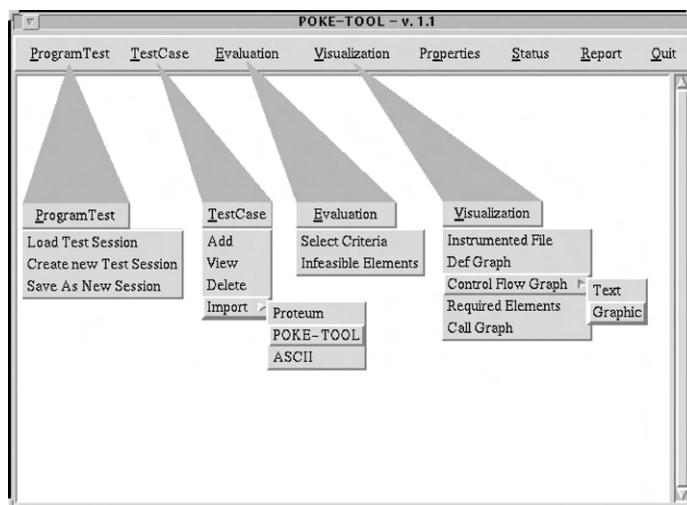


Figura 4.4 – Opções disponíveis na ferramenta POKE-TOOL.

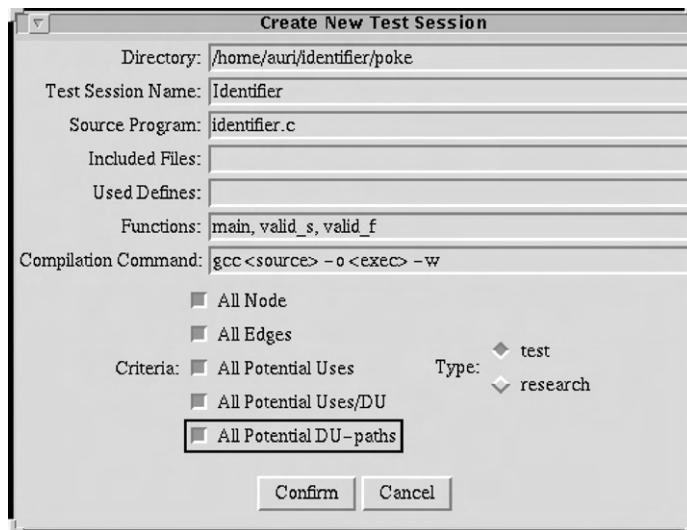


Figura 4.5 – Tela para criar uma sessão de teste na POKE-TOOL.

Após a entrada do conjunto de casos de teste, a ferramenta pode iniciar a execução do programa. Para verificar se o programa comportou-se corretamente, a POKE-TOOL possibilita a opção de visualização dos casos de teste. Com isso, pode-se verificar se o resultado produzido corresponde ao esperado e, portanto, se o programa foi executado com sucesso. A ferramenta permite, ainda, habilitar ou desabilitar os casos de teste – um caso de teste desabi-

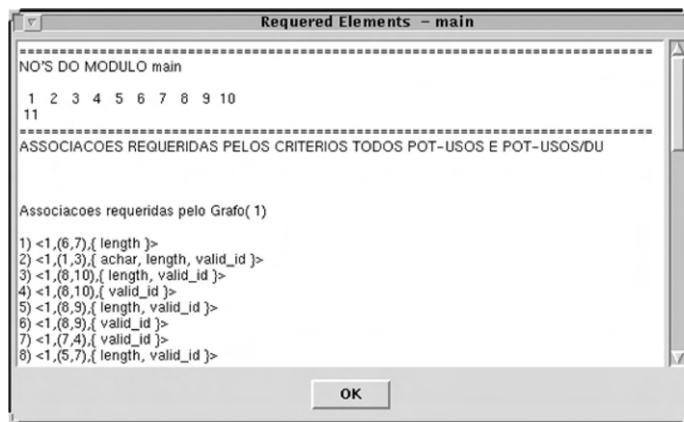


Figura 4.6 – Parte dos elementos requeridos pelos critérios Todos-Potenciais-Usos e Todos-Potenciais-Usos/Du.

litado continua fazendo parte do conjunto de casos de teste, mas não é utilizado em execuções posteriores até que seja novamente habilitado.

A título de ilustração, considere o conjunto de casos de teste  $T_0 = \{(a1, Válido), (2B3, Inválido), (Z-12, Inválido), (A1b2C3d, Inválido)\}$ , gerado a fim de satisfazer o critério Participionamento em Classes de Equivalência (Capítulo 2) para o teste do programa `identifier`. Para a condução dos testes estruturais, considere os critérios Todas-Arestas e Todos-Potenciais-Usos.

Nas Figuras 4.7 e 4.8 é apresentada a cobertura obtida com a execução do conjunto de casos de teste  $T_0$  em relação aos critérios Todas-Arestas e Todos-Potenciais-Usos, respectivamente. Os elementos requeridos e os elementos não executados para ambos os critérios também são ilustrados como parte dos relatórios gerados pela ferramenta. No caso do critério Todas-Arestas, não há arcos não executados, visto que a cobertura atingida foi de 100%.

É interessante observar que somente com os casos de teste funcionais foi possível cobrir o critério Todas-Arestas, ao passo que para se cobrir o critério Todos-Potenciais-Usos ainda é necessário analisar as associações que não foram executadas. Deve-se ressaltar que o conjunto  $T_0$  é Todas-Arestas-adequado, ou seja, o critério Todas-Arestas foi satisfeito; no entanto, o programa `identifier` contém um defeito cuja presença não foi revelada. Certamente, um conjunto adequado ao critério Todas-Arestas que revelasse a presença do defeito poderia ter sido gerado; o que se ilustra aqui é que isso nem sempre acontece.

Caso se deseje melhorar a cobertura em relação ao critério Todos-Potenciais-Usos, novos casos de teste devem ser inseridos com vistas a cobrir as associações que ainda não foram executadas. Antes disso, porém, deve-se verificar entre as associações não executadas se existem associações não executáveis. A sessão de teste deve prosseguir nessa iteração de analisar elementos não executáveis e inserir casos de teste até que um conjunto adequado ao critério seja obtido. No caso, as associações  $\langle 1, (8, 9), \{length, valid_id\} \rangle$ ,  $\langle 2, (8, 10), \{length\} \rangle$  e  $\langle 6, (8, 9), \{valid_id\} \rangle$  são não executáveis.

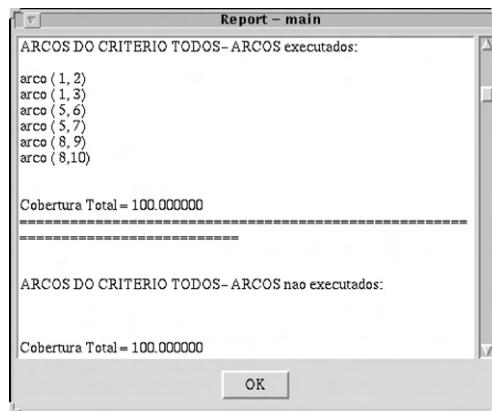


Figura 4.7 – Relatório gerado pela POKETOOL para o critério Todas-Arestas.

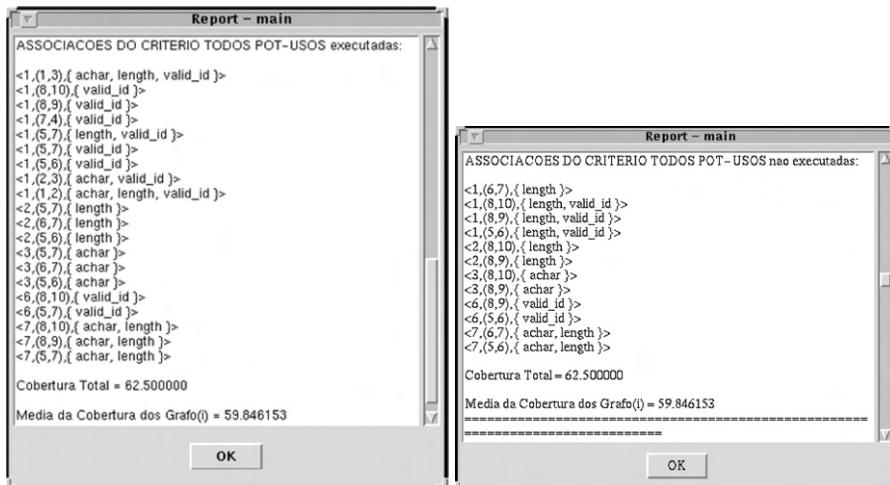


Figura 4.8 – Relatórios gerados pela POKETOOL para o critério Todos-Potenciais-Usos.

Na Tabela 4.7 é ilustrada a evolução da sessão de teste até que se atinja a cobertura de 100% para o critério Todos-Potenciais-Usos. Do total de 29 associações executáveis requeridas pelo critério Todos-Potenciais-Usos, 20 foram cobertas pelo conjunto  $T_0$ . A fim de cobrir as associações restantes, três novos casos de teste foram adicionados, resultando no conjunto  $T_1 = T_0 \cup \{(1\#, Inválido), (\%, Inválido), (c, Válido)\}$  e na cobertura de mais seis associações. As três associações restantes foram cobertas com a adição de um novo caso de teste ao conjunto  $T_1$ , totalizando oito casos de teste necessários para satisfazer o critério –  $T_2 = T_1 \cup \{(\#-\%, Inválido)\}$ . O símbolo ✓ indica quais associações foram cobertas por quais conjuntos de casos de teste e o símbolo ✗ mostra quais são as associações não executáveis.

Tabela 4.7 – Evolução da sessão de teste para cobrir o critério Todos-Potenciais-Usos

Associações Requeridas	$T_0$	$T_1$	$T_2$
$\langle 1, (6, 7), \{\text{length}\} \rangle$		✓	
$\langle 1, (1, 3), \{\text{achar}, \text{length}, \text{valid\_id}\} \rangle$	✓		
$\langle 1, (8, 10), \{\text{length}, \text{valid\_id}\} \rangle$		✓	
$\langle 1, (8, 10), \{\text{valid\_id}\} \rangle$	✓		
$\langle 1, (8, 9), \{\text{length}, \text{valid\_id}\} \rangle$	✗	✗	✗
$\langle 1, (8, 9), \{\text{valid\_id}\} \rangle$	✓		
$\langle 1, (7, 4), \{\text{valid\_id}\} \rangle$	✓		
$\langle 1, (5, 7), \{\text{length}, \text{valid\_id}\} \rangle$	✓		
$\langle 1, (5, 7), \{\text{valid\_id}\} \rangle$	✓		
$\langle 1, (5, 6), \{\text{length}, \text{valid\_id}\} \rangle$		✓	
$\langle 1, (5, 6), \{\text{valid\_id}\} \rangle$	✓		
$\langle 1, (2, 3), \{\text{achar}, \text{valid\_id}\} \rangle$	✓		
$\langle 1, (1, 2), \{\text{achar}, \text{length}, \text{valid\_id}\} \rangle$	✓		
$\langle 2, (8, 10), \{\text{length}\} \rangle$	✗	✗	✗
$\langle 2, (8, 9), \{\text{length}\} \rangle$		✓	
$\langle 2, (5, 7), \{\text{length}\} \rangle$	✓		
$\langle 2, (6, 7), \{\text{length}\} \rangle$	✓		
$\langle 2, (5, 6), \{\text{length}\} \rangle$	✓		
$\langle 3, (8, 10), \{\text{achar}\} \rangle$		✓	
$\langle 3, (8, 9), \{\text{achar}\} \rangle$		✓	
$\langle 3, (5, 7), \{\text{achar}\} \rangle$	✓		
$\langle 3, (6, 7), \{\text{achar}\} \rangle$	✓		
$\langle 3, (5, 6), \{\text{achar}\} \rangle$	✓		
$\langle 6, (8, 10), \{\text{valid\_id}\} \rangle$	✓		
$\langle 6, (8, 9), \{\text{valid\_id}\} \rangle$	✗	✗	✗
$\langle 6, (5, 7), \{\text{valid\_id}\} \rangle$	✓		
$\langle 6, (5, 6), \{\text{valid\_id}\} \rangle$			✓
$\langle 7, (8, 10), \{\text{achar}, \text{length}\} \rangle$	✓		
$\langle 7, (8, 9), \{\text{achar}, \text{length}\} \rangle$	✓		
$\langle 7, (5, 7), \{\text{achar}, \text{length}\} \rangle$	✓		
$\langle 7, (6, 7), \{\text{achar}, \text{length}\} \rangle$			✓
$\langle 7, (5, 6), \{\text{achar}, \text{length}\} \rangle$			✓
$T_0 = \{(a1, Válido), (2B3, Inválido), (Z-12, Inválido), (A1b2C3d, Inválido)\}$			
$T_1 = T_0 \cup \{(1#, Inválido), (%, Inválido), (c, Válido)\}$			
$T_2 = T_1 \cup \{(\#%, Inválido)\}$			

Na Figura 4.9 é ilustrado o estado final da sessão de teste. No exemplo, o conjunto de casos de teste  $T_2$  foi suficiente para satisfazer todos os elementos executáveis requeridos de cada um dos critérios estruturais suportados pela POKE-TOOL. Note-se que o andamento geral da sessão de teste pode ser conferido a qualquer momento durante a condução dos testes por meio da opção de status disponível na ferramenta.

Finalmente, é importante observar que, mesmo tendo satisfeito um critério mais rigoroso como o critério Todos-Potenciais-Usos, o defeito no programa `identifier` ainda não foi revelado. O defeito está presente no nó 8 (linha 24) e está relacionado ao número máximo de caracteres que um identificador válido deve conter. Trata-se de um defeito sensível a dado, o qual poderia ter sido revelado, por exemplo, se tivesse sido escolhido o caso de

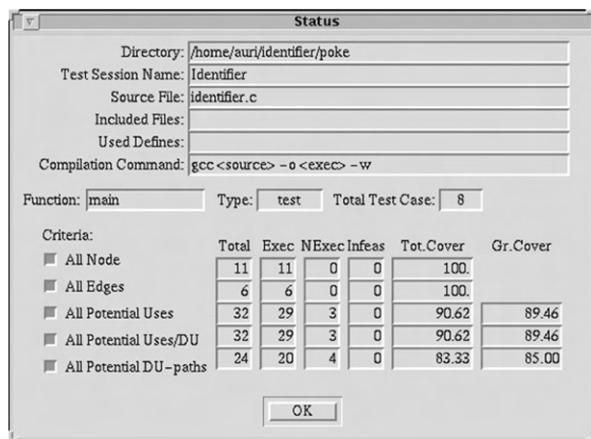


Figura 4.9 – Estado final da sessão de teste.

teste (ABCDEF, Válido) a fim de exercitar a associação  $\langle 1, (8, 10), \text{length} \rangle$ . Por outro lado, utilizando-se o critério Análise de Mutantes (apresentado no Capítulo 5), esse defeito teria sido necessariamente revelado, já que, ao se tentar “matar” os mutantes gerados pela aplicação do critério, o testador seria “forçado” a escolher esse tipo de caso de teste a fim de mostrar a diferença de comportamento entre o programa original e os programas mutantes. Essa análise fica como exercício para o leitor.

Na realidade, considerando esse contexto, motiva-se a pesquisa de critérios de teste que exercitem os elementos requeridos com maior probabilidade de revelar a presença de defeitos [404]. Além disso, outra perspectiva que se coloca é utilizar uma estratégia de teste incremental, a qual informalmente procurou-se ilustrar nesta seção. Em primeiro lugar, foram exercitados os requisitos de teste requeridos pelo critério funcional Particionamento em Classes de Equivalência (apresentado no Capítulo 2). Em seguida, foram considerados os critérios estruturais Todas-Arestas e Todos-Potenciais-Uso. Posteriormente, poder-se-ia considerar o critério Análise de Mutantes.

## 4.6 Considerações finais

Neste capítulo foram apresentados e discutidos os principais aspectos associados ao teste estrutural de programas. A técnica estrutural é baseada no conhecimento da estrutura interna da implementação e objetiva caracterizar um conjunto de elementos básicos do software que devem ser exercitados. Informalmente, a estrutura interna do programa é representada por um Grafo de Fluxo de Controle (GFC) a partir do qual são estabelecidos os critérios de teste. Diversos critérios estruturais de teste de unidade foram ilustrados. O uso de informações de fluxo de dados para derivar os requisitos de teste é um enfoque promissor – uma interpretação possível do teste baseado em análise de fluxo de dados é que comandos que têm uma relação de fluxo de dados são provavelmente partes de uma mesma função e devem ser exercitados (testados) juntamente.

É importante observar que os requisitos de teste exigidos pelos critérios estruturais discutidos neste capítulo limitam-se ao escopo da unidade. De fato, tais critérios foram inicialmente propostos para o teste de unidade de programas procedimentais. No entanto, esforços na tentativa de estender o uso de critérios estruturais para o teste de integração também podem ser identificados [160, 171, 200, 244].

Linnenkugel e Müllerburg [244] propuseram uma série de critérios que estendem os critérios baseados em fluxo de controle e em fluxo de dados para o teste de integração. O modelo de integração utilizado representa o programa por meio de um “Grafo de Chamada” (*call-graph*), no qual os módulos do programa são associados aos nós do grafo e o fluxo de controle (chamadas) é associado aos arcos. Os critérios de integração baseados em fluxo de controle são definidos sobre o Grafo de Chamada da mesma forma que os critérios para o teste de unidade são definidos sobre o GFC. Para definir os critérios de integração baseados em fluxo de dados, foi necessário estender o conceito de associação definição-uso para incluir caminhos interprocedimentais. Tais critérios baseiam-se na família de critérios de fluxo de dados definida por Rapps e Weyuker [336, 337].

Harrold e Soffa [171] também definiram uma abordagem para o teste de integração baseada nos critérios de Rapps e Weyuker. A principal diferença com o trabalho de Linnenkugel e Müllerburg é o modelo de integração utilizado. O programa é representado por um “Grafo de Fluxo de Dados Resumido” (*summary-graph*), no qual os nós representam regiões de código de interesse interprocedimental (entradas de procedimentos, saídas de procedimentos, chamadas de procedimentos e retornos) e os arcos representam informações de fluxo de controle.

Haley e Zweben [160] propuseram um critério para selecionar caminhos em um módulo que deveria ser testado novamente na fase de integração com base em sua interface. Jin e Offutt [200] definiram alguns critérios baseados em uma classificação de acoplamento entre módulos. Vilela [411], com base no conceito de potencial-uso, estendeu os critérios Potenciais-Usos para o teste de integração. Assim como Linnenkugel e Müllerburg, também é utilizado o Grafo de Chamada para representar a estrutura do programa em teste.

Uma das desvantagens do teste estrutural é a existência de elementos requeridos não executáveis. Existe também o problema de caminhos ausentes, ou seja, quando uma certa funcionalidade deixa de ser implementada no programa, não existe um caminho que corresponda àquela funcionalidade e, como consequência, nenhum caso de teste será requerido para exercitá-la. Mesmo assim, tais critérios estabelecem de forma rigorosa os requisitos de teste a serem exercitados, em termos de caminhos, associações definição-uso, ou outras estruturas do programa, fornecendo medidas objetivas sobre a adequação de um conjunto de teste para o teste de um dado programa. O rigor na definição dos requisitos favorece, ainda, a automatização desses critérios.

Por fim, é importante ressaltar o aspecto complementar das diversas técnicas e critérios de teste e a relevância de se conduzirem estudos experimentais para a formação de um corpo de conhecimento que favoreça o estabelecimento de estratégias de teste incrementais que explorem as diversas características dos critérios. Nessas estratégias seriam aplicados inicialmente critérios “mais fracos” e talvez menos eficazes para a avaliação da adequação do conjunto de casos de teste e, em função da disponibilidade de orçamento e de tempo, incrementalmente, poderiam ser utilizados critérios mais “fortes” e eventualmente mais eficazes, porém, em geral, mais caros. Estudos experimentais têm sido conduzidos no sentido de avaliar os aspectos de custo, *strength* e eficácia dos critérios de teste, buscando contribuir para

o estabelecimento de estratégias de teste eficazes, de baixo custo e para a transformação do estado da prática, no que tange ao uso de critérios e ferramentas de teste. No caso de critérios estruturais, os estudos conduzidos têm mostrado que os custos de aplicação desses critérios não inviabilizam sua aplicação em programas reais. Uma visão geral a respeito dos principais estudos experimentais conduzidos no contexto de teste estrutural será apresentada no Capítulo 10.

# Capítulo 5

## Teste de Mutação

*Márcio Eduardo Delamaro (UNIVEM)*

*Ellen Francine Barbosa (ICMC/USP)*

*Auri Marcelo Rizzo Vincenzi (UNISANTOS)*

*José Carlos Maldonado (ICMC/USP)*

### 5.1 Introdução

Myers [294] destaca que o objetivo da atividade de teste é revelar defeitos, uma vez que não se pode, por meio da execução do programa, provar sua correção. Já que o teste bem-sucedido é o que revela a presença de um defeito, um bom caso de teste é o que tem alta probabilidade de revelar um defeito ainda não descoberto.

Em princípio, é impossível medir-se a probabilidade de um determinado caso teste revelar ou não um defeito. Por isso, técnicas de teste como a funcional ou estrutural buscam aumentar a probabilidade de revelar defeitos por meio da escolha sistemática de um conjunto de casos de teste que represente as principais características do domínio de entrada do programa em teste. Os critérios dessas técnicas subdividem o domínio em subdomínios e obrigam o testador a selecionar dados de teste em cada um dos subdomínios, na esperança de que o conjunto construído dessa maneira possa revelar possíveis defeitos. Por exemplo, ao definir que determinado caminho do programa deve ser exercitado, estabelece-se um subconjunto do domínio de entrada, formado por aqueles dados de teste capazes de executar tal caminho e do qual um caso de teste deve ser selecionado.

Embora essa divisão em subdomínios seja útil, uma vez que força a escolha de casos de teste com características distintas e que exercitem o programa de maneiras diversas, não existe uma relação direta entre um subdomínio e a capacidade de um caso de teste dele extraído em revelar defeitos. Por exemplo, dado um caminho a ser exercitado, no qual existe um defeito, podem existir no subdomínio correspondente casos de teste que revelem a presença do defeito, ou seja, que levem o programa a falhar, e outros que, apesar de executar o caminho desejado, não revelem a presença de defeitos.

O Teste de Mutação ou Análise de Mutantes, como também é conhecido, é um critério de teste da técnica baseada em defeitos. Nessa técnica são utilizados defeitos típicos do processo de implementação de software para que sejam derivados os requisitos de teste. No caso do

Teste de Mutação, o programa que está sendo testado é alterado diversas vezes, criando-se um conjunto de programas alternativos ou mutantes, como se defeitos estivessem sendo inseridos no programa original. O trabalho do testador é escolher casos de teste que mostrem a diferença de comportamento entre o programa original e os programas mutantes.

Assim como nas demais técnicas, cada mutante determina um subdomínio do domínio de entrada, formado por aqueles dados capazes de distinguir o comportamento do programa original e do mutante. A diferença, nesse caso, é que cada subdomínio está claramente relacionado com a capacidade de revelar um defeito específico. A partir da “hipótese do programador competente” e do “efeito de acoplamento”, discutidos adiante neste capítulo, pode-se argumentar que um conjunto de casos de teste capaz de distinguir um conjunto de mutantes escolhido de forma criteriosa, e que represente boa parte dos defeitos mais comuns, seria capaz de revelar, também, outros tipos de defeitos.

Este capítulo está organizado como se segue. Na Seção 5.2 é feita uma revisão histórica do surgimento e desenvolvimento do Teste de Mutação. Em seguida, na Seção 5.3 são apresentadas as definições e os conceitos básicos associados ao critério. Os passos envolvidos na aplicação do Teste de Mutação são descritos na Seção 5.4. Na Seção 5.5 são apresentadas e discutidas as principais características de algumas ferramentas de suporte ao Teste de Mutação, em particular da ferramenta Proteum [107].

Embora seja um critério voltado principalmente ao teste de unidade, é possível adaptar o Teste de Mutação para outras fases do processo de software. Em particular, é discutida na Seção 5.6 uma extensão do critério que procura modelar defeitos relacionados a erros de integração entre as unidades do programa. Como esses erros estão, em geral, relacionados com a interface de uma unidade, esse critério é chamado de Mutação de Interface. Na Seção 5.7 são brevemente discutidos outros trabalhos relacionados ao Teste de Mutação, em particular sua aplicação a modelos de especificação como Máquinas de Estados Finitos e Redes de Petri. Os comentários finais sobre o tema são apresentados na Seção 5.8.

## 5.2 Histórico

O artigo “Hints on test data selection. Help for the practicing programmer”, publicado em 1978 na revista *Computer*, por pesquisadores do Georgia Institute of Technology e da Yale University [115], é considerado o marco inicial do Teste de Mutação. Nesse trabalho, os autores – DeMillo, Lipton e Sayward – descrevem as idéias principais sobre a técnica, apresentando as duas hipóteses que fundamentam a sua utilização: a hipótese do programador competente e o efeito de acoplamento. A primeira, apesar de não ser explicitamente apresentada com essa denominação no artigo, aparece fortemente relacionada com a segunda.

Os autores também apresentam uma série de exemplos que ilustram como casos de teste que revelam defeitos simples podem melhorar a qualidade da atividade de teste. Apresentam, ainda, uma comparação da técnica de mutação com critérios de teste baseado em fluxo de controle e com o teste randômico, mostrando as vantagens da técnica por eles proposta. Já nesse primeiro artigo os autores comentam sobre o “sistema de mutação” desenvolvido pelo grupo de pesquisa em que trabalham e que daria suporte à aplicação do Teste de Mutação.

Nos anos seguintes, muitos trabalhos foram desenvolvidos, abordando diversos aspectos do Teste de Mutação, como, por exemplo, aspectos teóricos, criação de critérios alternativos, avaliação do critério, definição de arquiteturas para a execução eficiente dos mutantes

ou desenvolvimento de ferramentas. Alguns desses trabalhos serão comentados na próxima seção.

Muitos dos trabalhos desenvolvidos buscavam solucionar ou minimizar o problema do custo de aplicação do Teste de Mutação. Principalmente nos primeiros anos, quando não se dispunha de hardware potente como os que se têm hoje, a necessidade de executar um alto número de programas alternativos era um sério empecilho para sua utilização. Nesse contexto, uma linha de trabalho que se destacou foi a da chamada “mutação fraca” [190]. Esse critério considera que um programa **P** a ser testado é composto por uma série de “componentes” que, normalmente, correspondem a estruturas computacionais elementares como, por exemplo, referência a uma variável, expressão aritmética, expressão booleana ou relação entre expressões. Dado um componente **C** do programa **P**, uma transformação é aplicada em **C** produzindo o componente **C'**. O critério de mutação fraca requer que um caso de teste *t* seja produzido de modo que, em pelo menos uma vez durante a execução de **P**, o componente **C** produza um resultado diferente de **C'**.

Com essa abordagem evita-se executar o programa inteiro para decidir quando um mutante (componente) é morto. Por outro lado, é possível que um caso de teste que diferencie os componentes não chegue a distinguir os programas, se considerada sua execução completa. Em outras palavras, um caso de teste *t* pode causar diferenças nos resultados dos componentes mas não levar o programa mutante a uma falha. Assim, conjuntos que sejam adequados à mutação fraca podem não ser adequados ao Teste de Mutação tradicional. Daí o nome do critério.

Em 2000 realizou-se em San Jose, nos Estados Unidos, o “Mutation 2000: A Symposium on Mutation Testing for the New Century”. Nesse simpósio reuniram-se pesquisadores de diversos países envolvidos em atividades relacionadas ao teste de software e, particularmente, ao Teste de Mutação. O grupo responsável pela produção deste livro teve forte participação no evento, sendo o grupo que mais teve artigos aceitos para apresentação. Para se ter uma idéia, dos quinze artigos aceitos no Mutation 2000, quatro tiveram a participação dos autores deste livro.

Diversos aspectos relacionados ao Teste de Mutação foram abordados nos artigos do simpósio. As sessões técnicas foram divididas em quatro tópicos principais: aplicação, efetividade e geração de teste, redução de custo, mutação de interface e ferramentas. Três palestras foram proferidas pelos pesquisadores que tiveram maior influência na criação da teoria e da aplicação do Teste de Mutação, incluindo o “pai” da técnica, Prof. Richard DeMillo. A seguir, é traduzida parte do texto, de autoria de DeMillo, que abre os anais deste evento. Nele, o próprio autor descreve a origem da idéia do Teste de Mutação.

No final do outono de 1975 fiz uma das minhas freqüentes viagens de Atlanta a New Haven para colaborar com Richard Lipton. Trabalhávamos na época em alguns resultados teóricos em linguagens de programação. Havíamos, também, iniciado uma discussão com Alan Perlis sobre prova de correção de software. Essas discussões levariam eventualmente ao nosso artigo “Social Processes and Proofs of Theorems and Programs”. Assim, a correção de programas permeava minha mente. Eu recordo que era um tanto tarde quando entrei na sala em que Lipton estava reunido com Fred Sayward, Dave Hanson e alguns poucos estudantes de pós-graduação, incluindo Tim Budd. Eles discutiam um artigo que Lip-

ton escrevera anos antes quando era um estudante de pós-graduação na Carnegie Mellon. Eu estava completamente despreparado para a idéia que ouvi aquele dia.

Credita-se a Edsger Dijkstra a observação (em defesa da prova de programas) que o teste de software pode apenas evidenciar a presença de erros, mas não a sua ausência. Mas, argumentava Lipton, e se existisse apenas um número finito de possíveis erros? Então, o axioma de Dijkstra falha, pois um conjunto de casos de teste no qual o programa em teste tem sucesso mas cada um dos possíveis programas incorretos produz uma saída incorreta, indica que o programa não contém nenhum dos possíveis erros e, portanto, está correto.

Nos próximos três dias, a maior parte das idéias do que era chamado “teste de perturbação” foi trabalhada na sua forma bruta. Sayward e Budd encontraram alguma base empírica para modelos de erros simples no trabalho seminal de Gannon. Isso levou a um elegante esquema para a criação e execução de erros usando operadores de perturbação que modificavam as folhas de uma árvore de derivação. Existia, ainda, o problema do crescimento combinatorial do número de erros simples que poderiam conspirar para causar um defeito de software “real”, mas decidimos deixar esse problema de lado, por ora.

Ao final do ano acadêmico, Sayward e Budd haviam construído um interpretador para Fortran IV no PDP10 de Yale. Eu recrutei Alan Acree para fazer o mesmo para Cobol na Georgia Tech. Jerry Feldman na Universidade de Rochester sugeriu que teste de mutação seria um termo mais chamativo (e mais preciso) do que perturbação. Lipton e eu passamos o próximo verão no famoso Mathematic Research Center em Madison, Wisconsin. Muito do suporte teórico sobre mutação foi estabelecido naquele verão, incluindo metaindução e a hipótese do programador competente, verificação de equivalência e o efeito de acoplamento.

O efeito de acoplamento foi naquela época, e continua a ser, o aspecto mais controverso da teoria, mas para nós ele não era assim tão extraordinário. Gastamos algum tempo tentando inventar contra-exemplos mas, exceto pelos exemplos acadêmicos, hoje bem conhecidos, levavam sempre de volta ao acoplamento de erros simples. Eu achei uma coleção de histórias de ficção científica de Robert Silverberg na qual ele explica: “Mutações raramente são espetaculares. Aquelas mutantes que são extraordinariamente diferentes dos seus pais tendem a não viver muito, ou porque a mutação faz com que não sejam capazes de funcionar normalmente ou porque são rejeitados por aqueles que os precederam.” Eu fui sempre impressionado pela elegante simplicidade dessa passagem e pensei ser essa a única forma sensata de justificar o efeito de acoplamento.

Naquela manhã de outono em 1975 euachei que “perturbar” programas com o explícito propósito de negar a suposição de Dijkstra sobre o teste de software era uma das mais poderosas idéias que já tinha encontrado. Os muitos estudantes e colegas que, ao longo dos anos, adicionaram profundidade às idéias e validaram os conceitos básicos ajudaram a criar o que é, na minha visão distorcida, a mais científica e racional abordagem para assegurar que programas de computador funcionem como deveriam.

Atualmente, o Teste de Mutação tem tido grande aceitação na comunidade de teste de software, que o considera, apesar de relativamente caro, um dos critérios mais efetivos para

revelar a existência de defeitos. Por isso, diversos trabalhos na área de Engenharia de Software experimental utilizam o Teste de Mutação como um “padrão” para verificar a efetividade de novos critérios de teste.

## 5.3 Definições e conceitos básicos

Alguns resultados importantes foram obtidos na construção de uma teoria que fundamente a atividade de teste. Importantes, principalmente porque evidenciam o que pode e o que não pode ser feito com o teste.

Particularmente, sabe-se que não é possível, em geral, provar por meio de testes a correção de um programa. É preciso, então, mostrar até que ponto pode-se chegar com a aplicação de testes ou, mais especificamente, com a aplicação de determinada técnica ou determinado critério de teste [116]. Esta seção procura extraír de alguns trabalhos dados que fornecem uma visão da teoria que sustenta o Teste de Mutação. Primeiramente, é preciso uma definição de correção (*correctness*) de um programa, que pode ser:

Um programa **P** é correto com relação a uma função **F** se **P** computa **F**.

Para provar a correção de um programa, um conjunto de teste deve ser confiável (*reliable*), de acordo com a definição de Howden [187]:

Um conjunto de teste **T** é confiável para um programa **P** e uma função **F**, dado que **F** e **P** coincidem em **T**, se e somente se **P** computa **F**. Se **P** não computa **F**, então **T** deve conter um ponto *t* tal que  $F(t) \neq P(t)$ .

Se por um lado é simples mostrar que para qualquer programa existe sempre um conjunto de teste finito e confiável, por outro lado não existe um procedimento efetivo para gerar casos de teste confiáveis ou avaliar se um conjunto de teste é ou não confiável [116].

Com o intuito de obter uma definição mais realista, no sentido de que possa ser empregada na prática, outra abordagem para correção se apresenta. Nessa abordagem, considera-se um conjunto  $\Phi$  de programas alternativos, que depende de **P** e do qual **P** é membro. Tal conjunto  $\Phi(P)$  é chamado de “vizinhança” de **P**.

A intenção é mostrar que um programa é correto por meio de um processo de eliminação, mostrando que nenhum programa na vizinhança de **P**, não equivalente a **P**, é correto. Isso pode ser feito usando-se um conjunto de teste **T** para o qual todos os membros de  $\Phi(P)$ , não equivalentes a **P**, falhem em pelo menos um caso de teste. Tal conjunto é definido da seguinte maneira:

Um conjunto de teste **T** é adequado para **P** em relação a  $\Phi$  se para cada programa **Q** pertencente a  $\Phi$ , ou **Q** é equivalente a **P** ou **Q** difere de **P** em pelo menos um caso de teste.

Observe que a definição não aborda o problema da indecidibilidade sobre a equivalência entre programas. A relação entre correção de um programa **P** e adequação de um conjunto de teste é mostrada pelo seguinte teorema:

Dado um programa  $\mathbf{P}$  e uma vizinhança  $\Phi(P)$ , se algum programa em  $\Phi(P)$  computa a função correta  $\mathbf{F}$  e  $\mathbf{T}$  é adequado para  $\mathbf{P}$  em relação a  $\Phi$  e  $\mathbf{P}$  coincide com  $\mathbf{F}$  em  $\mathbf{T}$ , então  $\mathbf{P}$  é correto [50].

Tomando  $\Phi$  como o conjunto de todos os programas, garante-se que existe um programa em  $\Phi$  que computa  $\mathbf{F}$  e, assim, é possível provar a correção de  $\mathbf{P}$ . Contudo, a escolha de uma vizinhança como essa acarreta um problema prático, como aponta DeMillo [116]. O problema é que  $\Phi$  é infinito, sendo impossível demonstrar que  $\mathbf{T}$  é adequado em relação a ele.

Na verdade, a escolha de qualquer vizinhança de tamanho infinito não interessa, pois, para tais conjuntos, nem sempre existe um conjunto de teste  $\mathbf{T}$  que seja adequado [49]. Pior que isso, existem vizinhanças para as quais é indecidível se existe ou não um conjunto de teste adequado. Ou seja, tomando uma vizinhança  $\Phi$  infinita, não é possível decidir-se sobre a existência, para um programa  $\mathbf{P}$ , de um conjunto de teste  $\mathbf{T}$  que seja adequado em relação a  $\Phi$ .

Por outro lado, esse problema não existe para  $\Phi(P)$  finito. Nesse caso, é demonstrado que sempre existe um conjunto  $\mathbf{T}$  adequado para  $\mathbf{P}$  em relação a  $\Phi$ . É importante notar que, ao restringir a vizinhança, escolhendo-se  $\Phi$  como um subconjunto entre todos os possíveis programas, se abandona o desejo ideal de provar que o programa está correto e procura-se demonstrar uma correção relativa de  $\mathbf{P}$  sobre um pequeno conjunto de programas.

No Teste de Mutação, a chamada “hipótese do programador competente” afirma que um programa, produzido por um programador competente, está correto ou está “próximo” do correto. Dessa forma, as atenções podem concentrar-se em uma vizinhança bem menor que o universo de todos os programas, ou seja, naqueles programas que são parecidos com  $\mathbf{P}$ . De modo similar, a noção de restringir o conjunto de programas alternativos pode ser vista como a delimitação dos tipos de erros que se desejam considerar.

Para todo programa  $\mathbf{P}$  e vizinhança  $\Phi(P)$  finita, sabe-se que sempre existe pelo menos um conjunto de teste  $\mathbf{T}$  adequado para  $\mathbf{P}$  em relação a  $\Phi$ . Entretanto, duas perguntas precisam ser respondidas:

- Existe um procedimento para construir  $\mathbf{T}$ ?
- Existe um procedimento que, dado um conjunto de teste  $\mathbf{T}$ , avalie se  $\mathbf{T}$  é ou não adequado?

Budd [49] define dois procedimentos:

Um procedimento  $A(P, T)$  é um “aceitador” (*acceptor*) se ele retorna verdadeiro quando  $\mathbf{T}$  é adequado para  $\mathbf{P}$  em relação a  $\Phi(P)$  e retorna falso caso contrário.

Um procedimento  $G(P)$  é um “gerador” (*generator*) se ele produz um conjunto de teste  $\mathbf{T}$  que é adequado para  $\mathbf{P}$  em relação a  $\Phi(P)$ .

Budd mostra que sempre que existir um aceitador existirá também um gerador e vice-versa, mas isso nem sempre acontece. A existência desses dois procedimentos está relacionada com a existência de um “verificador de equivalência”  $S(P_1, P_2)$  que responde verdadeiro quando os programas  $P_1$  e  $P_2$  são equivalentes e falso caso contrário. Mais precisamente, mostra que existe um aceitador se e somente se existir um gerador, se e somente se existir um verificador de equivalência.

Para classes gerais de programas como o conjunto de todos os programas escritos em C ou Fortran ou na maioria das linguagens de programação, o problema da equivalência entre programas é indecidível [49]. Por outro lado, no Teste de Mutação, tal problema está restrito a decidir se o programa **P** é ou não equivalente aos demais elementos de  $\Phi(P)$ , os quais são bastante parecidos com **P**. Nesse caso, existem heurísticas, ou mais comumente, a intervenção do usuário testador, que permitem que a equivalência seja decidida. Assim, torna-se possível criar aceitadores e geradores de conjuntos de casos de teste baseados no Teste de Mutação.

## 5.4 Aplicação do Teste de Mutação

Esta seção descreve o critério de Teste de Mutação. Vale notar que o modo como o critério é aplicado na prática estabelece o algoritmo de um aceitador e de um gerador, definidos na seção anterior. Basicamente, dados um programa **P** e um conjunto de casos de teste **T** cuja qualidade se deseja avaliar, os seguintes passos para aplicar o critério são estabelecidos:

- geração dos mutantes;
- execução do programa em teste;
- execução dos mutantes; e
- análise dos mutantes.

É importante notar: tal procedimento exige, além da execução do programa que está sendo testado, que algumas tarefas sejam executadas de forma automática. Pela grande quantidade de processamento que demandam tarefas como gerar e executar os mutantes e comparar os resultados produzidos, é praticamente impossível executá-las com precisão e eficiência sem apoio computacional. Assim, na prática, deve-se associar à aplicação do critério uma ferramenta de suporte que interaja com o testador na execução do teste. Tal ferramenta, além de interativa, é também iterativa, pois envolve a repetição de três passos básicos: 1) criar casos de teste; 2) executar os mutantes; e 3) analisar o resultado da execução. Em seguida, serão analisados com mais detalhes os passos descritos anteriormente. O Programa 5.1 será utilizado como exemplo.

---

Programa 5.1

---

```
1 void main()
2 {
3     int x, y, pow;
4     float z, ans;
5
6     scanf ("%d %d", &x, &y);
7     if (y >= 0)
8         pow = y;
9     else
10        pow = -y;
11     z = 1;
12     while (pow-- > 0)
13         z = z*x;
14     if (y < 0)
15         z = 1 / z;
16     ans = z + 1;
17     printf ("%-.2f", ans);
18 }
```

---

### 5.4.1 Geração dos mutantes

Esta é a primeira fase da aplicação do critério. De acordo com a teoria apresentada anteriormente, constrói-se a vizinhança  $\Phi(P)$  efetuando-se pequenas alterações em  $\mathbf{P}$ . Cada programa gerado dessa forma é um mutante de  $\mathbf{P}$ .

Esse é o passo-chave da Análise de Mutantes. O sucesso da aplicação do critério depende totalmente da escolha da vizinhança  $\Phi(P)$ , que deve ter as seguintes características:

- ser abrangente, de modo que um conjunto de casos de teste adequado de acordo com a vizinhança gerada consiga revelar a maior parte dos erros de  $\mathbf{P}$ ; e
- ter cardinalidade pequena, para que o problema de gerar ou de verificar se um conjunto de casos de teste é ou não adequado seja tratável.

A hipótese do programador competente afirma que todo programa criado por um programador competente está correto ou está próximo do correto. Caso se assuma essa hipótese como válida, pode-se afirmar que erros são introduzidos nos programas por meio de desvios sintáticos que, apesar de não causarem erros sintáticos, alteram a semântica do programa e, como consequência, conduzem o programa a um comportamento incorreto. Para revelar tais erros, a análise de mutantes identifica os desvios sintáticos mais comuns e, aplicando pequenas transformações sobre o programa em teste, encoraja o testador a construir casos de teste que mostrem que tais transformações conduzem a um programa incorreto [7].

Além disso, alguns estudos como [2, 51] mostram, como conclusão de procedimentos experimentais, que conjuntos de casos de teste que revelam erros simples como os descritos acima conseguem, em geral, revelar outros tipos de erros. Esse é o chamado “efeito de acoplamento” (*coupling effect*) que, segundo DeMillo [115], pode ser descrito da seguinte maneira:

Dados de teste que distinguem todos os programas que diferem de um programa correto somente em erros simples são tão sensíveis que também distinguem, implicitamente, erros mais complexos.

Para modelar os desvios sintáticos mais comuns, gerando o conjunto de mutantes  $\Phi(P)$ , utilizam-se “operadores de mutação” (*mutant operators*). Um operador de mutação aplicado a um programa  $\mathbf{P}$  transforma  $\mathbf{P}$  em um programa similar, ou seja, um mutante. Em geral, a aplicação de um operador de mutação gera mais que um mutante, pois, se  $\mathbf{P}$  contém várias entidades que estão no domínio do operador, então esse operador é aplicado a cada uma dessas entidades, uma de cada vez. Como exemplo, considere o operador de mutação que retira um comando de  $\mathbf{P}$ . Todos os comandos do programa estão no domínio desse operador que, quando aplicado em  $\mathbf{P}$ , gera tantos mutantes quantos forem os seus comandos. Cada mutante contém todos os comandos de  $\mathbf{P}$  exceto um, retirado pelo operador de mutação.

Um mutante criado como o descrito é um “mutante de erro induzido” (*fault induced mutant*). Outro tipo de operador de mutação é o que gera “mutantes instrumentados” (*instrumented mutants*). Nesse caso, uma “função armadilha” (*trap function*) é incluída no programa  $\mathbf{P}$  para gerar o mutante. Esse tipo de operador não tem como objetivo modelar algum tipo especial de erro e sim garantir que os casos de teste tenham alguma característica particular como a cobertura de todos os comandos do programa.

Além dos tipos de erros que se desejam revelar e da cobertura que se quer garantir, a escolha de um conjunto de operadores de mutação depende também da linguagem em que os programas a serem testados estão escritos. Budd [49] relaciona 22 operadores de mutação utilizados pelo sistema de mutação EXPER (Experimental Mutation System) [49], para programas em Fortran. Para Cobol, são 27 operadores, utilizados pelo sistema CMS [2]. Já para a linguagem C, Agrawal et al. [7] definem um conjunto de 77 operadores de mutação. Esse conjunto de operadores foi adaptado e implementado na ferramenta Proteum [107], descrita na Seção 5.5.

Tradicionalmente, não existe uma maneira direta de definir os operadores de mutação para uma dada linguagem. Em geral, os operadores de mutação são projetados tendo por base a experiência no uso de dada linguagem, bem como os enganos mais comuns cometidos durante a sua utilização. Nos trabalhos de Kim [212, 214] foi proposto o uso de uma técnica chamada “Hazard and Operability Studies – (HAZOP)” para se chegar ao conjunto de 13 operadores de mutação de classe para programas Java. Embora a técnica HAZOP não difira substancialmente da abordagem usual, ela tenta sistematizar e tornar mais rigorosa a forma como os operadores de mutação são gerados. Primeiramente, a técnica identifica na gramática de Java as construções alvo de mutações e, com base em algumas palavras-chave, tais como NO/NONE, MORE, LESS, AS WELL AS, PART OF, REVERSE, OTHER THAN, NARRROWING, WIDENING e EQUIVALENT, os operadores de mutação são definidos [213]. A idéia é que essas palavras-chave guiem a construção dos operadores de mutação. À medida que mais palavras-chave são aplicadas nas diversas partes que compõem determinado comando, os operadores de mutação vão sendo definidos. Esse processo é repetido considerando cada um dos comandos aos quais se desejam aplicar mutações.

Como exemplo, considere os seguintes operadores de mutação para C [7], aplicados sobre o programa exemplo da Figura 5.1:

- **SSDL:** eliminação de comandos;
- **ORRN:** troca de operador relacional;
- **STRI:** armadilha em condição de comando *if*; e
- **Vsrr:** troca de variáveis escalares.

O primeiro é o operador “eliminação de comando” (SSDL – *statement deletion*). Esse operador é projetado para mostrar que cada comando do programa tem um efeito sobre a saída calculada, encorajando o testador a escolher casos de teste que mostrem tal propriedade. A Tabela 5.1 mostra quais são os comandos que devem ser retirados do programa P para criar os mutantes.

O operador “troca de operador relacional” (ORRN – *relational operator replacement*) atua sobre cada expressão do programa que contenha algum operador relacional, trocando-o por todos os outros. A Tabela 5.2 mostra como ficam os predicados dos mutantes que contêm operadores relacionais.

O operador “armadilha em condição IF” (STRI – *trap on IF condition*) tem o intuito de auxiliar na análise da cobertura de desvios. Aplicado a P, STRI gera dois mutantes instrumentados para cada comando if. Por exemplo, dado o comando: `if (e) S`, dois mutantes são gerados:

Tabela 5.1 – Mutantes gerados pelo operador SSDL

1	scanf ("%d %d", &x, &y)
2	if (y >= 0) pow = y; else pow = -y;
3	pow = y
4	pow = -y
5	z = 1
6	while (pow-- > 0) z = z * x;
7	z = z * x
8	if (y < 0) z = 1 / z
9	z = 1 / z
10	ans = z + 1
11	printf ("%-.2f", ans)

Tabela 5.2 – Mutantes gerados pelo operador ORRN

12	if (y > 0)
13	if (y < 0)
14	if (y == 0)
15	if (y <= 0)
16	if (y != 0)
17	while (pow-- < 0)
18	while (pow-- == 0)
19	while (pow-- >= 0)
20	while (pow-- <= 0)
21	while (pow-- != 0)
22	if (y > 0)
23	if (y == 0)
24	if (y >= 0)
25	if (y <= 0)
26	if (y != 0)

- if (armadilha\_se\_verdadeiro(e)) S;
- if (armadilha\_se\_falso(e)) S;

Quando a função armadilha\_se\_verdadeiro (armadilha\_se\_falso) é executada, o mutante é abortado e morto se seu argumento é verdadeiro (falso). Caso contrário, a função retorna simplesmente o valor do argumento e a execução do mutante continua normalmente. Com este operador de mutação, o testador se vê forçado a escolher casos de teste de tal forma que

todos os desvios especificados por comandos if sejam exercitados pelo menos uma vez. A Tabela 5.3 mostra como ficam os mutantes instrumentados com essas funções.

Tabela 5.3 – Mutantes gerados pelo operador STRI

27	<code>if (armadilha_se_verdadeiro(y &gt;= 0))</code>
28	<code>if (armadilha_se_falso(y &gt;= 0))</code>
29	<code>if (armadilha_se_verdadeiro(y &lt; 0))</code>
30	<code>if (armadilha_se_falso(y &lt; 0))</code>

Com o operador “troca de referência a variável escalar” (*Vsrr – scalar variable reference replacement*), cada referência a uma variável não estruturada (escalar) é substituída pela referência a todas as outras variáveis escalares do programa, uma de cada vez. O erro modelado por esse operador é o uso de uma variável incorreta. A Tabela 5.4 mostra como trechos do programa devem ser alterados para criar os mutantes. Note-se o grande número de mutantes criados por esse operador. Como observa Budd [49], esse tipo de operador é o que, em geral, produz o maior número de mutantes.

É importante notar que, em princípio, é possível gerar mutantes com a aplicação de mais de um operador de mutação de uma só vez. Mutantes gerados a partir de k alterações simultâneas no programa **P** que está sendo testado são chamados de mutantes de ordem k. Estudos anteriores [51] mostram que mutantes de ordens superiores, além de não contribuir de forma significativa para a construção de casos de teste melhores, têm um custo de geração e execução demasiado alto. Portanto, tem-se utilizado no Teste de Mutação uma vizinhança composta apenas dos mutantes de primeira ordem.

#### 5.4.2 Execução do programa em teste

Este passo no Teste de Mutação não é diferente de outros critérios de teste. Consiste em executar o programa **P** usando os casos de teste selecionados e verificar se seu comportamento é o esperado. Se não for, ou seja, se o programa apresenta resultados incorretos para algum caso de teste, então um erro foi detectado. Caso contrário, o teste continua no passo seguinte.

Em geral, a tarefa de oráculo, ou seja, decidir se o resultado produzido é correto ou não, é desempenhada pelo testador. Isso não acontece apenas na Análise de Mutantes, mas na maioria das técnicas e dos critérios de teste.

Suponha que, inicialmente, dispõe-se do conjunto  $T = \{(2, 1)\}$  para o teste do programa exemplo, ou seja, **T** possui apenas um caso de teste com os valores  $X = 2$  e  $Y = 1$ . De acordo com a especificação, pode-se calcular o resultado esperado, que é  $2^1 + 1 = 3$ .

Executando o programa **P** do exemplo com esse caso de teste obtém-se a impressão do valor “3.00”, ou seja, o programa comportou-se como o esperado.

#### 5.4.3 Execução dos mutantes

Neste passo, cada um dos mutantes é executado usando os casos de teste de **T**. Se um mutante **M** apresenta resultado diferente de **P**, isso significa que os casos de teste utilizados são

Tabela 5.4 – Mutantes gerados pelo operador Vsrr

31	scanf ("%d %d", &y, &y)	32	scanf ("%d %d", &pow, &y)
33	scanf ("%d %d", &z, &y)	34	scanf ("%d %d", &ans, &y)
35	scanf ("%d %d", &x, &x)	36	scanf ("%d %d", &x, &pow;
37	scanf ("%d %d", &x, &z)	38	scanf ("%d %d", &x, &ans)
39	if (x >= 0)	40	if (pow >= 0)
41	if (z >= 0)	42	if (ans >= 0)
43	pow = x	44	pow = pow
45	pow = z	46	pow = ans
47	x = y	48	y = y
49	z = y	50	ans = y
51	pow = -x	52	pow = -pow
53	pow = -z	54	pow = -ans
55	x = -y	56	y = -y
57	z = -y	58	ans = -y
59	x = 1	60	y = 1
61	pow = 1	62	ans = 1
63	while (x-- > 0)	64	while (y-- > 0)
65	while (z-- > 0)	66	while (ans-- > 0)
67	z = z * x	68	z = z * pow
69	z = z * z	70	z = z * ans
71	z = x * x	72	z = y * x
73	z = pow * x	74	z = ans * x
75	x = z * x	76	y = z * x
77	pow = z * x	78	ans = z * x
79	if (x < 0)	80	if (pow < 0)
81	if (z < 0)	82	if (ans < 0)
83	z = 1 / x	84	z = 1 / y
85	z = 1 / pow	86	z = 1 / ans
87	x = 1 / z	88	y = 1 / z
89	pow = 1 / z	90	ans = 1 / z
91	ans = x + 1	92	ans = y + 1
93	ans = pow + 1	94	ans = ans + 1
95	x = z + 1	96	y = z + 1
97	pow = z + 1	98	z = z + 1
99	printf ("%-.2f", x)	100	printf ("%-.2f", y)
101	printf ("%-.2f", pow)	102	printf ("%-.2f", z)

sensíveis e conseguiram expor a diferença entre **P** e **M**; nesse caso, **M** está morto e é descartado. Por outro lado, se **M** apresenta comportamento igual a **P**, diz-se que **M** continua vivo. Isso indica ou uma fraqueza em **T**, pois não conseguiu distinguir **P** de **M**, ou que **P** e **M** são equivalentes.

Este passo pode ser totalmente automatizado, não requerendo a intervenção do testador. Os resultados obtidos com a execução dos mutantes são comparados com os produzidos pelo

programa original no passo anterior, e a própria ferramenta de mutação pode marcar como mortos os mutantes que apresentam resultados diferentes.

Após a execução dos mutantes, pode-se ter uma idéia da adequação dos casos de teste utilizados, por meio do “escore de mutação” (*mutation score*). O escore de mutação varia entre 0 e 1 e fornece uma medida objetiva de quanto o conjunto de casos de teste analisado aproxima-se da adequação. Dado o programa  $P$  e o conjunto de casos de teste  $T$ , calcula-se o escore de mutação  $ms(P, T)$  da seguinte maneira:

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

onde:

**DM( $P, T$ )**: número de mutantes mortos pelo conjunto de casos de teste  $T$ ;

**M( $P$ )**: número total de mutantes gerados a partir do programa  $P$ ; e

**EM( $P$ )**: número de mutantes gerados que são equivalentes a  $P$ .

Logo, o escore de mutação pode ser obtido calculando a razão entre o número de mutantes efetivamente mortos por  $T$  e o número de mutantes que se pode matar, dado pelo número total de mutantes gerados subtraído do número de mutantes equivalentes. Observe que apenas  $DM(P, T)$  depende do conjunto de casos de teste utilizado. Apesar disso, não se conhece, a princípio, o número de mutantes equivalentes gerados.  $EM(P)$  é obtido iterativamente à medida que o testador decide, no passo seguinte do teste, marcar como equivalente um mutante  $M$  que continua vivo.

Existem alguns problemas relacionados com esse passo da Análise de Mutantes. Um deles é decidir se um mutante está morto ou vivo; mais precisamente, determinar que entidades devem ser comparadas para decidir se o comportamento do mutante e o do programa original são os mesmos.

A opção mais simples é capturar as seqüências de caracteres enviadas pelos programas para os dispositivos de saída e comparar as seqüências produzidas pelo programa original e pelos mutantes. Alternativamente, podem-se comparar somente os caracteres não brancos [443]. Outros pontos podem ser usados para tentar distinguir o comportamento dos programas. O código de retorno, valor que indica se a execução terminou normalmente ou qual foi a causa da terminação, é também usado com esse propósito.

É importante observar que essas características excluem outros tipos de eventos de entrada e saída, como, por exemplo, em programas com interface gráfica. Nesses casos torna-se mais difícil capturar os casos de teste, reexecutar os mutantes e comparar os seus resultados utilizando tais eventos. Por isso, tende-se a utilizar o Teste de Mutação para o teste de unidade ou integração, e não em sistemas completos.

Se a Análise de Mutantes estiver sendo aplicada em um ambiente interpretado, no qual a ferramenta de teste tem controle sobre a execução dos mutantes e do programa em teste, então pode-se usar também o valor final das variáveis ou até mesmo estados intermediários do programa para decidir se um mutante deve morrer ou não [309].

Se o tempo de execução de um mutante ultrapassa muito o tempo de execução do programa original, isso pode indicar que a alteração fez com que o mutante entrasse em um laço infinito. Tal mutante pode, também, ser considerado morto por “timeout”.

Assim, decidir quais saídas devem ser monitoradas é mais uma questão de implementação da ferramenta de suporte do que uma característica intrínseca do Teste de Mutação. Não são bem claras as consequências de se escolher como parâmetro de decisão essa ou aquela entidade.

O principal problema associado à Análise de Mutantes diz respeito ao número de mutantes gerados. Muitas vezes, o número de mutantes criados é muito alto, fazendo com que a complexidade computacional da execução de todos eles e o esforço para identificação dos mutantes equivalentes sejam altos. Tal problema pode ser facilmente observado a partir do exemplo apresentado neste capítulo: para um programa com apenas 15 linhas foram gerados, utilizando apenas quatro operadores de mutação, mais de 100 mutantes.

Uma observação importante deve ser feita: em geral, independentemente da qualidade dos casos de teste utilizados, 80% dos mutantes gerados morrem na primeira execução, diminuindo sensivelmente o número necessário de execuções de programas. Tal fato não significa, porém, que a geração e a execução desses 80% seja um trabalho inútil, tendo em vista que sua eliminação não contribui para melhorar a qualidade do conjunto de casos de teste. De fato, determinar *a priori* quais são os mutantes que morrem rápido é, pelo menos, tão trabalhoso quanto matá-los usando os casos de teste [51].

Várias estratégias têm sido utilizadas para fazer com que a Análise de Mutantes possa ser utilizada de maneira mais eficiente. Essas estratégias podem ser divididas basicamente em dois grupos: as que atacam o problema tentando diminuir o tempo de execução dos mutantes e as que procuram diminuir o número de mutantes gerados.

O primeiro grupo de estratégias tem mais caráter teórico e acadêmico e não muito prático. Isso porque, em geral, essas estratégias sugerem a utilização de arquiteturas de hardware de alto desempenho para a execução eficiente dos mutantes. Como resultado, obtém-se uma diminuição no tempo de execução dos mutantes, mas um aumento significativo no custo do ambiente de teste, que requer hardware específico e caro.

Uma das abordagens para diminuir o tempo de execução dos mutantes é apresentada por Mathur [273], que propõe um algoritmo para otimizar a execução de mutantes gerados pelo operador Vsrr. Cada mutante desse tipo contém como diferença do programa original **P** uma única referência a alguma posição de memória, não existindo nenhuma diferença na estrutura dos comandos do mutante em relação a **P**. Para cada comando em cada mutante, as sequências de instruções a serem executadas são as mesmas. Somente as posições de memória de onde os operandos são retirados e onde os resultados são armazenados é que mudam. Assim, em vez de executar seqüencialmente cada mutante, sugere-se que as instruções dos mutantes sejam executadas numa arquitetura vetorializada.

Outro trabalho nessa área utiliza o modelo de máquinas SIMD para executar os mutantes de maneira concorrente [226]. Basicamente, uma máquina SIMD possui vários processadores simples com memória local, chamados de elementos processadores (EP), controlados por uma unidade de controle (UC) que cuida de obter as instruções a serem executadas e distribuí-las para os EPs, os quais executam a mesma instrução simultaneamente sobre os operandos que estão nas memórias locais. A idéia é agrupar os mutantes gerados por um

mesmo operador de mutação sobre o mesmo ponto do programa e executar as partes comuns desses mutantes de forma concorrente, um mutante em cada EP.

Choi et al. [77] descreve uma implementação de um sistema de mutação que utiliza uma arquitetura hipercubo para distribuir a execução dos mutantes. Basicamente, tem-se o sistema de mutação rodando numa máquina hospedeira e comunicando-se com um pool de processadores com memória local arranjados na forma de um hipercubo. Cada processador do hipercubo é alocado para executar e avaliar os resultados de um mutante de cada vez. No hospedeiro existe um escalonador que cuida da comunicação com os processadores do hipercubo fornecendo dados, ou seja, o mutante a ser executado, os casos de teste a serem empregados e os resultados esperados, e recebendo de cada nó uma resposta se o mutante foi morto ou não.

Para diminuir o tempo de execução dos mutantes existe também a solução por meio de software. Como cada mutante difere do programa original em apenas um ponto e como a execução de ambos até esse ponto é igual, então não é necessário executar o trecho que ambos têm em comum duas vezes. A idéia é, por meio da execução interpretada dos programas, fazer com que cada mutante inicie sua execução somente a partir do ponto no qual ele difere do programa original [309].

Uma abordagem completamente diferente das anteriores é tentar selecionar, entre os mutantes gerados, apenas um subconjunto deles. O objetivo é diminuir o número de mutantes a serem executados e analisados, porém sem perder a qualidade.

A idéia por trás dessas abordagens é sempre a mesma. Tomando um programa **P** e sua vizinhança  $\Phi(P)$ , gerada por um conjunto de operadores de mutação, estudos experimentais mostram ser possível selecionar um subconjunto  $\Phi'(P)$  de forma que, ao construir um conjunto de casos de teste **T** adequado de acordo com  $\Phi'(P)$ , tem-se que **T** é também adequado ou está muito próximo da adequação em relação a  $\Phi(P)$ . Em outras palavras, em vez de utilizar o conjunto total de mutantes para a aplicação do critério, pode-se utilizar um subconjunto desses mutantes, obtendo um conjunto de casos de teste de qualidade similar.

A questão que se coloca, então, é como selecionar esse subconjunto de mutantes, de modo a maximizar a adequação em relação ao conjunto total de mutantes. A primeira forma, e também a mais simples, utilizada para diminuir o número de mutantes a serem executados é a amostragem simples. Em vez de considerar todos os mutantes gerados, seleciona-se aleatoriamente uma porcentagem deles e a análise fica restrita apenas a esses mutantes selecionados. Em geral, mesmo uma amostragem com pequena porcentagem do total de mutantes é suficiente para construir bons casos de teste. Estudos conduzidos com programas Fortran [2] mostraram que conjuntos de teste gerados usando a mutação aleatória com apenas 10% dos mutantes chegaram a alcançar escore de mutação de 0,99 em relação ao total de mutantes. Tais resultados foram confirmados por Mathur e Wong para programas Fortran e C [274, 437, 439].

A mutação aleatória não leva em consideração características particulares de cada operador de mutação no que diz respeito à eficácia em revelar a presença de defeitos. Assim, foi proposta a mutação restrita, na qual são selecionados apenas alguns dos operadores de mutação para serem empregados no teste. Os experimentos de Mathur e Wong [274, 437, 439] também exploram a mutação restrita, a qual se mostrou efetiva na redução do número de mutantes e com alta adequação em relação ao total de mutantes.

Inicialmente, os trabalhos de mutação restrita não identificaram uma maneira sistemática para definir os operadores de mutação a serem utilizados. Assim, outros trabalhos buscaram identificar quais seriam os subconjuntos de operadores mais indicados, de forma a maximizar a adequação dos conjuntos de teste gerados. Offutt et al. [313] propõem o uso da mutação seletiva, na qual a seleção dos operadores a serem utilizados está relacionada com o número de mutantes gerados pelo operador. Os operadores que geram os maiores números de mutantes não são utilizados no teste. Na verdade, pode-se criar assim uma hierarquia de critérios: o critério n-seletivo é o que deixa de utilizar os mutantes dos  $n$  operadores que mais geram mutantes.

Nesse contexto, destacam-se ainda os trabalhos de Offutt et al. [310], Wong et al. [438] e Barbosa [24], que procuram determinar um subconjunto essencial de operadores de mutação que minimize o número de mutantes e maximize sua adequação em relação ao conjunto total de operadores. Em particular, esse último apresenta um procedimento, constituído de seis passos, que leva à seleção de um conjunto de operadores de mutação. Esse procedimento foi aplicado em um conjunto de programas C, utilizando os operadores de mutação da ferramenta Proteum, que resultou em um conjunto de operadores essenciais para aquela linguagem composto de nove operadores, com alta adequação em relação ao conjunto total de mutantes e, também, grande capacidade de redução no número de mutantes.

Por fim, pode-se utilizar uma estratégia incremental para a aplicação dos operadores de mutação. Ao utilizar qualquer técnica que se deseje como mutação aleatória, seletiva ou restrita, podem ser estabelecidas prioridades aos mutantes ou aos operadores de mutação. Inicialmente, são gerados mutantes com maior prioridade e são determinados casos de teste adequados a esse conjunto de mutantes. Em seguida, são gerados os mutantes com segundo grau de prioridade, e o conjunto inicialmente adequado é utilizado. Se necessário, novos casos de teste são adicionados ao conjunto para matar esses novos mutantes. Esse processo se repete enquanto os recursos de teste forem disponíveis ou até que se esgotem os operadores de mutação. Como coloca Budd [49], embora todos os mutantes desempenhem papel importante na tarefa de localizar erros, alguns tipos de mutantes são responsáveis pela detecção de um maior número de erros que outros. Assim, é interessante que tais mutantes sejam executados e mortos o mais cedo possível, pois, além de auxiliar na construção de casos de teste mais poderosos, fazem com que o trabalho de reteste, quando um erro é encontrado e corrigido, seja menor.

A técnica proposta por Marshall et al. [268] chamada SDAWM (*Static Dataflow-Aided Weak Mutation Analysis*) usa a análise estática para identificar mutantes que introduzem “anomalias de fluxo de dados” como a utilização de uma variável sem que tenha sido inicializada. Esses mutantes são eliminados sem a necessidade de execução por serem considerados versões incorretas *a priori*. Essa argumentação faz muito sentido, porém a técnica não ganhou popularidade e não tem sido utilizada na prática. Provavelmente, esses mutantes são facilmente mortos por qualquer conjunto de casos de teste e, portanto, o custo computacional para matar o mutante pode ser inferior ao custo da aplicação do SDAWM.

Voltando ao exemplo do Programa 5.1, antes de executarem-se os mutantes, aplica-se a estratégia SDAWM para diminuir o número de mutantes. Fazendo isso, são eliminados os mutantes 1, 2, 5, 6, 7, 10 e 11, gerados pelo operador SSDL. O operador Vsrr, como já foi mencionado, é o que gera maior número de mutantes. Por outro lado, é também o que cria mais mutantes com anomalias de fluxo de dados. Só não são eliminados os mutantes 39, 43, 48, 51, 56, 60, 71, 72, 73, 76, 77, 79, 80, 81, 83, 84, 85, 91, 92 e 93. Assim, do total de

83 mutantes gerados por operadores que podem introduzir anomalias, sobram 28. Ou seja, obtém-se uma redução de 66% desses mutantes e de 58% do total de mutantes.

O Programa 5.2 mostra o mutante 36, que é um dos mutantes eliminados por possuir anomalias de fluxo de dados. Claramente pode-se constatar uma anomalia no predicado do primeiro comando `if`. A variável `y` é utilizada no predicado sem ter sido inicializada. Isso acontece porque o comando que fazia a leitura do valor de `y` foi alterado, substituindo a leitura de `y` pela leitura de `pow`.

---

Programa 5.2

---

```

1 void main()
2 {
3     int x, y, pow;
4     float z, ans;
5
6     scanf ("%d %d", &x, &pow);
7     if (y >= 0)
8         pow = y;
9     else
10        pow = -y;
11     z = 1;
12     while (pow-- > 0)
13         z = z*x;
14     if (y < 0)
15         z = 1 / z;
16     ans = z + 1;
17     printf ("%-.5.2f", ans);
18 }
```

---

\*

---

Uma vez eliminados os mutantes com anomalias de fluxo de dados, resta executar os demais com os casos de teste de **T**, que até agora possui um único caso de teste (2, 1), e comparar os resultados com a execução de **P**. O resultado da execução está sumariado na Tabela 5.5.

Note-se que, dos mutantes mortos, alguns apresentaram resultado diferente de **P**, alguns tiveram erros de execução e outros morreram por “timeout”, ou seja, o tempo de execução do mutante superou muito o tempo de execução de **P**, indicando que possivelmente o mutante entrou num laço sem fim.

Dos 43 mutantes usados, 21 foram mortos pelo caso de teste. Considerando-se que até agora não se conhece nenhum mutante equivalente, pode-se calcular o escore de mutação como:

$$ms(P, T) = \frac{21}{43} = 0,49$$

#### 5.4.4 Análise dos mutantes vivos

Este é o passo que requer mais intervenção humana. Primeiro, é preciso decidir se o teste continua ou não. Se o escore de mutação for 1,0 – ou se estiver bem próximo de 1,0, de acordo com a avaliação do testador –, então, pode-se encerrar o teste e considerar **T** um bom conjunto de casos de teste para **P**.

Tabela 5.5 – Execução dos mutantes com  $T = \{(2, 1)\}$ 

Mutante	Resultado	Estado	Mutante	Resultado	Estado
3	erro	morto	4	3.00	<b>vivo</b>
8	3.00	<b>vivo</b>	9	3.00	<b>vivo</b>
12	3.00	<b>vivo</b>	13	2.00	morto
14	2.00	morto	15	2.00	morto
16	3.00	<b>vivo</b>	17	2.00	morto
18	2.00	morto	19	5.00	morto
20	2.00	morto	21	3.00	<b>vivo</b>
22	1.50	morto	23	3.00	<b>vivo</b>
24	1.50	morto	25	3.00	<b>vivo</b>
26	1.50	morto	27	armadilha	morto
28	3.00	<b>vivo</b>	29	3.00	<b>vivo</b>
30	armadilha	morto	39	3.00	<b>vivo</b>
43	5.00	morto	48	2.00	morto
51	3.00	<b>vivo</b>	56	3.00	<b>vivo</b>
60	1.00	morto	71	5.00	morto
72	3.00	<b>vivo</b>	73	3.00	<b>vivo</b>
76	2.00	morto	77	timeout	morto
79	3.00	<b>vivo</b>	80	3.00	<b>vivo</b>
81	3.00	<b>vivo</b>	83	3.00	<b>vivo</b>
84	3.00	<b>vivo</b>	85	3.00	<b>vivo</b>
91	3.00	<b>vivo</b>	92	2.00	morto
93	1.00	morto			

Caso se decida continuar o teste, é preciso analisar os mutantes que sobreviveram à execução com os casos de teste disponíveis e decidir se esses mutantes são ou não equivalentes ao programa original. Em geral, é indecidível se dois programas são equivalentes. Essa limitação teórica não significa que o problema deva ser abandonado por não ter solução. Na verdade, alguns métodos e algumas heurísticas têm sido propostos para determinar a equivalência de programas em uma grande porcentagem dos casos de interesse [306, 311]. E, em último caso, o testador deve decidir sobre a equivalência.

Nessa direção, podem ser destacados alguns trabalhos, como o de Offut e Craft [306], que visa à identificação de alternativas que permitam a determinação automática de mutantes equivalentes. Nesse estudo são apresentadas seis técnicas baseadas em estratégias de otimização de compiladores e análise de fluxo de dados para determinar mutantes equivalentes. As técnicas são:

- detecção de código não alcançável;
- propagação de constantes;
- propagação de invariantes;

- detecção de subexpressões comuns;
- detecção de invariantes de repetição; e
- *hoisting e sinking*.

A aplicação de tais técnicas mostra que, dependendo do tipo de programa que está sendo analisado, podem ser obtidos resultados bastante diferentes. Nos experimentos realizados por Offutt e Craft, o número de mutantes equivalentes corretamente identificados varia entre 0 e 50%. Complementando esse trabalho, Offutt e Pan [311] propuseram três estratégias baseadas em restrições matemáticas, originalmente desenvolvidas para a geração de dados de teste, para determinar mutantes equivalentes e caminhos não alcançáveis. Essas estratégias também chegam a detectar cerca de metade dos mutantes equivalentes, dependendo das características dos programas analisados.

No trabalho de Jorge [203] foram realizados estudos sobre os operadores de mutação para a linguagem C considerando os aspectos de equivalência. A partir dessa análise, foram determinadas heurísticas para a determinação de mutantes equivalentes, tanto pela semântica do operador como para o domínio de aplicação. Também foi feita uma análise estatística sobre os operadores de mutação, com o objetivo de fornecer diretrizes ao testador para auxiliá-lo na atividade de análise dos mutantes vivos. Observou-se que, apesar de não ser uma regra geral, seis operadores de mutação referentes ao teste de unidade e sete referentes ao teste de integração estão entre os dez operadores com custo mais elevado tanto em termos do número de mutantes gerados quanto do número de mutantes equivalentes. Assim, ao selecionar os operadores de mutação, deve-se evitar a utilização de tais operadores.

Uma vez decidido que um mutante não é equivalente ao programa original, deve-se, caso se deseje construir um conjunto de casos de teste adequado para **P**, elaborar um caso de teste que mate tal mutante. Em [116] é apresentada uma solução para tentar criar automaticamente casos de teste que matem os mutantes sobreviventes. O método, chamado de “geração automática baseada em restrições” (*constraint-based automatic test data generation*) associa a cada operador de mutação um modelo que fornece, para cada mutante gerado por esse operador, uma restrição que deve ser satisfeita na construção dos casos de teste.

Ao analisar os mutantes do Programa 5.1, são identificados alguns mutantes que são equivalentes e outros que não são. Os equivalentes são os de número 12, 20 e 25. A título de ilustração, considere o mutante 12. O programa original difere desse mutante na linha 7: o comando `if (y >= 0)` (programa original) foi substituído por `if (y > 0)` (mutante 12) como consequência da aplicação do operador ORRN, que realiza a troca de operadores relacionais. Essa alteração, no entanto, não provoca diferença de comportamento entre o programa original e o mutante. De fato, em ambos os programas, quando  $y = 0$ , `pow` é inicializada com 0.

Com a identificação dos mutantes equivalentes obtém-se um novo valor para o escore de mutação:

$$ms(P, T) = \frac{21}{43 - 3} = 0,53$$

Para os demais mutantes podem ser incluídos novos casos de teste com o intuito de matá-los. Por exemplo, analisando o mutante número 39, percebe-se que para tentar matá-lo deve ser utilizado um caso de teste com os valores de  $x$  e  $y$  com os sinais opostos; por exemplo,  $(2, -1)$ .

Assim, o caminho é retornar ao segundo passo do critério, executar **P** com esse novo caso de teste, executar os mutantes, recalcular o escore de mutação e assim por diante, até que se consiga um bom conjunto **T**.

O Programa 5.1, executado com o novo caso de teste  $(2, -1)$ , apresenta o resultado esperado, ou seja, “1.50”. A Tabela 5.6 mostra o resultado da execução dos mutantes, obtendo-se um escore de mutação de 0,95, bem próximo de 1.

Tabela 5.6 – Execução dos mutantes com  $T = \{(2, 1), (2, -1)\}$

Mutante	Resultado	Estado
4	erro	morto
8	3.00	morto
9	3.00	morto
16	2.00	morto
23	3.00	morto
28	armadilha	morto
29	armadilha	morto
39	2.00	morto
51	2.00	morto
56	2.00	morto
72	0.50	morto
73	1.50	<b>vivo</b>
79	3.00	morto
80	3.00	morto
81	3.00	morto
83	1.50	<b>vivo</b>
84	0.00	morto
85	erro	morto
91	3.00	morto

## 5.5 Ferramentas

Uma linha importante no desenvolvimento do critério Análise de Mutantes refere-se à construção de ferramentas de suporte a sua aplicação. Diversas ferramentas foram construídas nessa perspectiva, principalmente nos primeiros anos de utilização da técnica.

De acordo com a literatura, a primeira dessas ferramentas foi o FMS.1 (Fortran Mutation System – versão 1), desenvolvida na Yale University, em um computador PDP-10, e mais tarde em um PRIME 400 no Georgia Tech e em um VAX 11 em Berkeley. Essa ferramenta tratava de um conjunto restrito da linguagem Fortran, ou seja, programas com apenas uma rotina, com aritmética de inteiros apenas e sem comandos de entrada e saída. Esse projeto, apesar de suas restrições, motivou a construção de sistemas mais elaborados como o FSM.2 e o CMS.1 (Cobol Mutation System) [2]. Nessa mesma linha de ferramentas devem ser destacados o EXPER (Experimental Mutation System) [49] e a Mothra [76, 114], também

para Fortran. Essa última foi, certamente, a que mais se popularizou e mais influenciou o desenvolvimento do critério.

Na década de 1990, a família de ferramentas Proteum [107] iniciou um novo período na utilização do Teste de Mutação, apresentando diversas abordagens novas e permitindo seu uso de maneira mais abrangente pela comunidade científica. Essa família de ferramentas, inicialmente projetada para dar suporte ao Teste de Mutação em programas C [109], mais tarde foi estendida para outros modelos, tais como Máquinas de Estados Finitos, Redes de Petri e Statecharts [134, 131, 135].

Nas próximas seções as ferramentas Mothra e Proteum serão discutidas, dando-se destaque à última, apresentada em detalhes.

### 5.5.1 Mothra

A ferramenta Mothra foi desenvolvida por pesquisadores da Purdue University e do Georgia Institute of Technology. É um ambiente de teste para programas em Fortran 77 e, segundo seus criadores, o que a diferencia de outras ferramentas construídas anteriormente são os seguintes pontos:

- interface baseada em janelas, já que ferramentas anteriores eram baseadas em interface simplesmente textual;
- nenhuma restrição, *a priori*, quanto ao tamanho do programa a ser testado; e
- fácil incorporação de ferramentas como gerador de casos de teste, verificador de equivalência e oráculo.

O processo de testar um programa na ferramenta Mothra é chamado de “experimento”. É possível selecionar o conjunto de operadores de mutação a serem aplicados e um valor, chamado de “intensidade” do teste (*test strength*), que nada mais é senão a porcentagem dos mutantes que deve ser, aleatoriamente, selecionada para utilização no experimento. Selecionados esses parâmetros, a ferramenta cria um conjunto de descritores que representam as mutações a serem aplicadas. A criação de mutantes, ou habilitação dos operadores de mutação, de acordo com o jargão dos criadores da ferramenta, pode ser feita gradativamente à medida que o experimento se desenvolve.

Outra tarefa a ser executada durante um experimento é fornecer os dados de teste a serem avaliados. Um dado de teste é composto por dois tipos de valores: os valores iniciais, que são os argumentos fornecidos ao programa, e os valores em tempo de execução, que são lidos pelo programa durante a sua execução. O testador deve fornecer os valores iniciais, e, então, a ferramenta inicia a execução do programa em teste. Quando o programa executa um comando de leitura, o testador deve fornecer os valores em tempo de execução. Terminada a execução, caso nenhum erro seja revelado, os dados de entrada e os valores de saída produzidos são armazenados, caracterizando um caso de teste que será utilizado na execução dos mutantes.

A execução dos mutantes e a comparação com os resultados originais é feita de forma totalmente automática, sem necessidade de interação com o testador. Durante esse processo, a ferramenta cria, baseada nos descritores dos mutantes, uma representação interna do programa mutante, executa-o de maneira interpretada e compara os resultados produzidos com

os resultados do programa original. Se os resultados diferem, para algum caso de teste, a ferramenta não executa os mutantes para os demais casos de teste, pois o mutante já pode ser considerado morto.

Uma das características importantes adicionadas à Mothra foi o gerador automático de casos de testes, denominado Godzilla [116]. Associando a cada mutante uma restrição, o gerador procura criar casos de teste que satisfaçam essas restrições e, com isso, levem à distinção dos mutantes correspondentes.

Embora ainda seja possível encontrar uma versão da ferramenta para download no endereço <http://ise.gmu.edu/faculty/ofut/rsrch/mut.html>, sua instalação completa requer uma série de pacotes de software antigos e que hoje raramente são encontrados. Na Figura 5.1 é reproduzida a interface gráfica da Mothra, extraída do artigo sobre geração automática de casos de teste [116]. Como se trata de uma reprodução da figura que aparece no artigo, a qualidade não é a ideal.

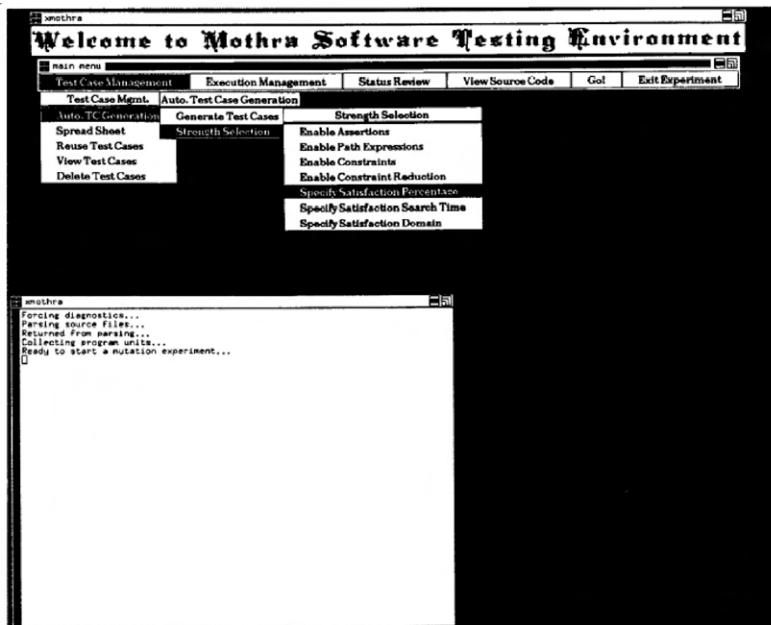


Figura 5.1 – Interface gráfica da ferramenta Mothra.

### 5.5.2 Proteum

A versão original (1.0) da ferramenta Proteum foi apresentada inicialmente no Simpósio Brasileiro de Engenharia de Software de 1993, no Rio de Janeiro [109]. Era um programa monolítico, que possuía uma interface gráfica e implementava basicamente as mesmas funcionalidades da Mothra para a linguagem C, utilizando o conjunto de operadores definidos por Agrawal et al. [7].

A realização de estudos experimentais que envolvem critérios de teste, muitas vezes, requer a repetição de uma determinada tarefa como, por exemplo, a criação de conjuntos de casos de teste adequados ao critério. A utilização de uma ferramenta de apoio por meio de sua interface gráfica exige muita interação com o testador e, por isso, pode fazer com que a condução de tais estudos consuma muito tempo e torne-se inviável. Por essa razão, promoveu-se a completa reestruturação da ferramenta a partir de sua versão 1.4.

Nessa versão a estrutura monolítica da Proteum foi quebrada em diversos programas separados, cada um com uma determinada funcionalidade específica, como, por exemplo, a geração de mutantes, a inserção de casos de teste, a execução dos mutantes ou a produção de relatórios. Cada um desses programas pode ser executado a partir da linha de comandos do console e também dentro de scripts, o que permite que sessões completas de teste possam ser programadas e executadas em modo *batch*. A interface gráfica da ferramenta continua existindo, mas funciona apenas como uma “shell” que invoca cada um dos programas que compõem a ferramenta.

Além da facilidade de utilização em modo *batch*, algumas outras características distinguem a Proteum de outras ferramentas, em particular, da Mothra:

- Grande flexibilidade na geração de mutantes. O testador pode selecionar para cada operador de mutação a porcentagem de mutantes a ser gerada e o número máximo de mutantes por ponto de mutação. Além disso, os mutantes podem ser gerados incrementalmente, à medida que a sessão de teste se desenvolve.
- Abordagem compilada, em vez de interpretada. A ferramenta gera, para cada mutante, um descritor, que define como os programas mutantes devem ser criados a partir do programa original. Na execução dos mutantes a ferramenta não interpreta os mutantes, mas, sim, cria o fonte modificado e o compila, usando as mesmas ferramentas usadas para compilar o programa original. Esse mutante executável é então executado normalmente utilizando os casos de teste fornecidos. A principal vantagem dessa abordagem é que a execução de um programa compilado tende a ser muitas vezes mais rápida que a de um programa interpretado. Como temos, em geral, um número grande de mutantes, o tempo salvo na condução do teste é grande. Além disso, tanto o programa original quanto os mutantes executam no seu ambiente real, e não por meio de um interpretador, que pode introduzir discrepâncias no comportamento desses programas.

Por outro lado, essa abordagem introduz, para a execução de cada mutante, um tempo extra que corresponde à execução do compilador para a geração do mutante executável. Em geral, esse ônus é aceitável, uma vez que o tempo gasto para a compilação é proporcional ao tamanho do programa-fonte, e o tempo de execução está relacionado com fatores como estruturas de laço, comandos de entrada e saída, etc. Além disso, a ferramenta busca minimizar o tempo de compilação dos mutantes por meio da criação de um único arquivo-fonte que incorpora o código de diversos mutantes. Por exemplo, para os 100 primeiros mutantes cria-se um único arquivo-fonte que contém as partes comuns entre eles apenas uma vez e que replica as partes alteradas do código original. Esse programa-fonte possui, também, um mecanismo que permite selecionar, em tempo de execução, qual parte do código deve ser executada, dependendo de qual mutante se deseja executar.

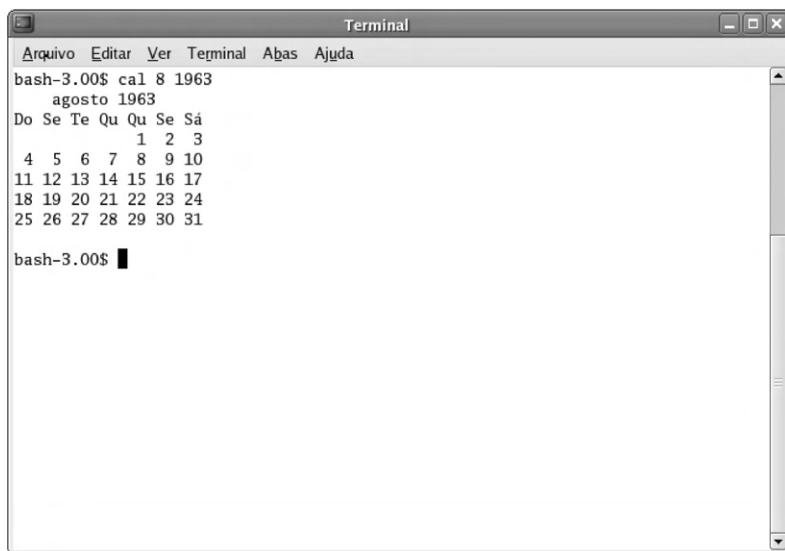
- Utilização de informação de fluxo de controle para evitar a execução de certos mutantes. Quando o testador fornece à ferramenta um dado de teste, ela executa uma

versão instrumentada do programa original, que permite a coleta de informações sobre os pontos do programa que foram efetivamente executados pelo caso de teste. Durante a execução dos mutantes, a ferramenta verifica se determinado caso de teste  $t$  executa o ponto do programa que sofreu a mutação. Caso não o execute,  $t$  não precisa ser utilizado para executar o mutante, pois sabe-se que o resultado seria o mesmo do programa original. Essa técnica reduz significativamente o número de execuções de mutantes e, portanto, o tempo gasto na condução do teste.

- Facilidade de importação de conjuntos de teste. Os casos de teste fornecidos à ferramenta não precisam ser inseridos interativamente, um a um, pelo testador. Para permitir a automatização e a execução de uma sessão de teste em modo *batch*, a Proteum permite que conjuntos de casos de teste sejam importados de três diferentes fontes: 1) arquivos texto normais; 2) outras sessões de teste da própria Proteum; ou 3) sessões de teste da POKE-TOOL [61], uma ferramenta de suporte ao teste estrutural, discutida em detalhes no Capítulo 4. Isso permite a integração dessas duas ferramentas, favorecendo a execução de estudos experimentais de comparação entre critérios de teste.
- Dois modos de execução dos mutantes: “teste” ou “pesquisa”. Na execução dos mutantes, o comportamento normal é interromper a execução de um mutante e descartá-lo quando se encontra o primeiro caso de teste que mate o mutante. Esse comportamento, padrão, é chamado na Proteum de modo teste. No modo pesquisa, cada mutante é executado, sempre, com todos os casos do conjunto de teste. Obviamente, esse comportamento é mais caro do que o modo teste, mas permite a coleta de informações importantes, principalmente no ambiente de pesquisa científica. Por exemplo, permite que se avalie o quanto um mutante é fácil de se matar contando-se o número de casos de teste que o distingue.
- Operações para habilitar/desabilitar casos de teste. Após inserir casos de teste, o testador pode manipulá-los de maneira bastante flexível, habilitando-os ou desabilitando-os. Um caso de teste desabilitado continua fazendo parte do conjunto de teste mas não é utilizado na execução dos mutantes. Com isso, o testador pode experimentar diversas combinações de casos de teste sem ter de efetivamente alterar o conjunto fornecido. Essa característica é, também, bastante útil para estudos experimentais.
- Da mesma forma, conjuntos de mutantes podem ser selecionados ou desselecionados de modo a permitir que o testador verifique o efeito de determinado conjunto de teste sobre diversos conjuntos distintos de mutantes.

A seguir ilustra-se o desenvolvimento de uma sessão de teste simples utilizando a Proteum. Usa-se, para isso, o programa *Cal*, que exibe o calendário de determinado ano ou de determinado mês. O programa pode ser invocado de três formas diferentes:

1. sem parâmetros: exibe o calendário do mês corrente;
2. com um único parâmetro numérico: exibe o calendário do ano cujo valor foi passado como parâmetro. Os valores válidos vão de 1 a 9999;
3. com dois parâmetros numéricos: o primeiro parâmetro representa o mês (1 a 12) e o segundo o ano (1 a 9999) do calendário a ser exibido.



```
Arquivo Editar Ver Terminal Ajuda
bash-3.00$ cal 8 1963
agosto 1963
Do Se Te Qu Qu Se Sá
          1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31

bash-3.00$
```

Figura 5.2 – Exemplo de execução do programa “Cal”.

Um exemplo da execução do comando com os argumentos "8 1963" é mostrado na Figura 5.2.

Na Figura 5.3 é ilustrada a janela principal da ferramenta. Para testar o programa, a primeira coisa a fazer é criar uma sessão de teste fornecendo alguns dados sobre o programa a ser testado. Na Figura 5.4 é apresentada a janela de criação da sessão para o programa Cal. Os dados a serem fornecidos são:

- o diretório no qual se encontram o programa-fonte e o programa executável e no qual serão criados os arquivos da sessão de teste;
- o nome da sessão de teste, uma vez que se podem criar diversas sessões para o mesmo programa;
- o nome do arquivo-fonte, no caso `Cal.c`;
- o nome do arquivo executável correspondente, no caso `Cal`;
- o comando de compilação usado para criar o executável a partir do fonte. Esse comando será usado para a criação dos mutantes executáveis, uma vez que os mutantes são compilados e não interpretados. Deve-se notar que isso permite ao testador escolher qualquer compilador e trabalhar, durante o teste, no ambiente real de execução do programa utilizando, inclusive, o seu compilador de produção; e
- o tipo de sessão de teste, que identifica o comportamento durante a execução dos mutantes, conforme descrito anteriormente.

Criada a sessão, ela permanece “aberta”, permitindo que o testador execute diversas operações como criação de mutantes ou casos de teste. O testador pode, também, abandonar

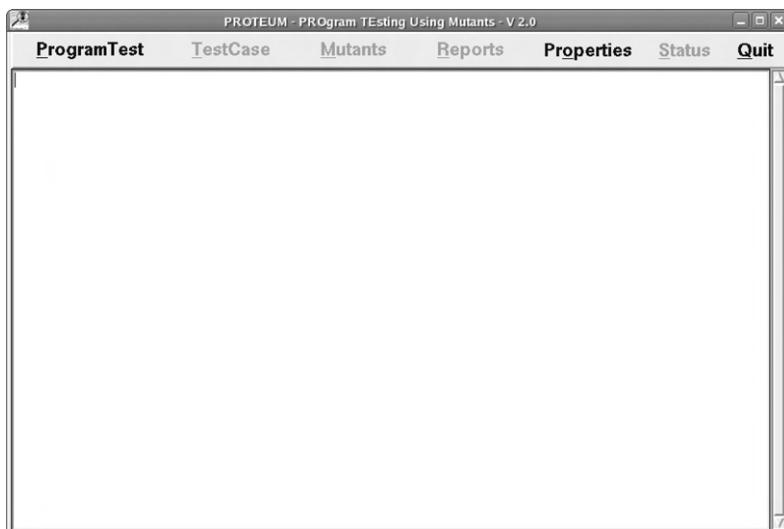


Figura 5.3 – Janela principal da ferramenta Proteum.

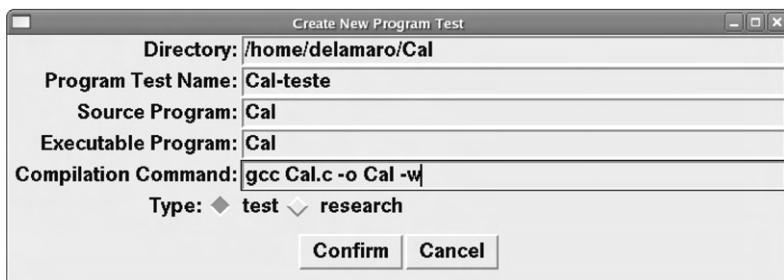
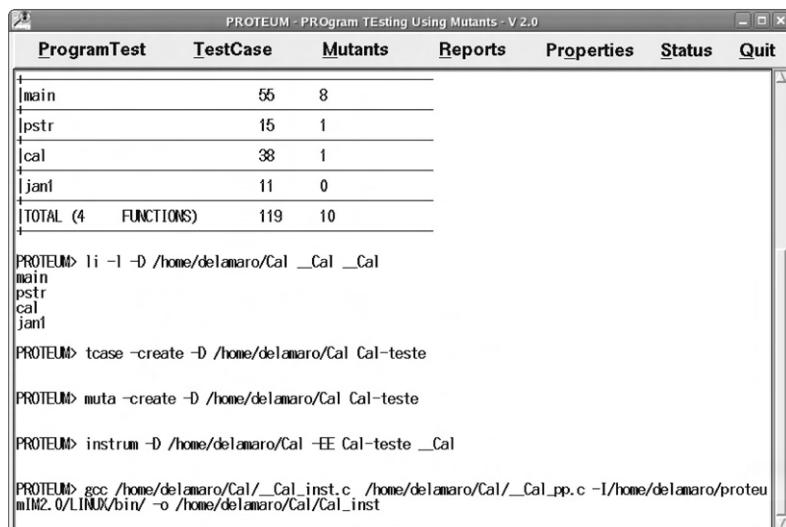


Figura 5.4 – Criação de uma sessão de teste.

a sessão de teste ou fechar a ferramenta e posteriormente recuperar a sessão do ponto que parou por meio da opção “Program Test/Load”, fornecendo o diretório no qual a sessão está armazenada e o seu nome.

Um ponto interessante a ser observado na Figura 5.5 é que, à medida que o testador realiza operações sobre a sessão de teste, a ferramenta exibe, na sua janela principal, quais seriam os comandos correspondentes para realizar aquela operação. Com isso, o testador menos experiente pode aprender, por meio da interface gráfica, a usar diretamente os comandos da ferramenta para utilizá-los em scripts de teste.

Em seguida são adicionados alguns casos de teste à sessão. Para isso, utilizamos a opção “TestCase/Add”, que abre uma pequena janela na qual é possível fornecer os argumentos iniciais para a execução do programa em teste. Como mostra a Figura 5.6, são fornecidos os valores 8 e 1963 para a execução do Cal. Isso feito, a ferramenta inicia a execução do programa com esses argumentos em uma janela separada. Essa execução é normal, como



The screenshot shows the PROTEUM - PROgram TEsting Using Mutants - V 2.0 application window. The menu bar includes ProgramTest, TestCase, Mutants, Reports, Properties, Status, and Quit. The main window displays a table of mutants and their counts, followed by a command history window containing the following text:

```

PROTEUM> li -l -D /home/delamaro/Cal _Cal __Cal
main
pstr
cal
jan1
TOTAL (4 FUNCTIONS) 119 10

PROTEUM> tcase -create -D /home/delamaro/Cal Cal-teste
PROTEUM> muta -create -D /home/delamaro/Cal Cal-teste
PROTEUM> instrum -D /home/delamaro/Cal -EE Cal-teste __Cal
PROTEUM> gcc /home/delamaro/Cal/_Cal_inst.c /home/delamaro/Cal/_Cal_pp.c -I/home/delamaro/proteu
mIM2.0/LINUX/bin/ -o /home/delamaro/Cal/cal_inst

```

Figura 5.5 – Visualização dos comandos correspondentes a uma operação da interface gráfica.

se o próprio usuário a tivesse iniciado, permitindo que o testador interaja com o programa fornecendo os dados de tempo de execução, quando necessário.

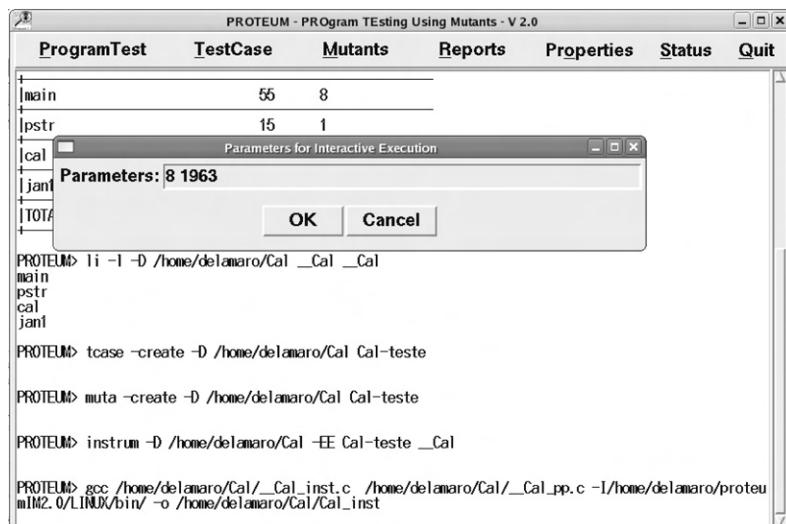


Figura 5.6 – Janela na qual são fornecidos os argumentos para a execução do programa.

No caso do programa `Cal`, como não existe interação com o usuário, o programa é executado completamente e a janela em que foi executado fecha-se ao seu término. Assim, para verificar se o programa comportou-se corretamente com esse dado de teste, utiliza-se a operação “TestCase/View”, que apresenta a janela ilustrada na Figura 5.7. A janela mostra o

número do caso de teste, os argumentos fornecidos, o tempo de execução (que será utilizado como parâmetro para decidir quando matar um mutante por “timeout”), o código de retorno (indicando que o programa finalizou normalmente), as entradas fornecidas por meio da entrada padrão (stdin) e as saídas produzidas pelo programa (stdout e stderr). Pode-se conferir que o resultado produzido corresponde ao esperado e, portanto, o programa foi executado com sucesso. O testador pode, também, habilitar ou desabilitar o caso de teste. Os casos de teste desabilitados não são utilizados nas próximas execuções dos mutantes, a menos que sejam novamente habilitados.

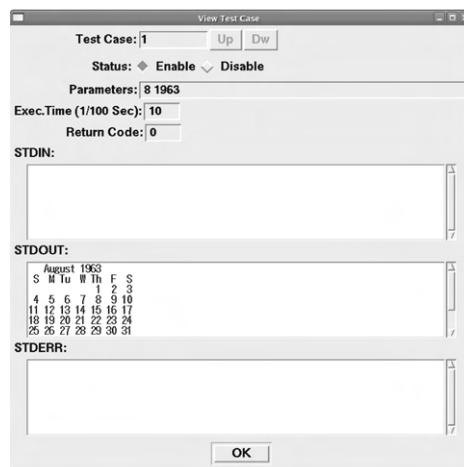


Figura 5.7 – Visualização de um caso de teste.

Antes de gerar os mutantes e avaliar os casos de teste, podemos fornecer mais alguns casos de teste que acharemos interessantes, como:

- caso de teste sem argumentos;
- caso de teste com um único argumento: 1963;
- caso de teste com um único argumento, ano bissexto: 1972;
- caso de teste com dois argumentos, ano bissexto: 2 1972;
- caso de teste com ano inválido: 10000; e
- caso de teste com mês inválido: 13 1972.

Utilizando a opção “Status”, pode-se conferir o andamento geral da sessão de teste, como mostra a Figura 5.8. Observe que os sete casos de teste foram devidamente inseridos na sessão.

Para avaliar a qualidade desse conjunto de casos de teste alguns mutantes são gerados e o escore de mutação obtido é verificado. Inicialmente, é escolhida a opção “Mutants/Generate Unit” para se ter acesso aos operadores de teste de unidade. Esses operadores são divididos em quatro grupos distintos:

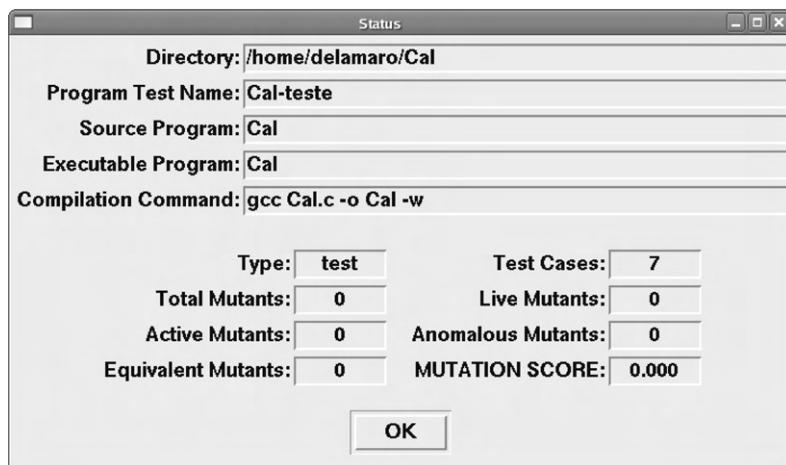


Figura 5.8 – Status inicial da sessão.

- operadores de operadores, são aplicados sobre operadores da linguagem C, como, por exemplo, troca de operadores relacionais;
- operadores de comandos, como, por exemplo, eliminação de um comando;
- operadores de variáveis, como troca de uma variável por outra; e
- operadores de constantes, como troca de uma constante por outra.

São gerados mutantes apenas para o primeiro grupo. Dois parâmetros podem ser escolhidos para cada operador: na primeira coluna do lado direito na janela “Operators Unit Mutants”, na Figura 5.9 pode-se escolher a porcentagem de mutantes desejada. Na segunda coluna, pode-se estabelecer o número máximo de mutantes por ponto de aplicação. Por exemplo, no caso de troca de operador relacional têm-se sempre cinco mutantes gerados em cada ponto de aplicação.<sup>1</sup> Entretanto, caso seja estabelecido que o número máximo de mutantes a ser gerado é dois, então são selecionados, aleatoriamente, apenas dois mutantes em cada ponto de mutação. A terceira coluna ilustra, depois da geração, o número de mutantes gerados. No exemplo da Figura 5.9 foram escolhidos 30% dos mutantes e o máximo de dois mutantes para todos os operadores de mutação. Com esses valores, tem-se um total de 247 mutantes gerados.

Em seguida, os mutantes são executados usando-se a opção “Mutants/Execute”. A ferramenta utiliza o conjunto de casos de teste definido para executar cada um dos 247 mutantes e decidir quais morrem e quais continuam vivos. Apesar do alto número de mutantes que pode ser gerado, as estratégias de redução da execução de tempo da Proteum permitem que se gastem apenas alguns segundos na execução desses mutantes. Ao consultar novamente a janela de status, obtém-se o resultado mostrado na Figura 5.10.

De acordo com a Figura 5.10, o número de mutantes que não foram mortos pelos casos de teste é 26, o que resulta em um escore de mutação de 0,894. Certamente, esse resultado

<sup>1</sup>Considerando os operadores relacionais: < > == >= <= !=.

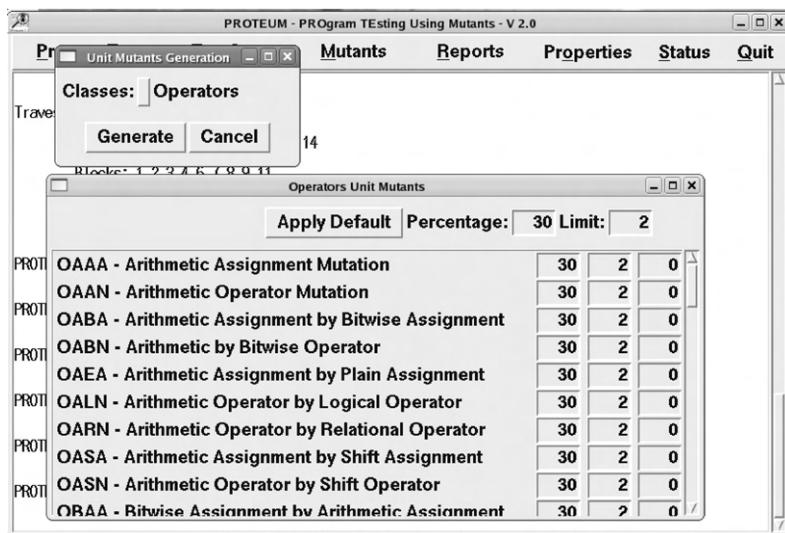


Figura 5.9 – Geração de mutantes de operadores.

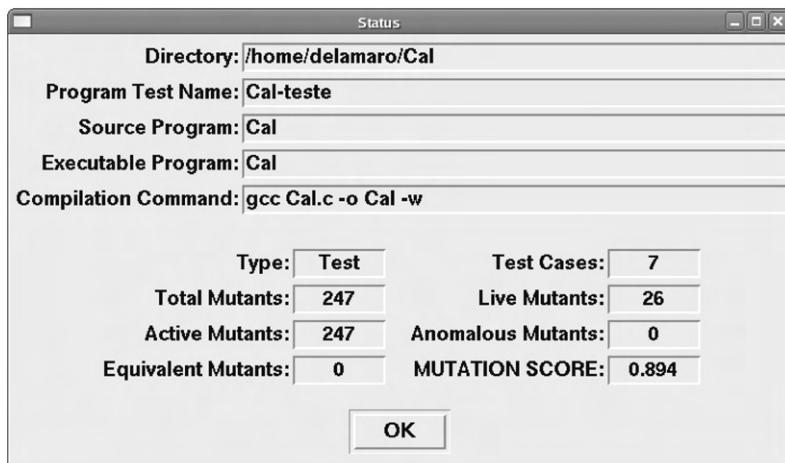
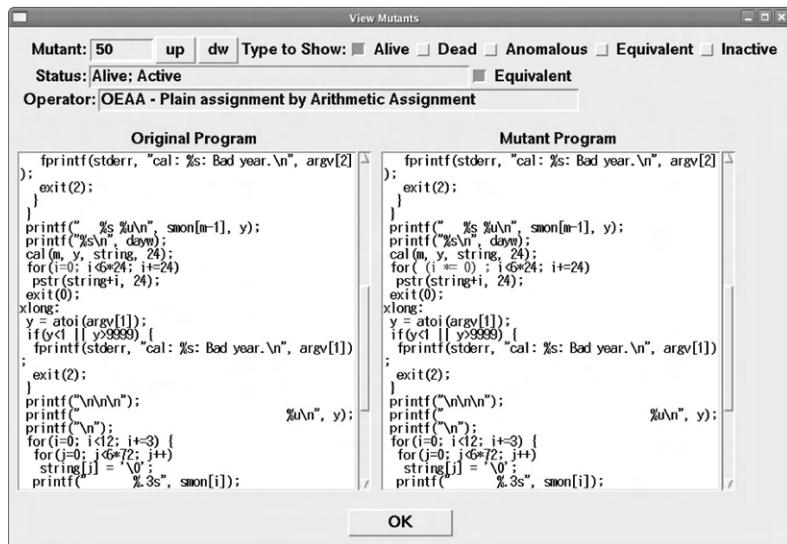


Figura 5.10 – Primeira execução dos mutantes.

pode ser melhorado analisando-se os mutantes vivos e decidindo-se se são equivalentes ou se é possível criar novos casos de teste que os matem. Para analisar os mutantes, utiliza-se a opção “Mutants/View”, o que faz aparecer a janela mostrada na Figura 5.11.

Na figura é mostrado o número do mutante que está sendo visualizado e seu código do lado direito da janela, com o trecho que sofreu mutação destacado em vermelho. O código do programa original é mostrado do lado esquerdo da janela. No topo da janela, o testador pode selecionar o tipo de mutante que deseja visualizar. No caso, apenas os mutantes vivos estão sendo visualizados. Analisando o mutante número 50, observa-se que a substituição do



The screenshot shows a software interface titled "View Mutants". At the top, there are fields for "Mutant: 50", "up", "dw", and "Type to Show:" with checkboxes for "Alive", "Dead", "Anomalous", "Equivalent", and "Inactive". Below these are "Status: Alive; Active" and "Operator: OEEA - Plain assignment by Arithmetic Assignment". The main area is divided into two panes: "Original Program" and "Mutant Program".

```

Original Program
```c
fprintf(stderr, "cal: %s: Bad year.\n", argv[2]);
exit(2);
}
printf("%s %u\n", smon[m-1], y);
printf("%s\n", day);
cal(m, y, string, 24);
for(i=0; i<24; i+=24)
    pstr(string+i, 24);
exit(0);
xlong;
y = atoi(argv[1]);
if(y<1 || y>9999) {
    fprintf(stderr, "cal: %s: Bad year.\n", argv[1]);
    exit(2);
}
printf("\n\n\n");
printf("\n");
printf("\n");
for(i=0; i<2; i+=3) {
    for(j=0; j<6/2; j++)
        string[j] = '\0';
    printf("%s", smon[i]);
}
```

```

```

Mutant Program
```c
fprintf(stderr, "cal: %s: Bad year.\n", argv[2]);
exit(2);
}
printf("%s %u\n", smon[m-1], y);
printf("%s\n", day);
cal(m, y, string, 24);
for(i=0; i<24; i+=24)
    pstr(string+i, 24);
exit(0);
xlong;
y = atoi(argv[1]);
if(y<1 || y>9999) {
    fprintf(stderr, "cal: %s: Bad year.\n", argv[1]);
    exit(2);
}
printf("\n\n\n");
printf("\n");
printf("\n");
for(i=0; i<2; i+=3) {
    for(j=0; j<6/2; j++)
        string[j] = '\0';
    printf("%s", smon[i]);
}
```

```

Figura 5.11 – Análise dos mutantes vivos.

operador de atribuição no comando `i = 0` por `i *= 0` cria um mutante equivalente. Desse modo, marca-se o mutante como tal, selecionando o botão “Equivalent” no topo da janela de visualização.

O mutante mostrado na Figura 5.12, por sua vez, não é equivalente e deve-se estabelecer uma forma de matá-lo. Analisando o código do programa, observa-se o seguinte: para matar esse mutante, é suficiente fornecer um caso de teste que contenha mês e ano, mas que este último possua valor não válido. Volta-se, então, à opção “TestCase/Add”, inserindo esse caso de teste e reexecutando os mutantes.

Depois de reexecutar os mutantes, observando novamente a janela de status, tem-se que com o mutante marcado como equivalente e o novo caso de teste foi possível aumentar (um pouco) o escore de mutação. Dessa forma, o conjunto de casos de teste inicial foi melhorado. Todavia, ainda assim, não se chegou ao final do teste, pois outros mutantes continuam vivos e devem ser analisados. A sessão de teste deve prosseguir nessa iteração de analisar mutantes e inserir casos de teste até que um conjunto adequado seja obtido. Pode-se ainda, dependendo da necessidade da aplicação e dos recursos disponíveis, gerar mais mutantes e voltar a procurar casos de teste para matá-los, sempre com o intuito de aprimorar o conjunto de casos de teste.

Recentemente, algumas novas ferramentas de suporte ao Teste de Mutação surgiram, buscando sua utilização em diferentes domínios, ambientes ou linguagens. A própria Proteum evoluiu em diversas direções, compondo, atualmente, uma família de ferramentas que podem ser utilizadas nos mais diversos domínios, incluindo uma interface para a Web. Pesquisadores dos Estados Unidos e Coréia, em colaboração, desenvolveram uma ferramenta de mutação para programas Java chamada  $\mu$ Java [256]. Essa pode ser uma ferramenta de interesse para programadores nessa linguagem, estando disponível no endereço <http://www.isse.gmu.edu/faculty/ofut/Ma05MACM/>.

**View Mutants**

Mutant: 69 up dw Type to Show:  Alive  Dead  Anomalous  Equivalent  Inactive

Status: Alive; Active  Equivalent

Operator: OLBN - Logical Operator by Bitwise Operator

**Original Program**

```

tm = (struct tm *) localtime(&t);
m = tm->tm_mon + 1;
y = tm->tm_year + 1900;
else {
    m = atoi(argv[1]);
    if(m<1 || m>12) {
        fprintf(stderr, "cal: %s: Bad month.\n", argv[1]);
        exit(1);
    }
    y = atoi(argv[2]);
    if(y<1 || y>9999) {
        fprintf(stderr, "cal: %s: Bad year.\n", argv[2]);
        exit(2);
    }
    printf("%s %u\n", mon[m-1], y);
    printf("%s\n", day);
    cal(m, y, string, 24);
    for(i=0; i<24; i+=2)
        pstr(string+i, 24);
    exit(0);
}
long:
y = atoi(argv[1]);

```

**Mutant Program**

```

tm = (struct tm *) localtime(&t);
m = tm->tm_mon + 1;
y = tm->tm_year + 1900;
else {
    m = atoi(argv[1]);
    if(m<1 || m>12) {
        fprintf(stderr, "cal: %s: Bad month.\n", argv[1]);
        exit(1);
    }
    y = atoi(argv[2]);
    if((y < 1) & (y > 9999)) {
        fprintf(stderr, "cal: %s: Bad year.\n", argv[2]);
        exit(2);
    }
    printf("%s %u\n", mon[m-1], y);
    printf("%s\n", day);
    cal(m, y, string, 24);
    for(i=0; i<24; i+=2)
        pstr(string+i, 24);
    exit(0);
}
long:
y = atoi(argv[1]);

```

OK

Figura 5.12 – Mutante não equivalente.

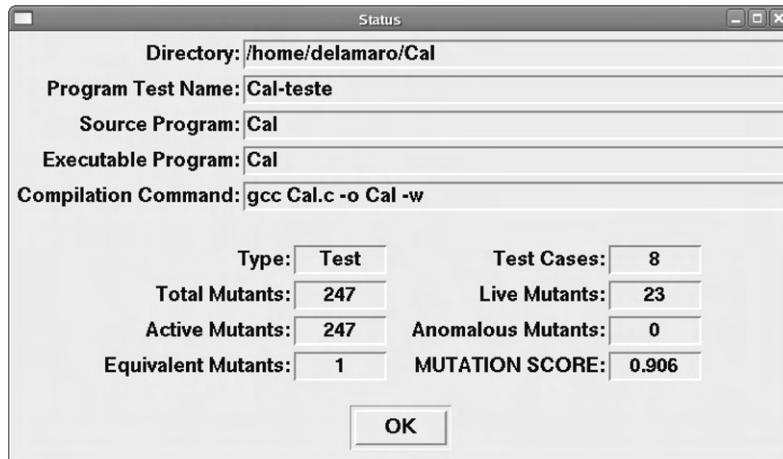


Figura 5.13 – Segunda execução dos mutantes.

## 5.6 Mutação de Interface

Conforme discutido nas seções anteriores, a idéia básica do Teste de Mutação é avaliar um conjunto de teste baseado na sua habilidade em revelar defeitos simples, injetados no programa sendo testado. Tais defeitos procuram modelar casos comuns de erros. Um conjunto que atenda a esse requisito deve ser também eficaz em revelar outros erros, causados por defeitos mais complexos.

Ao se aplicar o Teste de Mutação a uma unidade que faz parte de um programa maior, o conjunto de mutantes criado é sempre o mesmo, independente de como a unidade é usada (chamada) no programa. Por exemplo, se a unidade é usada em diversos pontos do programa, não se garante que todos esses pontos serão exercitados pelos casos de teste selecionados. Podem-se criar casos de teste que distinguem os mutantes mas executem apenas caminhos que passem por um desses pontos. Em resumo, o Teste de Mutação é um critério efetivo para o teste da estrutura interna da unidade, mas não necessariamente para exercitar as interações entre unidades num programa “integrado”.

No nível de integração é necessário testar não a unidade em si, mas principalmente as interações com as outras unidades do programa. Esse é o objetivo do critério Mutação de Interface [108]. De acordo com o modelo de falhas apresentado por Delamaro [108], 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 objetos relacionados com a interação entre as unidades.

Em outras palavras, os mutantes a serem distinguidos pelos casos de teste no nível de integração modelam “erros simples de integração”. Mais precisamente, como define Offut [307], um mutante deve ser um programa cuja diferença semântica em relação ao programa em teste seja pequena. No caso da Mutação de Interface, essa diferença deve se manifestar nas conexões entre as unidades do programa. Dessa forma, conjuntos adequados à Mutação de Interface (MI-adequados) devem ser, de acordo com o efeito de acoplamento, efetivos para revelar a presença de outros tipos de defeitos que se manifestem por meio de erros nas interações entre as unidades.

Em linguagens convencionais como C, Pascal, Fortran, etc., unidades são conectadas por meio de chamadas de subprogramas (funções, procedimentos ou sub-rotinas). Em uma chamada da unidade  $F$  à unidade  $G$ , existem quatro maneiras, não mutuamente exclusivas, de se trocarem dados entre as unidades:

- dados são passados para  $G$  por meio de parâmetros de entrada (passagem por parâmetro);
- dados são passados para  $G$  e/ou retornados para  $F$  por meio de parâmetros de entrada-saída (passagem por referência);
- dados são passados para  $G$  e/ou retornados para  $F$  por meio de variáveis globais; e
- dados são retornados para  $F$  por meio de comandos de retorno (como *return* em C).

A Figura 5.14 ilustra um programa, composto por algumas conexões, sobre o qual se aplica a Mutação de Interface. O testador pode, por exemplo, escolher testar a conexão  $A-B$ , e, nesse caso, um operador de mutação OP é aplicado aos pontos relacionados apenas a essa conexão. Em outras palavras, o critério é sempre aplicado de uma forma “ponto a ponto”, limitando os requisitos de teste a uma determinada conexão.

Para testar uma conexão A-B, introduzindo defeitos simples que levem a erros de integração, a Mutação de Interface aplica mutações em: 1) pontos nos quais a unidade  $A$  chama a unidade  $B$ , portanto antes da execução de  $B$ ; e 2) pontos dentro de  $B$  relacionados com sua interface.

Um ponto importante para se aplicar Mutação de Interface é que os operadores de mutação devem contemplar a idéia de que uma conexão é testada e não uma unidade. Por exemplo,

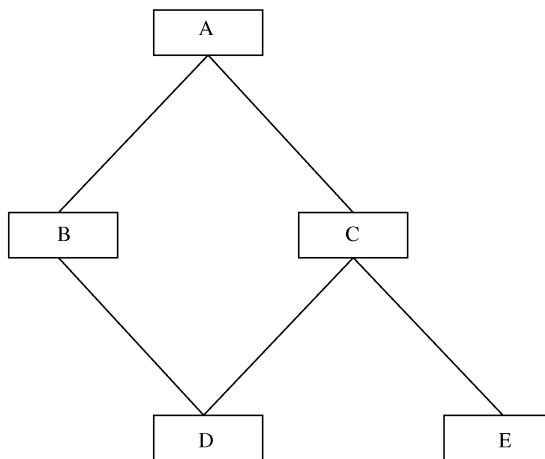


Figura 5.14 – Estrutura de um programa, utilizada para exemplificar algumas características da Mutação de Interface.

caso se deseje testar a conexão  $C-D$ , um mutante cuja mutação  $OP(v)$  seja efetuada dentro da unidade  $D$  pode ser distinguido por meio de uma chamada de  $D$  feita em  $B$ . Nesse caso, a conexão desejada não está sendo exercitada. Assim, a aplicação de  $OP$ , quando feita dentro da unidade chamada, deve levar em consideração o ponto em que foi feita a chamada. No exemplo, a mutação só pode ser “ativada” se a unidade  $D$  for chamada por  $C$ . Quando chamada por  $B$ ,  $D$  deve se comportar exatamente como no programa original, fazendo com que a conexão  $C-D$  tenha de ser exercitada para distinguir o mutante. Para algumas linguagens, como C por exemplo, isso significa que a decisão de aplicar-se ou não a mutação precisa ser feita em tempo de execução do mutante. Para mutações feitas no ponto em que  $C$  chama  $D$ , esse problema não existe pois, nesse caso, o ponto de mutação garante que a conexão desejada está sendo exercitada.

Segundo DeMillo e Offut [116], existem três condições que devem ser satisfeitas para que um caso de teste  $t$  distinga um mutante  $M$ :

1. alcançabilidade: a execução de  $M$  e, consequentemente, do programa original  $P$  com  $t$  deve passar pelo ponto no qual a mutação foi feita;
2. necessidade: os estados de  $P$  e do mutante  $M$  devem diferir, imediatamente após alguma execução do comando no qual a mutação foi aplicada; e
3. suficiência: essa diferença no estado interno de  $P$  e  $M$  deve ser propagada até um ponto de maneira que  $M$  produza uma saída diferente de  $P$ , já que no Teste de Mutação apenas as saídas são consideradas para distinguir os mutantes.

No contexto da Mutação de Interface, quando o operador é aplicado dentro da unidade que é chamada, pode-se afirmar que a condição de alcançabilidade foi alterada. Para distinguir um mutante  $M$ , o ponto de mutação tem de ser alcançado por um caminho que inclua a conexão que se deseja testar. Mais precisamente, redefine-se a condição de alcançabilidade para esses casos como:

*Condição de alcançabilidade para a Mutação de Interface:* a execução de um mutante **M**, gerado para a conexão *A-B*, com um caso de teste *t* deve fazer com que o ponto no qual a mutação foi introduzida seja alcançado por meio de uma chamada de *A* para *B*.

Operadores de Mutação de Interface são relacionados a uma conexão entre duas unidades e cada mutante é relacionado a uma chamada de função. Por exemplo, em um programa com uma função *f* que faça duas chamadas a uma função *g*, como na Figura 5.15, a cada chamada de *g* corresponde um conjunto de mutantes que só podem ser distinguidos por meio da execução da chamada correspondente. Considerando *g'* a função criada ao se aplicar um operador *OP* em um ponto no interior de *g*, têm-se dois mutantes distintos relacionados ao teste da conexão *f-g*. O primeiro é aquele no qual a primeira chamada de *g* é substituída pela chamada a *g'* e o segundo é aquele no qual a segunda chamada é substituída, como exemplificado na Figura 5.16.

Por isso, como citado anteriormente, uma mutação aplicada dentro da função chamada só deve ser efetivamente aplicada se a função foi chamada do ponto cuja conexão se deseja testar. No exemplo da Figura 5.16, no primeiro mutante, que foi gerado para a primeira chamada a *g*, deve-se substituir *g* pela chamada a *g'*. No entanto, a segunda chamada deve permanecer inalterada, ou seja, nessa segunda chamada a função *g* deve comportar-se exatamente como no programa original, de modo que **M<sub>1</sub>** só possa ser distinguido por meio da primeira chamada. Somente desse modo pode-se exercitar de maneira mais completa a conexão que, na verdade, é composta por mais de uma chamada de função.

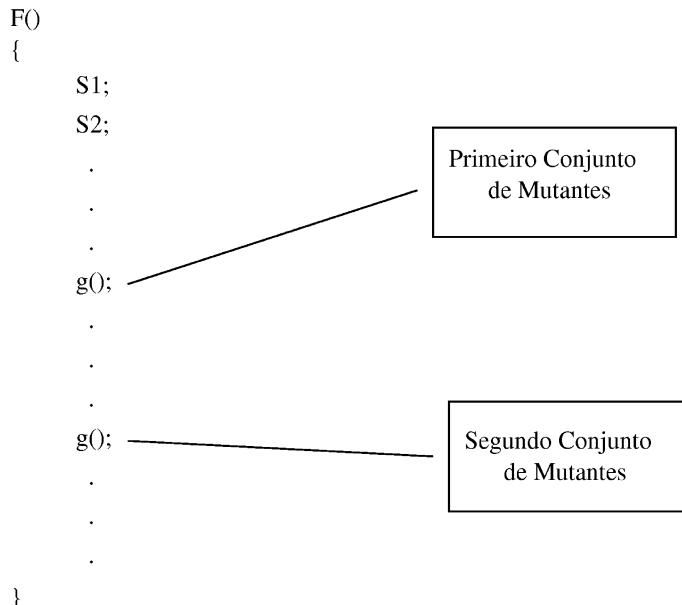


Figura 5.15 – Indicação dos conjuntos de mutantes requeridos para o teste da conexão *f-g*.

Dois grupos de operadores foram definidos para a Mutação de Interface e a linguagem C. O primeiro grupo é composto por operadores que, ao se testar a conexão *f-g*, são aplicados

```

f()
{
    S1;
    S2;
    .
    .
    .
    g'();
    .
    .
    .
    g();
    .
    .
    .
}
    f()
    {
        S1;
        S2;
        .
        .
        .
        g();
        .
        .
        .
    }

```

Figura 5.16 – Mutantes associados a diferentes chamadas da conexão  $f-g$ .

dentro da função  $g$ . O segundo é de operadores aplicados nos pontos em que a função  $f$  chama a função  $g$ . Na definição dos operadores são utilizados os seguintes conjuntos, considerando que a Mutação de Interface está sendo aplicada na conexão entre as funções  $f$  e  $g$ :

$P(g)$ : é o conjunto dos parâmetros formais de  $g$ . Esse conjunto inclui também derreferências a parâmetros do tipo ponteiro ou vetor. Por exemplo, se um parâmetro formal  $v$  é definido como “int \*v”,  $v$  e “\*v” pertencem a esse conjunto;<sup>2</sup>

$G(g)$ : é o conjunto de variáveis globais utilizadas na função  $g$ ;

$L(g)$ : é o conjunto de variáveis declaradas em  $g$  (variáveis locais);

$E(g)$ : é o conjunto de variáveis globais não utilizadas em  $g$ ; e

$C(g)$ : é o conjunto de constantes utilizadas em  $g$ .

Os elementos do conjunto  $G$  são chamados de “variáveis globais” e os do conjunto  $E$  são chamados de “variáveis externas”. Os elementos pertencentes aos conjuntos  $P$  e  $G$  são aqueles por meio dos quais valores podem ser passados para a função  $g$  ou dela retornados, fazendo parte da interface da função. Assim, os elementos desses dois conjuntos são chamados de “variáveis de interface”. Os elementos dos demais conjuntos são chamados de “variáveis não de interface e constantes”.

$R$  é o conjunto de “constantes requeridas”. Contém valores especiais, relevantes para alguns tipos primitivos da linguagem C e operações associadas a esses tipos. A Tabela 5.7 sumaria essas constantes. A Tabela 5.8 apresenta um resumo com todos os operadores de Mutação de Interface definidos.

<sup>2</sup>Como a linguagem C faz passagem de parâmetros sempre por valor e a passagem por referência é simulada passando-se como valor o endereço da variável, incluiu-se no conjunto  $P(g)$  a derreferenciação dos parâmetros ponteiros ou vetores, de forma que esse tipo de interface seja exercitado de maneira completa.

Tabela 5.7 – Conjunto de constantes requeridas

| Tipo de variável  | Constantes requeridas                   |
|---|---|
| signed integer  | -1, 1, 0, <i>MAXINT</i> , <i>MININT</i> |
| signed char   |   |
| signed long   |   |
| unsigned integer  | -1, 1, 0, <i>MAXUNSIGNED</i>            |
| unsigned char   |   |
| unsigned long   |   |
| enum  |   |
| float   | -1.0, 1.0, 0.0, -0.0                    |
| double  |   |
| As constantes <i>MAXINT</i> , <i>MININT</i> e <i>MAXUNSIGNED</i> correspondem respectivamente ao maior inteiro positivo, ao inteiro mais negativo e ao maior inteiro sem sinal. Esses valores são dependentes de máquina e de tipos de dados. |   |

Por fim, com a definição de Mutação de Interface, criou-se uma nova versão da ferramenta Proteum (discutida na Seção 5.5) com vistas a apoiar a aplicação desse critério. A *PROTEUM/IM* [108], como foi denominada, possui arquitetura e implementação similares às da Proteum. O que as diferencia é o conjunto de operadores de mutação utilizados em cada uma e o fato de a *PROTEUM/IM* oferecer recursos para testar a conexão entre as unidades do software. Posteriormente, a *PROTEUM/IM* fundiu-se com a Proteum e criou-se, assim, a versão atual da ferramenta que apóia, de forma integrada, um único ambiente de teste, o Teste de Mutação e a Mutação de Interface.

## 5.7 Outros trabalhos

Muitos trabalhos têm sido desenvolvidos no contexto do Teste de Mutação. Inúmeros aspectos, tais como estudos teóricos e experimentais, extensões ou adaptações do critério, definição de operadores de mutação, construção de ferramentas, entre outros, têm sido explorados. Alguns deles foram discutidos nas seções anteriores. Nesta seção é dada ênfase a uma linha de pesquisa importante relacionada ao Teste de Mutação – o teste de modelos de software.

Como visto na Seção 5.6, a técnica de mutação é bastante flexível. Pode-se mudar o enfoque do teste, passando do teste de unidade para o teste de integração, realizando, basicamente, mudanças no conjunto de operadores de mutação utilizado. Na verdade, a flexibilidade propiciada pelo Teste de Mutação vai além disso, sendo possível utilizá-lo para validar qualquer tipo de artefato executável, não apenas programas.

De modo mais preciso, para que seja possível aplicar o Teste de Mutação é preciso apenas definir um conjunto significativo de operadores de mutação e uma forma de avaliar se o resultado produzido pelo artefato original é o mesmo de cada um dos seus mutantes, para um dado conjunto de teste. É o que se mostra na Figura 5.17. O processo *S* representa um “executor”, como o sistema operacional executando um programa em linguagem de máquina ou um interpretador executando um programa-fonte ou um simulador reproduzindo o comporta-

Tabela 5.8 – Operadores de Mutação de Interface

| GRUPO I        |   |
|----------------|---|
| DirVarRepPar   | troca variáveis de interface por elementos de $P$               |
| DirVarRepGlob  | troca variáveis de interface por elementos de $G$               |
| DirVarRepLoc   | troca variáveis de interface por elementos de $L$               |
| DirVarRepExt   | troca variáveis de interface por elementos de $E$               |
| DirVarRepConst | troca variáveis de interface por elementos de $C$               |
| DirVarRepReq   | troca variáveis de interface por elementos de $R$               |
| IndVarRepPar   | troca variáveis não de interface por elementos de $P$           |
| IndVarRepGlob  | troca variáveis não de interface por elementos de $G$           |
| IndVarRepLoc   | troca variáveis não de interface por elementos de $L$           |
| IndVarRepExt   | troca variáveis não de interface por elementos de $E$           |
| IndVarRepConst | troca variáveis não de interface por elementos de $C$           |
| IndVarRepReq   | troca variáveis não de interface por elementos de $R$           |
| DirVarIncDec   | acrescenta incremento e decremento em variável de interface     |
| IndVarIncDec   | acrescenta incremento e decremento em variável não de interface |
| DirVarAriNeg   | acrescenta negação aritmética em variáveis de interface         |
| IndVarAriNeg   | acrescenta negação aritmética em variáveis não de interface     |
| DirVarLogNeg   | acrescenta negação lógica em variáveis de interface             |
| IndVarLogNeg   | acrescenta negação lógica em variáveis não de interface         |
| DirVarBitNeg   | acrescenta negação de bit em variáveis de interface             |
| IndVarBitNeg   | acrescenta negação de bit em variáveis não de interface         |
| RetStaDel      | elimina comando de retorno                                      |
| RetStaRep      | troca comando de retorno  |
| CovAllNode     | garante cobertura de nós  |
| CovAllEdge     | garante cobertura de desvios                                    |
| GRUPO II       |   |
| ArgRepReq      | troca argumentos por elementos de $R$                           |
| ArgIncDec      | incrementa e decremente argumento                               |
| ArgSwiAli      | troca posição de argumentos de tipos compatíveis                |
| ArgSwiDif      | troca posição de argumentos de tipos diferentes                 |
| ArgDel         | elimina argumento   |
| ArgAriNeg      | acrescenta negação aritmética antes de argumento                |
| ArgLogNeg      | acrescenta negação lógica antes de argumento                    |
| ArgBitNeg      | acrescenta negação de bit antes de argumento                    |
| FuncCalDel     | elimina chamada de função                                       |

mento de uma Máquina de Estados Finitos. O processo **D** é um “comparador” que decide se o resultado de duas execuções de **S** é igual.

Dessa forma, alguns trabalhos têm utilizado o Teste de Mutação para validar modelos de software descritos por técnicas como, por exemplo, Máquinas de Estados Finitos [130], Redes

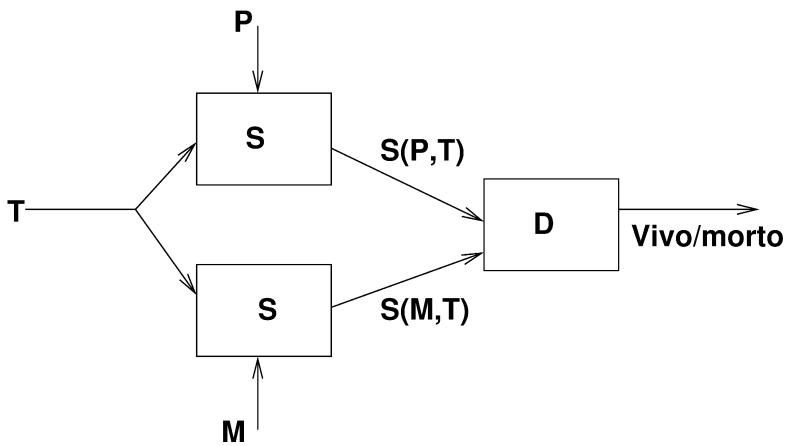


Figura 5.17 – Processos necessários para a aplicação do Teste de Mutação.

de Petri [134] ou Statecharts [135]. Como exemplo, considere a MEF da Figura 5.18(a). Sua execução com uma seqüência de eventos como  $s_1 = \langle T_{Creq}, CC, DT \rangle$  produz como saída  $o_1 = \langle CR, T_{Cconf}, T_{DTind} \rangle$ .

A MEF da Figura 5.18(b) representa um mutante da MEF original criado pela aplicação do operador de mutação que troca uma das saídas da máquina. A transição afetada pela mutação tem origem e destino no estado 4. É importante observar que a seqüência  $s_1$  não é capaz de distinguir o comportamento desse mutante. Seria preciso, por exemplo, uma seqüência  $s_2 = \langle T_{Creq}, CC, T_{DTreq} \rangle$ , que na MEF original produziria a saída  $o_2 = \langle CR, T_{Cconf}, DT \rangle$  e na máquina mutante produziria  $o_2 = \langle CR, T_{Cconf}, CC \rangle$ .

A ferramenta Proteum ganhou sucessores também nessa área. Foram construídas ferramentas de suporte ao Teste de Mutação tanto para máquinas de estados quanto para Redes de Petri e Statecharts. Na Figura 5.19 ilustra-se uma das janelas da Proteum-RS/PN que trata de especificações desenvolvidas usando Redes de Petri [360]. Assim como a Proteum no nível de programas, a Proteum-RS/PN possui uma interface gráfica e, também, uma interface de comandos, o que permite a criação de sessões de teste em modo *batch*.

Algumas das principais características da ferramenta, descritas por Simão [361], são:

- sessão de teste: a Proteum-RS/PN utiliza a noção de sessão de teste. Dessa forma, pode-se interromper uma sessão e retomá-la posteriormente. Além disso, pode-se realizar mais de uma sessão para o teste de uma mesma rede;
- edição/simulação de RPs: a ferramenta permite a criação e a edição de RPs, possuindo um editor e simulador de Rede de Petri acoplado à ferramenta;
- casos de teste: no contexto de RPs, os casos de teste são caracterizados pela seqüência de transições a ser disparada. Os casos de teste podem ser incluídos em uma sessão de diversas formas. Por exemplo, um caso de teste pode ser incluído interativamente por meio do simulador de Redes de Petri. Caso o comportamento da rede esteja de acordo com o esperado, pode-se, então, incluir o caso de teste. A ferramenta também oferece

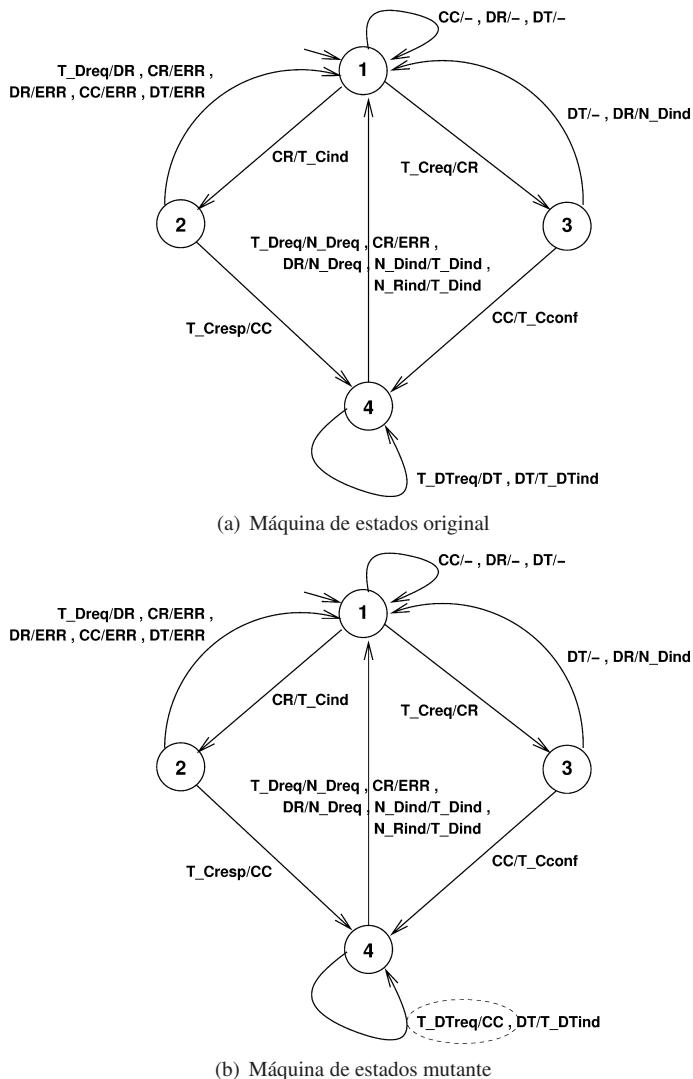


Figura 5.18 – Exemplo de mutação em uma MEF.

opções para a importação de casos de teste a partir de outra sessão da própria Proteum-RS/PN ou de um arquivo ASCII. Um módulo para auxiliar a geração automática de casos de teste também existe, incorporado à ferramenta;

- mutantes: a ferramenta implementa todos os operadores de mutação para RPs definidos por Fabbri [130]. Para dar suporte a estudos experimentais, é possível escolher quais operadores aplicar e que porcentagem de mutantes de cada operador será gerada. Os mutantes podem então ser executados com os casos de teste e pode-se calcular o escore de mutação. Apenas os mutantes e os casos de teste selecionados são considerados durante a execução e o cálculo do escore de mutação; e

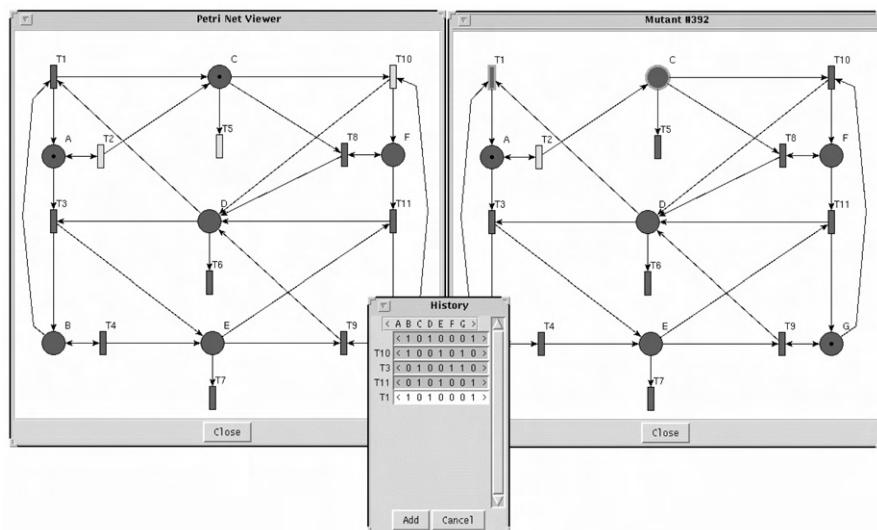


Figura 5.19 – Janela da Proteum-RS/PN.

- relatórios: além do escore de mutação, a ferramenta oferece relatórios mais detalhados sobre a sessão de teste. Por exemplo, pode-se solicitar um relatório sobre os casos de teste, com diversas informações, tais como quantos mutantes foram executados e quantos foram mortos com cada caso de teste. Outro relatório importante é o que informa o status dos mutantes agrupados por operador de mutação.

## 5.8 Considerações finais

Neste capítulo procurou-se mostrar uma forma “diferente” de avaliar um conjunto de casos de teste. O Teste de Mutação usa conceitos com os quais a maioria das pessoas não está habituada a tratar durante a atividade de teste. Quando se trata de técnicas de teste baseadas na implementação, a maioria dos desenvolvedores está acostumada a pensar em termos de cobertura de código, principalmente de comandos ou de desvios. Por esse motivo e, possivelmente, pelo custo associado ao Teste de Mutação, esse critério é, ainda, pouco conhecido e pouco utilizado. Por outro lado, estudos têm mostrado que se trata de um critério altamente eficaz em revelar defeitos.

A princípio, os conceitos e a teoria que suporta o critério podem parecer complicados. Contudo, a idéia por trás da técnica é, na verdade, simples. Suponha a seguinte situação: você, leitor, desenvolveu um programa a pedido do seu chefe e criou um conjunto de casos de teste **T** para testá-lo. Em determinado momento, você decide que o programa está pronto e que **T** é o conjunto de teste adequado para testá-lo. Então, você leva o programa ao seu chefe que, depois de reclamar que você está atrasado na entrega, lhe pergunta: “Testou ?”.

Você, orgulhosamente, responde que sim e, como prova, exibe o conjunto de teste **T** ao seu chefe. Ele, desconfiado, resolve desafiá-lo tomando o seu programa e executando-o com

todos os seus casos de teste. Obviamente, tudo funciona como o esperado. Não satisfeito, seu chefe decide realizar outra prova: toma seu código-fonte e altera-o, de forma que um dos comandos se comporte de maneira incorreta. Ao executar esse novo programa que seu chefe criou, com o conjunto de casos de teste **T**, qual é o resultado que você espera obter?

Exato. Espera que, pelo menos para algum elemento de **T**, o resultado produzido seja incorreto, pois, afinal, seu programa está correto com **T** e o de seu chefe não deveria estar. Mas, se o resultado obtido pelo programa do seu chefe for exatamente igual ao seu, então podemos desconfiar que seu conjunto de teste não é assim tão eficiente, pois nem conseguiu “desbancar” o truque do seu chefe.

O Teste de Mutação nada mais é do que utilizar, de maneira sistemática, a idéia aplicada pelo seu chefe, criando uma série de implementações alternativas e forçando o testador a projetar casos de teste que revelem os defeitos nelas introduzidos. Os casos de teste gerados dessa forma devem ser tão sensíveis que seriam capazes de revelar, também, outros tipos de defeitos.

# Capítulo 6

## Teste Orientado a Objetos e de Componentes

*Auri Marcelo Rizzo Vincenzi (UNISANTOS)*

*André Luís dos Santos Domingues (ICMC/USP)*

*Márcio Eduardo Delamaro (UNIVEM)*

*José Carlos Maldonado (ICMC/USP)*

### 6.1 Introdução

No decorrer dos tempos, novos paradigmas de programação são desenvolvidos com o objetivo de suprir deficiências dos paradigmas já existentes. No caso da orientação a objetos, essa surgiu para tentar suprir deficiências, principalmente relacionadas ao paradigma procedimental. O objetivo do paradigma procedural é estruturar um problema em termos de um conjunto de dados e de um conjunto de procedimentos/funções que manipulam tais dados. Entretanto, quando grandes problemas precisam ser resolvidos utilizando-se tal paradigma, a dependência entre os procedimentos/funções e os dados torna-se muito grande, de modo que pequenas alterações em como os dados estão organizados podem levar a alterações profundas na forma com que os procedimentos/funções fazem acesso a esses dados.

Nesse sentido, o paradigma de programação orientada a objetos surgiu a fim de fornecer um mecanismo para isolar os dados da forma como são manipulados. A idéia por trás da orientação a objetos é agrupar em uma entidade (denominada classe) os dados (atributos) e os procedimentos/funções (métodos) que realizam as operações sobre os dados. Assim, os dados podem permanecer isolados (ou encapsulados na classe) e o acesso a eles só pode ser realizado por meio dos métodos definidos na classe, ocasionando o chamado ocultamento de informação. Desse modo, em vez de estruturar o problema em termos de dados e procedimentos/funções, na orientação a objetos o desenvolvedor é incentivado a pensar em termos de classes e objetos que podem ser encarados como uma abstração de mais alto nível. Conceitualmente, deveria ser mais simples modelar um problema em termos de classes e objetos do que em termos de dados e funções. Além do encapsulamento, a orientação a objetos é também baseada em outras características, tais como herança, polimorfismo e acoplamento dinâmico.

Além disso, um forte apelo dos defensores da orientação a objetos é o fato de, durante as diferentes fases do ciclo de vida, serem utilizadas as mesmas definições e nomenclaturas, facilitando o mapeamento e a verificação de consistência entre os diferentes modelos.

Devido a essa característica, a orientação a objetos começou a se destacar como uma linguagem favorável ao desenvolvimento de soluções reusáveis e tornou-se a base para o desenvolvimento dos chamados componentes de software, que podem ser definidos como porções reutilizáveis de produtos de software.<sup>1</sup>

Sempre se menciona que o desenvolvimento de software utilizando programação Orientada a Objetos (OO) é mais fácil de entender, promove o reuso e torna a manutenção mais fácil em virtude do ocultamento de informação. Entretanto, o uso da orientação a objetos não é capaz de garantir o correto funcionamento de um programa por si próprio, visto que ela não impede que os desenvolvedores cometam enganos durante o desenvolvimento de um produto de software. Pelo contrário, o que se tem observado é que as características principais da orientação a objetos introduzem novas fontes de falhas de modo que atividades de Verificação, Validação e Teste (VV&T) são de fundamental importância no contexto de programas orientados a objetos.

Assim sendo, neste capítulo são apresentados alguns conceitos, terminologia e perspectivas relacionados ao teste de software orientado a objetos. Inicialmente, na Seção 6.2 é dada uma breve descrição das principais características da orientação a objetos. Em seguida, na Seção 6.3 são descritas as principais dificuldades relacionadas ao teste de programas orientados a objetos (POO). Na Seção 6.4 são caracterizadas as fases de teste para POO, contrapondo-as às fases de teste de programas procedimentais. Na Seção 6.5 as principais estratégias, técnicas e critérios de teste definidos para o teste de POO e de componentes são descritas. É dada ênfase aos critérios de teste baseados em fluxo de dados. Na Seção 6.6 são sintetizadas as características do teste de componentes de software, considerando as perspectivas do desenvolvedor e do usuário do componente. A importância da automatização da atividade de teste e as principais contribuições no contexto do teste de POO são discutidas na Seção 6.7. Na Seção 6.8 são feitas as considerações finais sobre este capítulo.

## 6.2 Definições e conceitos básicos

Independente da linguagem de programação utilizada no desenvolvimento de um produto de software, este pode ser liberado contendo defeitos não detectados durante a realização dos testes. Na programação estruturada, a ênfase é dada ao desenvolvimento de procedimentos, implementados em blocos estruturados, e à comunicação entre procedimentos por passagem de dados. Na programação OO, dados e procedimentos passam a fazer parte de um só elemento básico, o objeto. Esse paradigma de programação introduz uma abordagem na qual o programador visualiza seu programa em execução como uma coleção de objetos cooperantes que se comunicam por meio de troca de mensagens. Cada um dos objetos é uma instância de uma classe e todas as classes formam uma hierarquia de classes unidas via relacionamentos de herança. Esses e outros conceitos de orientação a objetos são descritos brevemente a seguir.

Uma classe é uma entidade estática que engloba atributos (ou dados) e métodos (ou funções-membro) que representam operações que podem ser realizadas sobre os dados. Um objeto é definido como sendo uma instância de uma classe criada em tempo de execução.

<sup>1</sup>Observe que está sendo utilizado o termo “produtos”, pois entende-se que o reuso não ocorre somente em termos de código-fonte, mas em termos dos diferentes artefatos (produtos) gerados ao longo do processo de desenvolvimento. É possível, por exemplo, reutilizar partes de um projeto orientado a objetos e não somente de código-fonte.

Cada objeto tem uma cópia dos dados existentes na classe e encapsula estado e comportamento.

Em vez da idéia procedural de entrada e saída para procedimentos, os objetos interagem entre si e são ativados por meio de mensagens. Uma mensagem é uma solicitação para que um objeto execute um de seus métodos. O método solicitado pode alterar o estado interno do objeto e/ou enviar mensagens a outros objetos. Ao encerrar sua execução, ele retorna o controle, e possivelmente algum valor, ao objeto que enviou a mensagem solicitando a operação.

A capacidade que um objeto tem de impedir que outros objetos tenham acesso aos seus dados é denominada de encapsulamento. O encapsulamento é uma técnica empregada para garantir o ocultamento de informações na qual a interface e a implementação de uma classe são separadas sintaticamente. Com isso, somente os métodos pertencentes a um objeto podem ter acesso aos dados encapsulados. O encapsulamento encoraja a modularidade do programa e permite que decisões de projeto fiquem “escondidas” dentro da implementação de maneira a restringir possíveis interdependências com outras classes, exceto por meio de sua interface. Com isso, mudanças na implementação de um método não afetam outras classes, a menos que a interface do método seja alterada.

Novas classes podem ser definidas em função de classes já existentes. Tal relacionamento entre classes é obtido por meio de herança. Com a utilização de herança, as classes são inseridas em uma hierarquia de especialização, de modo que uma classe mais especializada herde todas as propriedades da(s) classe(s) mais genérica(s), ou seja, daquela(s) que está(ão) níveis acima na hierarquia. À classe mais genérica dá-se o nome de superclasse, classe pai ou classe-base, e à classe mais especializada dá-se o nome de classe-filho ou subclasse. Com o mecanismo de herança, uma subclasse pode estender ou restringir as características herdadas da superclasse.

Algumas linguagens de programação OO, tais como C++, permitem que uma subclasse herde características de mais de uma superclasse, caracterizando o que é chamado de herança múltipla. Entretanto, tal mecanismo pode trazer problemas quando duas superclasses oferecem atributos ou métodos com o mesmo nome.

O termo polimorfismo representa a qualidade ou o estado de um objeto ser capaz de assumir diferentes formas. Quando aplicado a linguagens de programação, indica que uma mesma construção de linguagem pode assumir diferentes tipos ou manipular objetos de diferentes tipos. Por exemplo, o operador “+” na linguagem C++ pode ser utilizado para fazer a adição de dois valores inteiros ou ponto flutuante, bem como para concatenar duas *strings*.

Uma característica fortemente relacionada com herança e polimorfismo é o acoplamento dinâmico. Em programas procedimentais, sempre que uma nova funcionalidade deve ser acrescentada, a aplicação deve ser alterada e recompilada. Com o conceito de polimorfismo, é possível acrescentar novos métodos a classes já existentes sem a necessidade de recompilar a aplicação. Isto é possível utilizando-se a técnica de acoplamento dinâmico (*dynamic binding*), que permite que novos métodos sejam carregados e ligados (*binding*) à aplicação em tempo de execução.

Outro termo utilizado no contexto de orientação a objetos é o de *cluster*. *Cluster* pode ser definido como um conjunto de classes que cooperam entre si na implementação de determinada(s) funcionalidade(s). As classes dentro de um *cluster* podem ser fortemente acopladas

e trabalhar juntas para fornecer um comportamento unificado, ou podem ser independentes e fornecer diferentes tipos de funções similares [291].

Além dos *clusters*, outro termo que tem sido utilizado é o de componente de software. Embora possam existir componentes de rotinas em Cobol, por exemplo, foi com a orientação a objetos que o conceito de Desenvolvimento Baseado em Componentes (DBC) ganhou força e tem se tornado uma tendência de desenvolvimento. No contexto de componentes, freqüentemente são utilizados os conceitos de orientação a objetos. Um componente de software pode ser definido como uma unidade de composição com interfaces bem definidas e especificadas e dependências de contexto explícitas [381]. A idéia é que um componente de software pode ser desenvolvido independentemente e utilizado por terceiros na composição de um novo sistema.

Ocultamento de informação e encapsulamento são conceitos-chave em desenvolvimento baseado em componentes. Em geral, quando se utiliza um componente de software, o desenvolvedor só tem acesso à sua especificação e à interface de acesso que deve ser utilizada para a obtenção de determinada funcionalidade. O código de um componente, em geral, não é disponibilizado. Com isso, mantendo-se a mesma interface de acesso, a implementação de determinada funcionalidade pode ser alterada sem causar maiores problemas aos projetos que utilizam um componente. Mais informações sobre componentes de software podem ser obtidas na Seção 6.6.

Dada a natureza dinâmica de um objeto e seu comportamento dependente de estados, uma máquina de transição de estado ou outras formas de representação de sistemas reativos, tais como *Statecharts*, têm sido freqüentemente utilizadas para modelar o comportamento dinâmico dos objetos. Por exemplo, o diagrama de estado, utilizado na UML [303], representa a seqüência de estados pela qual um objeto passa durante a sua vida, em resposta a um estímulo recebido, junto com suas respostas e ações. A semântica e a notação desse diagrama são substancialmente as mesmas dos *Statecharts* [162]. A partir desses modelos, casos de teste podem ser derivados para garantir o correto funcionamento dos objetos (veja o Capítulo 3 para saber mais detalhes sobre a geração de casos de testes a partir de máquinas de estado).

### 6.3 Tipos de defeitos em POO

O paradigma de programação OO possui um conjunto de construções poderosas, que apresentam “risco de erros” (*fault hazard*) e problemas de teste. Isso é um resultado inevitável do encapsulamento de métodos e atributos dentro de uma classe, da variedade de modos como um subsistema pode ser composto e da possibilidade de, em poucas linhas de código, dar um comportamento ao sistema que só será definido em tempo de execução (acoplamento dinâmico). Cada nível em uma hierarquia de herança dá um novo contexto para as características herdadas; o comportamento correto nos níveis mais elevados não é garantido nos níveis mais baixos.

Por exemplo, considere uma classe composta por herança múltipla, com seis superclasses, todas contribuindo na hierarquia de herança e muitos métodos polimórficos. O desenvolvedor deverá gastar tempo e esforço consideráveis para garantir que todos os métodos das superclasses funcionem adequadamente na subclasse e que não exista nenhuma interação indesejável entre os métodos. Além disso, polimorfismo e acoplamento dinâmico aumentam

dramaticamente o número de caminhos que devem ser testados, e o encapsulamento pode criar obstáculos que limitam a visibilidade do estado dos objetos.

A questão da reusabilidade também traz novos desafios para a atividade de teste. Componentes de software disponibilizados para reuso devem ser altamente confiáveis e um teste mais rigoroso é exigido quando se deseja obter um reuso efetivo. Entretanto, nem o reuso nem a composição de componentes testados previamente irão eliminar a necessidade do reteste, considerando o novo contexto no qual esses componentes serão reutilizados [168, 166].

Embora a programação OO possa reduzir a ocorrência de alguns tipos de defeitos cometidos na programação procedural, ela aumenta a chance de ocorrência de outros. Tendo em vista que, em geral, métodos contêm somente algumas poucas linhas de código, defeitos de fluxo de controle são menos prováveis de ocorrer. O encapsulamento previne defeitos resultantes do acesso a dados armazenados em variáveis globais, que podem vir a ocorrer em linguagens procedimentais. Entretanto, não há motivos para supor que programadores que utilizam programação OO estão menos propensos a cometer enganos. Enganos de codificação referentes à sintaxe ou à grafia incorreta são tão comuns quanto antes. Além disso, algumas características das linguagens OO trazem novos riscos de defeitos [35]. Defeitos na programação de interface são comuns em programas procedimentais. Programas orientados a objetos têm, tipicamente, muitos métodos e, consequentemente, muitas interfaces, aumentando a ocorrência desses tipos de defeitos. Esses e outros problemas que podem ocorrer em programas orientados a objetos são discutidos a seguir.

### 6.3.1 Efeitos colaterais da programação OO

#### Encapsulamento

Como definido anteriormente, o encapsulamento refere-se ao mecanismo de controle de acesso que determina a visibilidade de atributos e métodos dentro de uma classe. Com o controle de acesso previnem-se dependências indesejadas entre uma classe cliente e uma classe servidora, por exemplo, tornando visível ao cliente somente a interface da classe, ocultando detalhes de implementação. O encapsulamento auxilia no ocultamento de informação e na obtenção da modularidade do sistema que está sendo desenvolvido.

Embora o encapsulamento não contribua diretamente para a ocorrência de defeitos, ele pode apresentar-se como um obstáculo para a atividade de teste. O teste requer um relatório completo do estado concreto e abstrato de um objeto, bem como a possibilidade de alterar esse estado facilmente [35]. As linguagens OO dificultam a atividade de se obter (*get*) ou atualizar (*set*) o estado de um objeto. No caso específico de C++, as funções amigas (*friend functions*) podem ser utilizadas para solucionar esse problema. Entretanto, no caso de linguagens que não possuem esse recurso, outras providências devem ser tomadas. Harrold [166], comentando a respeito do teste de componentes de software, diz que uma solução seria a implementação de métodos *get* e *set* para todos os atributos de uma classe. Outra alternativa seria utilizar recursos de reflexão computacional, embora, como destacam Rosa e Martins [346], algumas linguagens não permitam que as características de métodos privados sejam refletidas, somente a de métodos protegidos e públicos. Esse é o caso, por exemplo, da linguagem OpenC++ 1.2, utilizada pelas autoras [346].

## Herança

A herança é essencial à programação OO, pois permite a reusabilidade via o compartilhamento de características presentes em uma classe definida anteriormente. Entretanto, como destacado por Binder [35], a herança enfraquece o encapsulamento e pode ser responsável pela criação de um risco de defeito similar ao uso de variáveis globais em programas procedimentais. Quando se está implementando uma classe que faz uso de herança, é de fundamental importância compreender os detalhes de implementação das classes ancestrais. Sem tomar esse cuidado, é possível o desenvolvimento de classes que aparentemente funcionam corretamente mas violam condições implícitas requeridas para garantir o correto funcionamento das classes ancestrais. Grandes hierarquias de herança podem dificultar a compreensão, aumentar a chance de ocorrência de enganos e reduzir a testabilidade das classes.

## Herança múltipla

A herança múltipla permite que uma subclasse herde características de duas ou mais superclasses, as quais podem conter características comuns (atributos com mesmo nome e métodos com mesmo nome e mesmo conjunto de atributos). Perry e Kaiser destacam que, embora a herança múltipla leve a pequenas mudanças sintáticas, ela pode levar a grandes mudanças semânticas [327]. Alguns dos riscos de defeitos causados por herança múltipla são [35]:

- Mudanças em qualquer uma das superclasses podem resultar em interações indesejadas na subclasse. Por exemplo, suponha que  $Z$  seja uma subclasse das classes  $X$  e  $Y$  e que ambas as superclasses tenham um método  $m$ , conforme ilustrado na Figura 6.1. Originalmente,  $Z$  utilizava o método  $m()$  da classe  $X$  ( $X.m()$ ) mas foi alterada para passar a utilizar  $Y.m()$ . Com isso,  $Z$  deve ser retestada e é pouco provável que o conjunto de teste utilizado para testar  $Y.m()$  seja adequado para testar  $Z.m()$ , pois quando o método  $m()$  é utilizado no contexto da classe  $Z$ , este pode interagir com outros métodos e ser executado com outros valores que não foram previstos anteriormente.
- Herança repetida ocorre quando uma superclasse aparece mais de uma vez na hierarquia de herança. O teste na presença de herança repetida é ainda mais complicado que no caso de herança múltipla, uma vez que existe um número de características que são renomeadas ou removidas e a possibilidade de defeitos aumenta. Por exemplo, suponha a existência de classes  $B$  e  $C$  derivadas da superclasse  $A$  e uma classe  $D$  derivada de  $B$  e  $C$ , conforme ilustrado na Figura 6.2. Essa estratégia pode levar à ocorrência de defeitos. Se métodos e atributos não forem explicitamente qualificados pelo nome da classe, pode ocorrer um conflito de nomes que resulte no comportamento inesperado de métodos virtuais.<sup>2</sup>

Assim sendo, herança pública e privada, classes abstratas *versus* classes concretas e a visibilidade dos dados membros da superclasse constituem os riscos de defeitos associados com a herança múltipla.

<sup>2</sup>Em C++, métodos virtuais são aqueles declarados na superclasse que precisam ser redefinidos nas subclasses. Para ter acesso aos objetos de diferentes classes usando a mesma instrução, os métodos da superclasse que serão reescritos nas subclasses devem ser declarados virtuais.

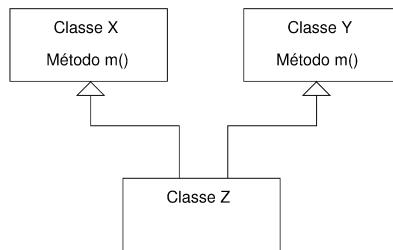


Figura 6.1 – Exemplo de herança múltipla (adaptado de Binder [35]).

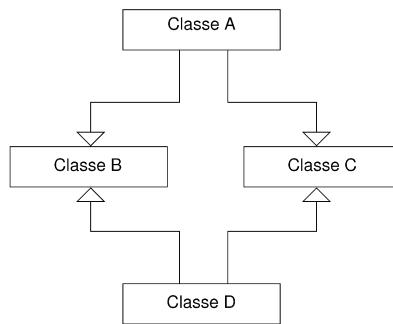


Figura 6.2 – Exemplo de herança (adaptado de Binder [35]).

## Classes abstratas e genéricas

Uma classe abstrata é a que fornece somente uma interface sem nenhuma implementação, fornecendo um importante suporte para o reuso [35]. O teste de uma classe abstrata só poderá ser realizado após esta ter sido especializada e uma classe concreta ter sido obtida. Não é possível criar objetos ou instâncias de classes abstratas. Esse processo pode ser complicado se um método concreto utiliza um método abstrato para implementar sua funcionalidade.

Classes genéricas, por sua vez, não são necessariamente abstratas, mas também fornecem um importante suporte para o reuso, uma vez que são a base para a ocorrência do acoplamento dinâmico. Uma classe genérica permite que sejam declarados atributos e parâmetros que podem ser instanciados com objetos de tipos específicos. Por exemplo, suponha uma classe genérica A a partir da qual foram derivadas as classes B e C. Suponha também a existência de uma classe D derivada a partir de B. Declarando-se um atributo ou parâmetro do tipo da classe genérica A, esse atributo ou parâmetro pode ser instaciado por qualquer um dos objetos criados a partir das classes B, C ou D. Segundo Smith e Robson [365], os principais problemas no teste de classes genéricas são:

- Classes genéricas precisam ter o parâmetro genérico substituído por um parâmetro de um tipo específico para serem testadas. Com isso, surge o problema de escolher que tipo utilizar para o teste. Se for escolhido um tipo simples, talvez a classe genérica

não seja testada corretamente. O uso de algum mecanismo de restrições que impeça que tipos inadequados sejam utilizados pode ser utilizado para reduzir a ocorrência de defeitos.

- Da mesma forma que superclasses abstratas, as classes derivadas da classe genérica devem ser retestadas se mudanças são feitas nas classes genéricas.

## Polimorfismo

Polimorfismo é caracterizado pela possibilidade de um único nome (denominado nome polimórfico), ou seja, uma variável ou uma referência, denotar instâncias (objetos) de várias classes que, usualmente, devem possuir uma única superclasse em comum. Considerando um esquema de herança de subtipo, todos os objetos denotados possuem, pelo menos, as propriedades da superclasse raiz da hierarquia. Assim sendo, um objeto pertencente a qualquer uma das subclasses poderia ser substituído em um contexto em que é requerida uma instância da superclasse, sem causar nenhum erro de tipo em execuções subsequentes do código. Observe que essa hierarquia de classes pode ser tão vasta quanto a totalidade das classes do sistema no caso de todas as classes derivarem de uma única superclasse comum, como ocorre em linguagens como Smalltalk, Eiffel e Java [22].

A seguir são descritos dois problemas, relatados por Barbey e Strohmeier [22], nos quais o uso do polimorfismo pode afetar o correto funcionamento de um programa e causar problemas para o teste:

- Indecidibilidade no acoplamento dinâmico

O polimorfismo traz indecidibilidade para o teste baseado em programa. Uma vez que nomes polimórficos podem denotar objetos de diferentes classes, é impossível, ao invocar um método por meio de um nome polimórfico, predizer em tempo de compilação qual trecho de código será executado, ou seja, se será executado o método original da superclasse ou o método refinado de uma possível subclasse. Tal decisão só ocorre em tempo de execução.

Embora o polimorfismo possa ser utilizado para produzir código elegante e extensível, alguns aspectos problemáticos podem ser detectados na sua utilização. Suponha a existência de um método  $x()$  em uma superclasse que precisa ser testado. Posteriormente, o método  $x()$  é sobreescrito. O correto funcionamento do método  $x$  na subclasse não é garantido, pois as precondições e pós-condições na subclasse para a execução do método  $x$  podem não ser as mesmas da superclasse [35].

Cada possibilidade de acoplamento de uma mensagem polimórfica é uma computação única. O fato de diversos acoplamentos polimórficos funcionarem corretamente não garante que todos o farão. Objetos polimórficos com acoplamento dinâmico podem facilmente resultar no envio de mensagens para a classe errada e pode ser difícil identificar e executar todas as combinações de acoplamento permitidas, o que cria um risco de erro.

- Extensibilidade de hierarquias

Existe um problema quando se deseja testar um ou mais métodos com parâmetros polimórficos. O teste de um método consiste em verificar seu comportamento quando

executado com diversos valores de parâmetros diferentes. Um bom conjunto de teste deveria garantir que todos os possíveis acoplamentos polimórficos sejam exercitados ao menos uma vez.

Entretanto, dada uma chamada polimórfica ou uma chamada a um método com um ou mais parâmetros polimórficos, é impossível desenvolver um conjunto de teste que garanta a execução do acoplamento em relação a todas as possíveis classes, uma vez que a hierarquia de classes é livremente extensível.

Para resolver esse problema e o anterior, Barbey e Strohmeier [22] sugerem o uso de assertivas (*assertions*). As pré e pós-condições necessárias para executar cada método são especificadas em termos de assertivas. Toda vez que um método é sobreescrito/refinado, as assertivas são também refinadas, mas não podem ser relaxadas. Dessa forma, fica mais fácil identificar quando objetos polimórficos estão executando métodos de classes indevidas ou não esperadas.

### Containers heterogêneos e *type casting*

Containers heterogêneos são estruturas de dados que armazenam componentes que podem pertencer a várias classes diferentes, da mesma forma que nomes polimórficos.

Entretanto, alguns dos objetos armazenados podem não ter o mesmo conjunto de operações daquelas pertencentes à superclasse raiz de uma hierarquia. Para permitir que tais objetos façam uso de todos os seus métodos, é possível fazer a conversão (*casting*) do objeto contido na estrutura de dados heterogênea para qualquer classe da hierarquia. Isso pode levar a dois tipos comuns de falha:

1. um objeto pode estar sendo convertido para uma classe à qual ele não pertence, sendo, desse modo, incapaz de selecionar uma determinada característica ou invocar um método da classe em questão. Esse problema é conhecido como *downcasting*;
2. considerando um esquema de herança de subclasse, um objeto não convertido para o tipo de sua classe pode estar sendo usado para invocar um método que foi removido de sua classe, resultando em um erro em tempo de execução caso o método não exista em nenhuma outra classe da hierarquia, ou na invocação de um método indesejado.

Esses dois tipos de defeito, em geral, não são detectados durante a fase de compilação. Assim, um cuidado especial deve ser tomado durante o desenvolvimento do conjunto de teste para evitar que tais defeitos passem despercebidos.

### Outros problemas

Além dos problemas apresentados, Binder [35] descreve ainda defeitos relacionados a seqüências de mensagens e estados dos objetos. O “empacotamento” de métodos dentro de uma classe é fundamental na programação OO; como resultado, mensagens devem ser executadas em alguma seqüência, originando a questão: “Quais seqüências de envio de mensagens são corretas?”

Objetos são entidades criadas em tempo de execução, cada um podendo conter o próprio conjunto de atributos em memória, caracterizando seu estado. Cada nova configuração assumida por esse espaço de memória caracteriza um novo estado do objeto. Assim sendo, além do comportamento encapsulado por um objeto por meio de seus métodos e atributos, objetos também encapsulam estados.

Segundo McDaniel e McGregor [277] existem duas definições de estado: normal e baseada em projeto. A definição normal de estado refere-se a todas as possíveis combinações de valores que os atributos de um objeto podem receber, caracterizando, em geral, um conjunto de estados infinito. Por outro lado, estados baseados em projeto referem-se ao conjunto de valores dos atributos que permitem claramente diferenciar e determinar o comportamento do objeto que está sendo observado. Por exemplo, para ilustrar ambos os conceitos, considere uma pilha de  $n$  elementos. Na visão tradicional, que considera um estado em separado para cada atributo de dado, a pilha deveria ter  $n$  estados. Portanto, para se testar  $n$  possíveis estados seriam necessários  $n$  casos de teste. Considerando agora uma representação de estados baseada em projeto, seria suficiente considerar os estados de pilha cheia, pilha vazia e, por exemplo, o estado pilha com mais de um elemento e não cheia. Com isso, o número de estados para representar uma pilha seria reduzido de  $n$  para três [277].

Ao examinar como a execução de um método pode alterar o estado de um objeto, quatro possibilidades são observadas [277]: 1) ele pode levar o objeto a um novo estado válido; 2) ele pode deixar o objeto no mesmo estado em que se encontra; 3) ele pode levar o objeto para um estado indefinido; ou 4) ele pode alterar o estado para um estado não apropriado.

As opções 3 e 4 caracterizam estados de erro. A opção 2 pode caracterizar um erro se o método executado devia ter se comportado como na opção 1. A opção 1 também pode caracterizar um erro se a execução do método devia ter o comportamento da opção 2.

## 6.4 Fases de teste OO

Além da utilização de técnicas e critérios de teste, quando grandes programas são testados, é necessário dividir a atividade de teste em várias fases. Com isso, o testador pode se concentrar em aspectos diferentes do software e em diferentes tipos de defeitos, além de utilizar diferentes estratégias de seleção de dados de teste e medidas de cobertura em cada uma delas [244].

Como descrito na literatura e comentado no Capítulo 1, a atividade de teste pode ser considerada uma atividade incremental realizada em três fases: teste de unidade, teste de integração e teste de sistema [329]. Variações são identificadas no contexto de software orientado a objetos, conforme discutido mais adiante nesta seção.

Inicialmente, os testes de unidade focalizam cada unidade a fim de garantir que os aspectos de implementação de cada uma estejam corretos. O objetivo é identificar defeitos de lógica e de implementação em cada unidade do software. Durante esta fase, com freqüência é utilizada a técnica de teste estrutural, que requer a execução de elementos específicos da estrutura de controle de cada unidade, com o objetivo de garantir uma completa cobertura e a máxima detecção de defeitos. O teste de mutação também tem sido aplicado nesta fase.

Após cada unidade ter sido testada, inicia-se a fase de integração e, consequentemente, o teste de integração. Todavia, por que um programa construído a partir de unidades que in-

dividualmente trabalham de maneira correta – todas foram submetidas ao teste de unidade – não trabalharia corretamente? A resposta é que o teste de unidade apresenta limitações e não pode garantir que cada unidade trabalhe adequadamente em todas as situações: por exemplo, uma unidade pode sofrer uma influência adversa, não prevista, de outra unidade; subfunções, quando combinadas, podem produzir resultados inesperados e estruturas de dados globais podem apresentar problemas [108]. Além disso, deve-se ressaltar que os tipos de erro em geral revelados com o teste de integração elevam em muito o custo da atividade de teste se forem detectados nos estágios mais avançados, em especial se a correção do erro forçar modificações em unidades previamente testadas. Desse modo, a realização de testes de integração é de fundamental importância para assegurar uma melhor qualidade do software que está sendo construído e reduzir os custos associados. Segundo Pressman [329], as técnicas de projeto de casos de teste funcional são as mais utilizadas durante esta fase. Iniciativas de extensões de critérios utilizados no teste de unidade para o teste de integração são identificadas na literatura, tais como a extensão de critérios baseados em fluxo de dados e de critérios baseados em mutação [110, 160, 171, 200, 244, 411, 420].

Como destacado por Jorgensen e Erickson [204] e por Binder [35], o software orientado a objetos é composto de classes as quais encapsulam uma série de métodos, em geral de baixa complexidade, que cooperam entre si na implementação de dada funcionalidade. Com isso, devido ao grande número de métodos e de conexões existentes, mais ênfase deve ser dada ao teste de integração.

Depois que o software foi integrado e funciona como um todo, são realizados os testes de sistema. O objetivo é assegurar que o software e os demais elementos que compõem o sistema (por exemplo, hardware e banco de dados) combinem-se adequadamente e que a função/desempenho global desejada seja obtida. A técnica de teste funcional é a que tem sido mais utilizada nesta fase de teste [329].

A Figura 6.3, adaptada de Binder [35], ilustra as três fases de teste mencionadas, bem como os componentes utilizados em cada uma das fases tanto para programas procedimentais como para POO.

Segundo o padrão IEEE 610.12-1990 [196], uma unidade é um componente de software que não pode ser subdividido. Assim sendo, considerando que teste é uma atividade dinâmica, em programas procedimentais, uma unidade  $F$  refere-se a uma sub-rotina ou a um procedimento, que é a menor parte funcional de um programa possível de ser executada. Observa-se que durante os testes de unidade é necessária a implementação de *drivers* e *stubs*. O *driver* é uma unidade que coordena o teste de  $F$ , sendo responsável por ler os dados de teste fornecidos pelo testador, repassar esses dados na forma de parâmetros para  $F$ , coletar os resultados relevantes produzidos por  $F$  e apresentá-los para o testador. Um *stub* é uma unidade que substitui, na hora do teste, uma unidade usada (chamada) por  $F$ . Na maior parte dos casos, um *stub* é uma unidade que simula o comportamento da unidade chamada por  $F$  com o mínimo de computação ou manipulação de dados.

Dadas essas definições, pode-se considerar que, em POO, a menor unidade a ser testada é um método, sendo que a classe à qual o método pertence pode ser vista como o *driver* do método. Sem a existência da classe, não é possível executar um método. No paradigma procedural, o teste de unidade também é chamado de intraprocedimental, e no paradigma de programação OO, intramétodo [169].

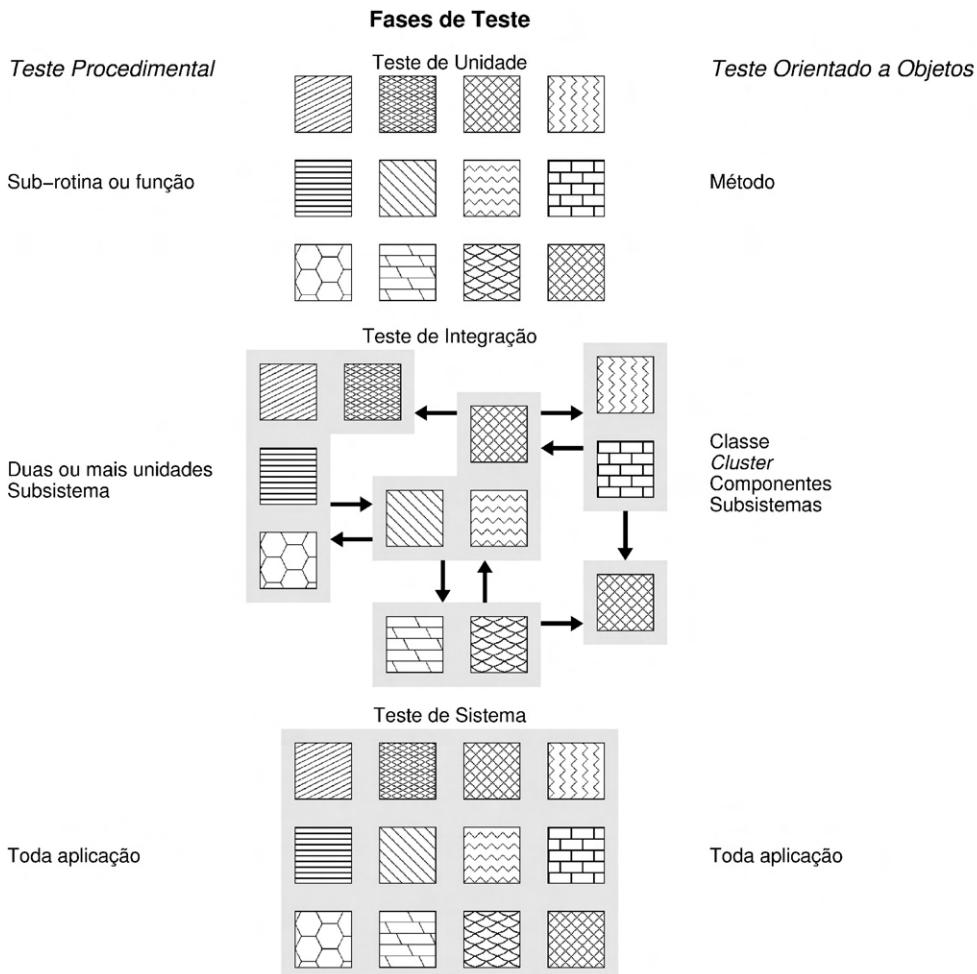


Figura 6.3 – Relacionamento entre teste de unidade, de integração e de sistema: programas procedimentais e OO (adaptada de Binder [35]).

Por definição, uma classe engloba um conjunto de atributos e métodos que manipulam esses atributos. Assim, considerando uma única classe, já é possível pensar-se em teste de integração. Métodos da mesma classe podem interagir entre si para desempenhar funções específicas que caracterizam uma integração entre métodos que deve ser testada: teste intermétodo [169]. No paradigma procedural esta fase de teste também pode ser chamada de teste interprocedimental.

Harrold e Rothermel [169] definem ainda outros dois tipos de teste para POO: teste intraclasse e teste interclasse. No teste intraclasse são testadas interações entre métodos públicos fazendo-se chamadas a esses métodos em diferentes sequências. O objetivo é identificar possíveis sequências de ativação de métodos inválidas que levem o objeto a um estado inconsistente. Segundo os autores, como o usuário pode invocar sequências de métodos públi-

cos em uma ordem indeterminada, o teste intraclassé aumenta a confiança de que diferentes seqüências de chamadas interagem adequadamente. No teste interclassé o mesmo conceito de invocação de métodos públicos em diferentes seqüências é utilizado; entretanto, esses métodos públicos não necessitam estar na mesma classe.

Finalmente, após realizados os testes acima, o sistema todo é integrado e podem ser realizados os testes de sistema que, por se basear em critérios funcionais, não apresentam diferenças fundamentais entre o teste procedural e orientado a objetos.

Pequenas variações quanto à divisão das fases de teste para POO são identificadas na literatura. Por exemplo, Colanzi [88] caracteriza a fase do teste de classe, cuja finalidade é descobrir defeitos de integração entre os métodos dentro do escopo da classe em teste, e a fase do teste de integração para software orientado a objetos, que tem o objetivo de encontrar defeitos na integração de classes do sistema. Assim, segundo Colanzi [88], o teste de POO é organizado em quatro fases:

- teste de unidade: testa os métodos individualmente;
- teste de classe: testa a interação entre métodos de uma classe;
- teste de integração: testa a interação entre classes do sistema; e
- teste de sistema: testa a funcionalidade do sistema como um todo.

Considerando-se o método como a menor unidade, o teste de classe, proposto por Colanzi [88], pode ser visto como parte do teste de integração, juntamente com o teste intraclassé e interclassé.

Alguns autores entendem que a classe é a menor unidade no paradigma de programação OO [17, 35, 277, 279, 327]. Nessa direção o teste de unidade poderia envolver o teste intramétodo, intermétodo e intraclassé, e o teste de integração corresponderia ao teste interclassé. Nesse contexto, o teste de classe proposto por Colanzi seria enquadrado como teste de unidade.

Na Tabela 6.1 são sintetizados os tipos de teste que podem ser aplicados em cada uma das fases tanto em programas procedimentais quanto em POO, considerando o método ou a classe a menor unidade.

## 6.5 Estratégias, técnicas e critérios de teste OO

Na literatura podem ser encontrados trabalhos que buscam assegurar que a atividade de teste seja aplicada em todas as fases de desenvolvimento do software [35, 88, 279]: da fase de análise até a fase de implementação.

Howden [192] define que o teste pode ser classificado de duas maneiras: teste baseado em especificação e teste baseado em programa, dependendo em qual nível os critérios de teste são aplicados. No contexto de programação OO encontra-se uma série de trabalhos relacionados com o teste baseado em especificação [35, 62, 67, 184, 185, 228, 278, 277, 308, 399]. Basicamente, esses trabalhos utilizam diagramas que modelam o comportamento dos objetos para derivar os requisitos de teste. Com relação ao teste baseado em programa,

Tabela 6.1 – Relação entre fases de teste de programas procedimentais e OO

| Menor Unidade: Método |                     |  |
|-----------------------|---------------------|--|
| Fase                  | Teste Procedimental | Teste Orientado a Objetos              |
| Unidade               | Intraprocedimental  | Intramétodo                            |
| Integração            | Interprocedimental  | Intermétodo, Intraclasse e Interclasse |
| Sistema               | Toda Aplicação      | Toda Aplicação                         |
| Menor Unidade: Classe |                     |  |
| Fase                  | Teste Procedimental | Teste Orientado a Objetos              |
| Unidade               | Intraprocedimental  | Intramétodo, Intermétodo e Intraclasse |
| Integração            | Interprocedimental  | Interclasse                            |
| Sistema               | Toda Aplicação      | Toda Aplicação                         |

embora em menor número, algumas iniciativas também são encontradas [34, 35, 67, 69, 169, 198, 213, 257, 364, 370, 420].

A seguir é dada uma descrição mais detalhada a respeito de alguns desses trabalhos, descrevendo e exemplificando critérios e estratégias de teste desenvolvidos para o teste de POO. Inicialmente são descritos brevemente os critérios funcionais e posteriormente são descritos os critérios estruturais que podem ser utilizados nesse contexto. A estratégia de teste incremental hierárquica também é descrita por enfatizar o reuso de casos de testes que pode ser obtido por meio do uso de herança.

## Teste baseado em especificação

Como mencionado antes, devido ao caráter dinâmico dos objetos e de sua capacidade de armazenar estado interno, a modelagem do comportamento dos objetos pode ser feita por meio de um diagrama de estados. Nesse sentido, as técnicas e os critérios de teste identificados no Capítulo 3 são diretamente aplicáveis nesse contexto. O mesmo ocorre com os critérios de teste funcionais, descritos no Capítulo 2, que, por serem independentes do paradigma de programação, também são diretamente aplicáveis no teste de POO.

No contexto de teste de especificação de POO, o que tem também sido investigado é o desenvolvimento de critérios de teste que utilizam diferentes tipos de diagramas utilizados no projeto orientado a objetos para auxiliar na geração de casos de testes. Por exemplo, Chaim et al. [62] definiram uma série de critérios de teste para realizar teste de cobertura em especificações UML, mais especificamente em diagramas de casos de uso. Os critérios de teste propostos por Chaim et al. almejam assegurar que casos de testes garantam a cobertura dos casos de usos utilizados para modelar sistemas orientados a objetos. Uma das idéias básicas dos critérios é assegurar que as interações entre casos de usos e entre atores e casos de usos sejam exercitadas, contribuindo para uma especificação mais precisa do problema que está sendo modelado. Além do trabalho de Chaim et al. [62], outros trabalhos nessa mesma linha podem ser citados, como os de Abdurazik e Offutt [1], Beckman e Grupta [29], Heumann [181], Offutt e Abdurazik [305], entre outros.

## Teste baseado em programa

Um problema com os critérios de teste baseados em especificação é a dificuldade de quantificar a atividade de teste, visto que não se pode garantir que partes essenciais ou críticas do programa sejam executadas. Outro problema é que o teste baseado em especificação está sujeito às inconsistências decorrentes de uma especificação de má qualidade, pois é ela a base a partir da qual são derivados os casos de teste. Como, em geral, a especificação é feita de forma descriptiva e informal, os requisitos derivados da especificação também são, de certa maneira, descriptivos e informais, dificultando a automatização dos critérios funcionais.

Entre os critérios de teste baseados em programa, destacam-se os critérios estruturais de fluxo de controle e de dados e os critérios baseados em mutação.

### 6.5.1 Critérios estruturais

A técnica de teste estrutural também apresenta uma série de limitações e desvantagens decorrentes das limitações inerentes às atividades de teste de programa, como estratégia de validação, tais como a determinação de caminhos e associações não executáveis [141, 191, 261, 302, 337]. Esses aspectos introduzem sérias limitações na automatização do processo de validação de software [261]. Independente dessas desvantagens, essa técnica é vista como complementar à técnica funcional [329], e informações obtidas pela aplicação desses critérios têm sido consideradas relevantes para as atividades de manutenção, depuração e confiabilidade de software [172, 166, 320, 329, 402, 403].

A definição desses critérios foi feita originalmente para o teste de programas procedimentais, mas vem sendo estendida ao longo dos anos para se adequar ao teste de POO.

Harrold e Rothermel [169] estenderam o teste de fluxo de dados para o teste de classes. Os autores comentam que os critérios de fluxo de dados destinados ao teste de programas procedimentais [144, 170, 337] podem ser utilizados tanto para o teste de métodos individuais quanto para o teste de métodos que interagem entre si dentro de uma mesma classe. Entretanto, esses critérios não consideram interações de fluxo de dados quando os usuários de uma classe invocam seqüência de métodos em uma ordem arbitrária.

Para viabilizar o teste de fluxo de dados nos níveis intramétodo, intermétodo e intraclasse, Harrold e Rothermel [169] propuseram as seguintes representações de programa: Grafo de Chamadas de Classe (*class call graph*), Grafo de Fluxo de Controle de Classe (*CCFG - class control flow graph*) e o CCFG encapsulado (*framed CCFG*). Com base nessas representações, os três níveis de teste foram considerados:

- teste intramétodo – testa os métodos individualmente. Esse nível é equivalente ao teste de unidade de programas procedimentais;
- teste intermétodo – testa os métodos públicos em conjunto com outros métodos dentro de uma mesma classe. Esse nível de teste é equivalente ao teste de integração de programas procedimentais;
- teste intraclasse – testa a interação entre métodos públicos quando eles são chamados em diferentes seqüências. Como os usuários de uma classe podem invocar seqüências de métodos em uma ordem indeterminada, o teste intraclasse serve para aumentar a

confiança de que essas diferentes seqüências de invocação não colocam a classe em um estado inconsistente. Entretanto, os autores destacam que, como o conjunto de todas as possíveis seqüências de invocação é infinito, somente um subconjunto dessas seqüências pode ser testado.

Com base nesses níveis de teste, Harrold e Rothermel [169] definiram pares definição e uso que permitem avaliar relações de fluxo de dados em POO. Seja  $C$  uma classe em teste. Se  $d$  representa um comando contendo uma definição e  $u$  um comando contendo um uso de uma variável, seguem-se estas definições:

- 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 que são 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  $M_i \neq M_j$  e  $M_i$  e  $M_j$  são invocações separadas do mesmo método, então  $(d, u)$  é um par def-uso intermétodo;
- pares def-uso intraclasse – seja  $M_0$  um método público de  $C$  e seja  $\{M_1, M_2, \dots, M_n\}$  o conjunto de métodos que são 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 que são chamados, direta ou indiretamente, quando  $N_0$  é invocado. Suponha que  $d$  está em alguns dos métodos em  $\{M_1, M_2, \dots, M_n\}$  e  $u$  em alguns dos métodos em  $\{N_1, N_2, \dots, N_n\}$ . Se existe um programa  $P$  que chama  $M_0$  e  $N_0$ , tal que  $(d, u)$  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 intraclasse.

Para ilustrar as definições apresentadas, considere a classe `TabelaSimbolo`, adaptada de Harrold e Rothermel [169], cuja definição aparece em Programa 6.1 e a implementação parcial dos métodos em Programa 6.2.

---

Programa 6.1

---

```

1 class TabelaSimbolo {
2     private:
3         TabelaEntrada *tabela;
4         int numeroentradas, maximoentradas;
5         int *Procurar(char *, int);
6     public:
7         TabelaSimbolo(int n) {
8             maximoentradas = n;
9             numeroentradas = 0;
10            tabela = new TabelaEntrada[maximoentradas]; };
11        TabelaSimbolo() { delete tabela; };
12        int AdicionanaTabela(char *simbolo, char *info);
13        int ObterdaTabela(char *simbolo, char *info);
14    };

```

---

---

Programa 6.2

---

```
1 #include "simbolo.h"
2
3 int TabelaSimbolo::Procurar(char *chave, int indice) {
4     int guardaindice;
5     int Hash(char *);
6     guardaindice = indice = Hash(chave);
7     while (strcmp(ObterSimbolo(indice),chave) != 0) {
8         indice++;
9         if (indice == maximoentradas) /* wrap around */
10            indice = 0;
11         if (ObterSimbolo(indice)==0 || indice==guardaindice)
12             return NAOACHOU;
13     }
14     return ACHOU;
15 }
16
17 int TabelaSimbolo::AdicionanaTabela(char *simbolo, char *info ) {
18     int indice;
19     if (numeroentradas < maximoentradas) {
20         if (Procurar(simbolo,indice) == ACHOU)
21             return NAOOK;
22         AdicionaSimbolo(simbolo,indice);
23         AdicionaInfo(info,indice);
24         numeroentradas++;
25         return OK;
26     }
27     return NAOOK;
28 }
29
30 int TabelaSimbolo::ObterdaTabela(char *simbolo, char **info) {
31     int indice;
32     if (Procurar(simbolo,indice) == NAOACHOU)
33         return NAOOK;
34     *info = GetInfo(indice);
35     return OK;
36 }
37
38 void TabelaSimbolo::AdicionaInfo(info,indice)
39     ...
40     strcpy(tabela[indice].info,info);
41 }
42
43 char *TabelaSimbolo::GetInfo(indice)
44     ...
45     return tabela[indice].info;
46 }
```

---

Informalmente, o teste intramétodo permite testar pares def-uso dentro de um único método. Por exemplo, considerando a classe `TabelaSimbolo` (Programa 6.2), o método `Procurar` contém um par def-uso intramétodo em relação à variável `indice`, pois a definição da variável `indice` na linha 8 tem um uso na linha 9.

Pares def-uso intermétodo ocorrem quando métodos dentro do contexto de uma invocação interagem e a definição de uma variável dentro dos limites de um método alcança um uso dentro dos limites de outro método chamado direta ou indiretamente por um método público. Considerando a classe `TabelaSimbolo`, o método público `AdicionanaTabela` invoca o método `Procurar`, o qual define a variável `indice`, que, posteriormente, é utilizada

para chamar o método `AdicionaSímbolo`. Logo, existe um par def-uso intermétodo entre a variável `indice` definida na linha 10 de `Procurar` e o uso de `indice` na linha 22 de `AdicionanaTabela`.

Finalmente, pares def-uso intraclasse ocorrem quando seqüências de métodos públicos são invocadas. Por exemplo, considere a seqüência de chamadas `(AdicionanaTabela, AdicionanaTabela)`. Na primeira chamada a `AdicionanaTabela`, se um símbolo é adicionado à tabela, a variável `numeroentradas` (linha 24) é incrementada (redefinida). Na segunda chamada a `AdicionanaTabela` a variável `numeroentradas`, definida anteriormente, é utilizada na condição da linha 19. Então, é definido um par def-uso intraclasse da variável `numeroentradas` definida na linha 24 e depois utilizada na linha 19.

Assim sendo, o CCFG é o modelo base para representar o fluxo de dados de uma classe, facilitando a geração de requisitos de teste intramétodo, intermétodo e intraclasse. A descrição detalhada do algoritmo utilizado para a construção do Grafo de Fluxo de Controle de Classe pode ser encontrada no trabalho de Harrold e Rothermel [169]. Entretanto, para a identificação dos requisitos de teste interclasse, é necessário o desenvolvimento de outra representação, que expresse o tipo de interação a ser considerada, o que não foi levantado pelo trabalho desses pesquisadores quando proposto.

Algumas das limitações da técnica proposta por Harrold e Rothermel [169] incluem: 1) não identificar alguns pares definição-uso intramétodo, intermétodo e intraclasse resultantes de apelidos (*aliases*) específicos; e 2) não manipular características específicas da programação OO, tais como polimorfismo e acoplamento dinâmico. Posteriormente, Rothermel et al. [349] propuseram uma extensão na construção do CCFG incluindo um nó polimórfico específico nos pontos de chamada que conecta o conjunto de métodos polimórficos que poderiam ser invocados daquele ponto, conjunto esse obtido a partir de análise estática da hierarquia de classes. Como ressalta Clarke e Malloy [86], mesmo essa abordagem ainda requer algum refinamento, uma vez que o número de métodos polimórficos identificado pode ser grande, tornando a construção do grafo impraticável.

Sinha e Harrold [364] desenvolveram uma família de seis critérios de teste destinados especificamente ao teste do comportamento de construções relacionadas ao tratamento de exceções da linguagem Java. A definição dos critérios foi baseada em uma representação de programa conhecida como Grafo de Fluxo de Programa Interprocedimental (ICFG) [363], a partir da qual os requisitos de teste são derivados. Exceções em Java podem ser classificadas como sendo síncronas ou assíncronas. Uma exceção síncrona ocorre em um ponto particular do programa e é causada durante a avaliação de uma expressão, da execução de um comando ou pela execução explícita de um comando `throw`. Uma exceção assíncrona pode ocorrer de forma arbitrária, em qualquer ponto do programa. A principal limitação na construção do ICFG é que ele não representa exceções assíncronas nem aquelas que podem ser geradas implicitamente, ou seja, o algoritmo para a construção do ICFG realiza uma busca local nas instruções de cada método para determinar os tipos de exceções que possam ser gerados explicitamente por meio de um comando `throw`.

Considerando o teste intramétodo, Vincenzi et al. [420] revisitaram diversos trabalhos na definição de critérios de teste de fluxo de controle e de dados que permitem a derivação de requisitos de testes a partir de código objeto Java (Java bytecode). A principal motivação para trabalhar nesse nível é viabilizar o teste estrutural não somente para programas Java tradicionais, mas também permitir o teste estrutural de componentes Java para os quais, em

geral, nem sempre o código-fonte está disponível. Além da definição dos critérios de teste, Vincenzi et al. [422] também implementaram uma ferramenta de teste, denominada JaBUTi (**J**ava **B**yte**C**ode **U**nderstanding and **T**e**sting**), que automatiza a aplicação dos critérios.

Para ilustrar os critérios de teste definidos por Vincenzi et al. [420], considere o programa Java apresentado na Figura 6.4(a). A classe Vet contém um método `Vet.average()` que calcula e retorna a média dos números armazenados em um vetor de inteiros. As instruções de *bytecode* que representam o método `Vet.average()` são apresentadas na Figura 6.4(b). Cada instrução de *bytecode* é precedida de um contador de programa (cp) que identifica unicamente cada instrução em um dado método. No exemplo, a primeira instrução de *bytecode*, `aload_0`, é indicada pelo cp 0, e a última, `freturn`, pelo cp 101.

A Figura 6.4(c) ilustra a tabela de exceções do método `Vet.average()`. A tabela indica, para cada segmento de instruções de *bytecode* (colunas *from* e *to*), o início (coluna *target*) de cada tratador de exceção válido, bem como o tipo de exceção que é tratada pelo tratador (coluna *type*). Por exemplo, do cp 12 ao cp 54 o tratador de exceção válido começa no cp 60 e é responsável por tratar qualquer exceção da classe `java.lang.Exception` ou qualquer uma de suas subclasses. Um tratador de exceção que esteja registrado como sendo do tipo `<Class a11>` é responsável por tratar quaisquer exceções que sejam geradas dentro de seu escopo de atuação.

As instruções de *bytecode* podem ser relacionadas com linhas do código-fonte, pois o arquivo `.class` armazena informações que permitem mapear cada instrução de *bytecode* para a linha de código-fonte que lhe deu origem. Com isso, se o código-fonte estiver disponível, as análises realizadas no *bytecode* podem ser mapeadas de volta para o código-fonte. A tabela de números de linha (*line number table*), ilustrada na Figura 6.4(d), fornece tal correspondência. Por exemplo, o comando localizado na linha número 06 do código-fonte corresponde às instruções de *bytecode* do cp 0 ao cp 2. As instruções de *bytecode* do cp 5 ao 7 correspondem ao comando da linha 07 no código-fonte, e assim por diante.

O chamado Grafo de Instruções (GI), ilustrado na Figura 6.4(e), representa o fluxo de controle das instruções de *bytecode* do método `Vet.average()`. Cada nó do grafo corresponde a uma única instrução de *bytecode*. Os nós do GI são numerados de acordo com o número do contador de programa (cp) das respectivas instruções de *bytecode*.

Dois tipos diferentes de arestas são utilizados para conectar as instruções de *bytecode*: 1) arestas regulares (linhas contínuas), considerando a existência de transferência de controle entre cada instrução; e 2) arestas de exceção (linhas pontilhadas), considerando a tabela de exceções. Os desvios condicionais são identificados a partir da semântica das instruções de *bytecode* responsáveis por tais desvios. Também por meio da análise semântica das instruções de *bytecode* é possível identificar quais delas são responsáveis por causar uma definição e/ou um uso de variável e, consequentemente, dar origem aos conjuntos `def(i)` e `uso(i)`, correspondentes aos conjuntos de variáveis definidas e usadas em cada nó *i* do grafo GI, respectivamente. Para mais informações, pode ser consultado o trabalho de Vincenzi et al. [420].

Uma vez que cada instrução de *bytecode* corresponde a um único nó em GI, o número de nós e arestas é relativamente grande, mesmo para métodos com poucas linhas de código-fonte. No caso do GI apresentado na Figura 6.4(e), alguns nós foram omitidos para reduzir o tamanho do GI e melhorar a sua legibilidade. Reticências (“...”) foram utilizadas para representar os nós omitidos.

```

/*01*/public class Vet {
/*02*/    int v[];           de ate destino tipo
/*03*/    float out;
/*04*/
/*05*/    float average(int[] in) { 12 54   60 <Class java.lang.Exception>
/*06*/        v = in;          12 57   74 <Class all>
/*07*/        out = 0.0f;      60 71   74 <Class all>
/*08*/        int i = 0;       74 79   74 <Class all>
/*09*/
/*10*/        try {          (c) Tabela de Exceção
/*11*/            while (i < v.length) { Linha 06: cp = 0
/*12*/                out += v[i]; Linha 07: cp = 5
/*13*/                i++;          Linha 08: cp = 10
/*14*/            }                  Linha 11: cp = 12
/*15*/            out = out / i; Linha 12: cp = 15
/*16*/        } catch (Exception e) { Linha 13: cp = 31
/*17*/            out = 0.0f;      Linha 14: cp = 34
/*18*/            i = 0;          Linha 15: cp = 43
/*19*/        } finally {      Linha 16: cp = 54
/*20*/            v = null;     Linha 17: cp = 60
/*21*/        }
/*22*/        print((float) i); Linha 18: cp = 66
/*23*/        return out;    Linha 19: cp = 68
/*24*/    }                  Linha 20: cp = 74
/*25*/
/*26*/    public void print(float n) { Linha 22: cp = 91
/*27*/        System.out.print(n + "\n");
/*28*/
/*29*/

```

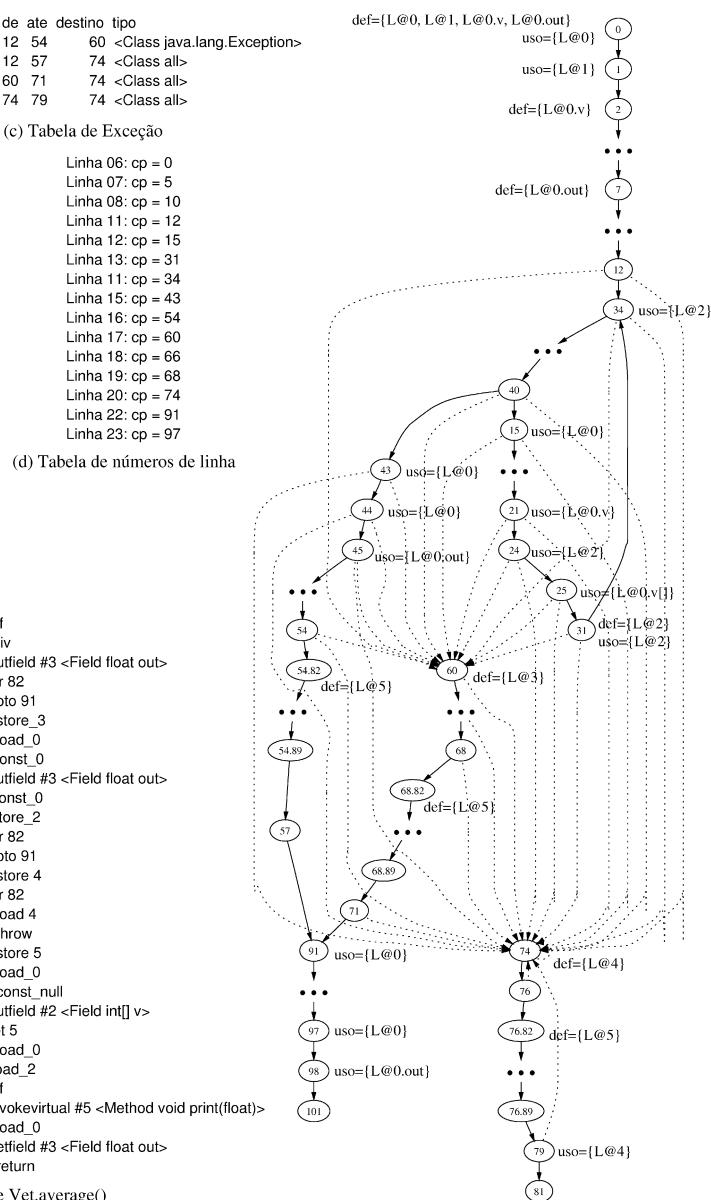
(a) Código-fonte Vet.java

```

0 aload_0          49 i2f
1 aload_1          50 fdiv
2 putfield #2 <Field int[] v> 51 putfield #3 <Field float out>
5 aload_0          54 jsr 82
6 fconst_0         57 goto 91
7 putfield #3 <Field float out> 60 astore_3
10 iconst_0        61 aload_0
11 istore_2        62 fconst_0
12 goto 34         63 putfield #3 <Field float out>
15 aload_0          66 iconst_0
16 dup             67 istore_2
17 getfield #3 <Field float out> 68 jsr 82
20 aload_0          71 goto 91
21 getfield #2 <Field int[] v> 74 astore 4
24 iload_2          76 jsr 82
25 iload           79 aload 4
26 i2l             81 athrow
27 fadd            82 astore 5
28 putfield #3 <Field float out> 84 aload_0
31 inc 2 1          85 aconst_null
34 iload_2          86 putfield #2 <Field int[] v>
35 aload_0          89 ret 5
36 getfield #2 <Field int[] v> 91 aload_0
39 arraylength      92 iload_2
40 if_icmpgt 15    93 i2f
43 aload_0          94 invokevirtual #5 <Method void print(float)>
44 iload_0          97 aload_0
45 getfield #3 <Field float out> 98 getfield #3 <Field float out>
48 iload_2          101 freturn

```

(b) Instruções de bytecode de Vet.average()

Figura 6.4 – Ilustração de um Grafo de Instruções para o método `Vet.average()`.

O GI oferece um modo prático de se percorrer o conjunto de instruções de um dado método, identificando o conjunto de variáveis definidas e usadas por tais instruções. Entretanto, uma vez que um nó em GI é criado para cada instrução de *bytecode* de um método *m*, o número de nós e arestas pode ser demasiadamente grande. Assim sendo, uma vez coletadas

todas as informações necessárias a respeito do conjunto de instruções de um dado método, o conceito de bloco de instruções é utilizado para reduzir o número de nós e arestas de um grafo GI tanto quanto possível.

Um bloco de instruções é definido como um conjunto de instruções que são, “normalmente”, executadas em seqüência. Quando a primeira instrução do bloco é executada, todas as demais instruções também o são; desvios de execução somente ocorrem para o início de um bloco. A palavra “normalmente” é utilizada anteriormente para indicar o fluxo de controle determinado pelos comandos do programa, ou seja, sem considerar que possíveis interrupções possam impedir a execução do bloco de comandos como um todo.

Desse modo, é percorrido o Grafo de Instruções e, por meio do agrupamento de diversas instruções individuais, são formados blocos de instruções que dão origem a um novo nó no Grafo Def-Uso (GDU). Os conjuntos de variáveis definidas e usadas, relacionados aos nós do GDU são obtidos por meio da união dos conjuntos individuais de cada instrução que compõe o bloco. O GDU representa o modelo base que é utilizado para se derivarem requisitos de teste de fluxo de controle e de dados intramétodo para o teste de programas e componentes Java. A Figura 6.5 representa o Grafo Def-Uso resultante do GI da Figura 6.4(e).

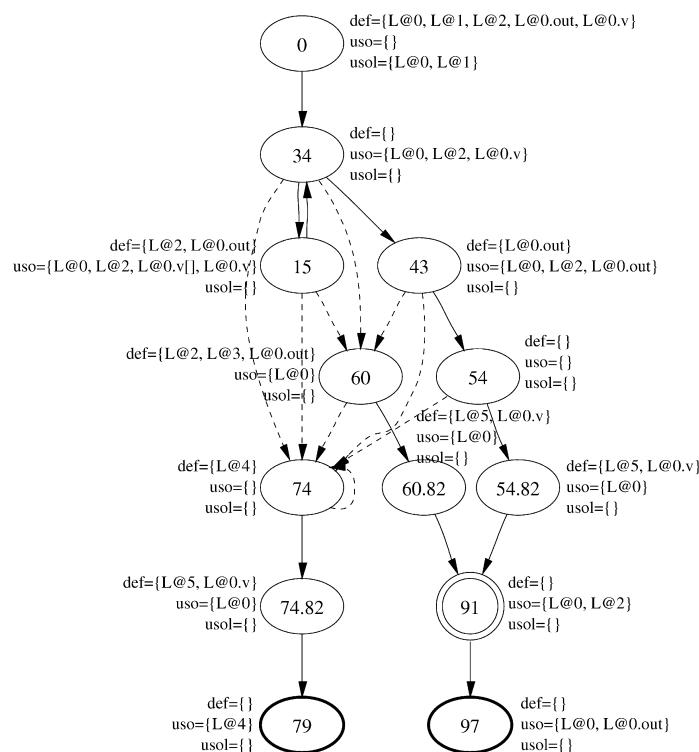


Figura 6.5 – Grafo Def-Uso do método `Vet.average()`.

Com base no GDU, um conjunto de oito critérios de teste foi definido por Vincenzi et al. [420]. As Tabelas 6.2 e 6.3 descrevem de forma sucinta tais critérios.

Tabela 6.2 – Critérios de teste baseados em fluxo de controle

| Critério      |                             | Descrição   |
|---------------|-----------------------------|---|
| Todos-Nós     | Todos-Nós <sub>ei</sub>     | Exige a cobertura de todos os comandos não relacionados ao tratamento de exceção                                  |
|               | Todos-Nós <sub>ed</sub>     | Exige a cobertura de todos os comandos relacionados ao tratamento de exceção                                      |
| Todas-Arestas | Todas-Arestas <sub>ei</sub> | Exige cobertura de todos os desvios condicionais do método (desvios não decorrentes do lançamento de uma exceção) |
|               | Todas-Arestas <sub>ed</sub> | Exige cobertura de todos os desvios de execução decorrentes do lançamento de uma exceção                          |

Tabela 6.3 – Critérios de teste baseados em fluxo de dados

| Critério       |                              | Descrição  |
|----------------|------------------------------|--|
| Todos-Usos     | Todos-Usos <sub>ei</sub>     | Exige a cobertura de todas as associações definição-uso não relacionadas ao tratamento de exceção            |
|                | Todos-Usos <sub>ed</sub>     | Exige a cobertura de todas as associações definição-uso relacionadas ao tratamento de exceção                |
| Todos-Pot-Usos | Todos-Pot-Usos <sub>ei</sub> | Exige a cobertura de todas as potenciais-associações definição-uso não relacionadas ao tratamento de exceção |
|                | Todos-Pot-Usos <sub>ed</sub> | Exige a cobertura de todas as potenciais-associações definição-uso relacionadas ao tratamento de exceção     |

Considerando os critérios de fluxo de controle e de fluxo de dados e o GDU da Figura 6.5, os requisitos de testes resultantes estão ilustrados nas Tabelas 6.4, 6.5 e 6.6, respectivamente.

Observe que os critérios de teste descritos formam uma hierarquia de critérios, sendo que, quanto mais alto na hierarquia, maior o número de casos de testes necessários para satisfazer o critério e, conseqüentemente, maior o custo de sua aplicação. Quanto mais baixo na hierarquia, o critério é mais fácil de ser satisfeito e menor seu custo de aplicação. A Figura 6.6 ilustra a hierarquia entre os critérios definidos e os critérios tradicionais de fluxo de controle e de dados. Com base nessa hierarquia de critérios, Vincenzi [413] definiu uma estratégia incremental de utilização dos critérios apresentada a seguir.

Tabela 6.4 – Conjunto de requisitos de teste estruturais derivados pelos critérios de fluxo de controle para o método `Vet.average()`

| Critério      |                             | Requisitos de Teste   |
|---------------|-----------------------------|---|
| Todos-Nós     | Todos-Nós <sub>ei</sub>     | {0, 15, 34, 43, 54, 54.82, 91, 97}  |
|               | Todos-Nós <sub>ed</sub>     | {60, 60.82, 74, 74.82, 79 }   |
| Todas-Arestas | Todas-Arestas <sub>ei</sub> | {(0,34), (15,34), (34,15), (34,43), (43,54), (54,54.82), (54.82,91), (91,97)}   |
|               | Todas-Arestas <sub>ed</sub> | {(15,60), (15,74), (34,60), (34,74), (43,60), (43,74), (54,74), (60,60.82), (60,74), (60.82,91), (74,74), (74,74.82), (74.82,79)} |

Tabela 6.5 – Conjunto de requisitos de teste estruturais derivados pelo critério Todos-Usos para o método `Vet.average()`

| Critério   |                                     | Requisitos de Teste  |  |   |
|------------|-------------------------------------|--|--|---|
| Todos-Usos | Todos-Usos <sub>ei</sub>            | $\langle 0, 15, L@0 \rangle$<br>$\langle 0, 43, L@0 \rangle$<br>$\langle 0, 97, L@0 \rangle$<br>$\langle 43, 97, L@0.out \rangle$<br>$\langle 0, (34, 43), L@0.v \rangle$<br>$\langle 15, (34, 15), L@2 \rangle$<br>$\langle 0, (34, 43), L@2 \rangle$<br>$\langle 0, 91, L@2 \rangle$ | $\langle 0, (34, 15), L@0 \rangle$<br>$\langle 0, 54.82, L@0 \rangle$<br>$\langle 0, 43, L@0.out \rangle$<br>$\langle 0, 15, L@0.v \rangle$<br>$\langle 0, (34, 15), L@0.v[] \rangle$<br>$\langle 0, (34, 15), L@2 \rangle$<br>$\langle 0, 43, L@2 \rangle$<br>$\langle 15, 91, L@2 \rangle$ | $\langle 0, (34, 43), L@0 \rangle$<br>$\langle 0, 91, L@0 \rangle$<br>$\langle 15, 43, L@0.out \rangle$<br>$\langle 0, (34, 15), L@0.v \rangle$<br>$\langle 0, 15, L@2 \rangle$<br>$\langle 15, (34, 43), L@2 \rangle$<br>$\langle 15, 43, L@2 \rangle$ |
|            | Todos-Usos <sub>ed</sub>            | $\langle 0, 60, L@0 \rangle$<br>$\langle 60, 97, L@0.out \rangle$  | $\langle 0, 60.82, L@0 \rangle$<br>$\langle 60, 91, L@2 \rangle$   | $\langle 0, 74.82, L@0 \rangle$<br>$\langle 74, 79, L@4 \rangle$  |
|            | Todos-Potenciais-Usos               |  |  |   |
|            | Todos-Potenciais-Usos <sub>ei</sub> |  |  |   |
|            | Todos-Potenciais-Usos <sub>ed</sub> |  |  |   |
|            | Todos-Usosei                        |  |  |   |
|            | Todas-Arestas <sub>ei</sub>         |  |  |   |
|            | Todas-Arestas <sub>ed</sub>         |  |  |   |
|            | Todos-Nós <sub>ei</sub>             |  |  |   |
|            | Todos-Nós <sub>ed</sub>             |  |  |   |

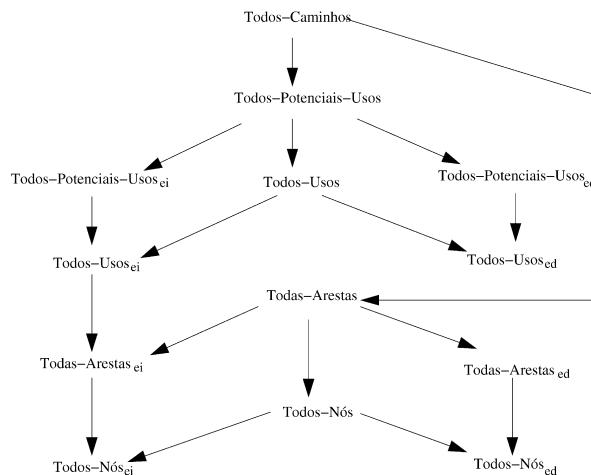


Figura 6.6 – Hierarquia entre os critérios de testes estruturais intramétodo [413].

## Estratégia incremental

Para atingir uma alta cobertura dos requisitos de teste é possível utilizar um conjunto de teste gerado em diversas fases, considerando uma estratégia incremental de aplicação dos critérios de teste. Com isso, o testador pode se concentrar em diferentes aspectos do programa que

Tabela 6.6 – Conjunto de requisitos de teste estruturais derivados pelo critério Todos-Potenciais-Usos para o método `Vet.average()`

| Critério                     | Requisitos de Teste                     |   |  |
|------------------------------|---|---|--|
| Todos-Pot-Usos <sub>ei</sub> | $\langle 0, 15, L@0 \rangle$            | $\langle 0, (34, 15), L@0 \rangle$      | $\langle 0, (34, 43), L@0 \rangle$     |
|                              | $\langle 0, 43, L@0 \rangle$            | $\langle 0, 54, L@0 \rangle$            | $\langle 0, 54.82, L@0 \rangle$        |
|                              | $\langle 0, 91, L@0 \rangle$            | $\langle 0, 97, L@0 \rangle$            | $\langle 0, 15, L@0.out \rangle$       |
|                              | $\langle 0, (34, 15), L@0.out \rangle$  | $\langle 15, (34, 15), L@0.out \rangle$ | $\langle 0, (34, 43), L@0.out \rangle$ |
|                              | $\langle 15, (34, 43), L@0.out \rangle$ | $\langle 15, 43, L@0.out \rangle$       | $\langle 0, 43, L@0.out \rangle$       |
|                              | $\langle 43, 54, L@0.out \rangle$       | $\langle 43, 54.82, L@0.out \rangle$    | $\langle 43, 91, L@0.out \rangle$      |
|                              | $\langle 43, 97, L@0.out \rangle$       | $\langle 0, 15, L@0.v \rangle$          | $\langle 0, (34, 15), L@0.v \rangle$   |
|                              | $\langle 0, (34, 43), L@0.v \rangle$    | $\langle 0, 43, L@0.v \rangle$          | $\langle 0, 54, L@0.v \rangle$         |
|                              | $\langle 0, 54.82, L@0.v \rangle$       | $\langle 54.82, 91, L@0.v \rangle$      | $\langle 54.82, 97, L@0.v \rangle$     |
|                              | $\langle 0, 15, L@1 \rangle$            | $\langle 0, (34, 15), L@1 \rangle$      | $\langle 0, (34, 43), L@1 \rangle$     |
|                              | $\langle 0, 43, L@1 \rangle$            | $\langle 0, 54, L@1 \rangle$            | $\langle 0, 54.82, L@1 \rangle$        |
|                              | $\langle 0, 91, L@1 \rangle$            | $\langle 0, 97, L@1 \rangle$            | $\langle 0, 15, L@2 \rangle$           |
|                              | $\langle 15, (34, 15), L@2 \rangle$     | $\langle 0, (34, 15), L@2 \rangle$      | $\langle 15, (34, 43), L@2 \rangle$    |
|                              | $\langle 0, (34, 43), L@2 \rangle$      | $\langle 15, 43, L@2 \rangle$           | $\langle 0, 43, L@2 \rangle$           |
|                              | $\langle 15, 54, L@2 \rangle$           | $\langle 0, 54, L@2 \rangle$            | $\langle 15, 54.82, L@2 \rangle$       |
|                              | $\langle 0, 54.82, L@2 \rangle$         | $\langle 15, 91, L@2 \rangle$           | $\langle 0, 91, L@2 \rangle$           |
|                              | $\langle 15, 97, L@2 \rangle$           | $\langle 0, 97, L@2 \rangle$            | $\langle 54.82, 91, L@5 \rangle$       |
|                              | $\langle 54.82, 97, L@5 \rangle$        |   |  |
| Todos-Pot-Usos               | $\langle 0, 60, L@0 \rangle$            | $\langle 0, 60.82, L@0 \rangle$         | $\langle 0, 74, L@0 \rangle$           |
|                              | $\langle 0, 74.82, L@0 \rangle$         | $\langle 0, 79, L@0 \rangle$            | $\langle 15, 60, L@0.out \rangle$      |
|                              | $\langle 0, 60, L@0.out \rangle$        | $\langle 43, 60, L@0.out \rangle$       | $\langle 60, 60.82, L@0.out \rangle$   |
|                              | $\langle 60, 74, L@0.out \rangle$       | $\langle 15, 74, L@0.out \rangle$       | $\langle 43, 74, L@0.out \rangle$      |
|                              | $\langle 0, 74, L@0.out \rangle$        | $\langle 15, 74.82, L@0.out \rangle$    | $\langle 0, 74.82, L@0.out \rangle$    |
|                              | $\langle 43, 74.82, L@0.out \rangle$    | $\langle 60, 74.82, L@0.out \rangle$    | $\langle 60, 79, L@0.out \rangle$      |
|                              | $\langle 15, 79, L@0.out \rangle$       | $\langle 0, 79, L@0.out \rangle$        | $\langle 43, 79, L@0.out \rangle$      |
|                              | $\langle 60, 91, L@0.out \rangle$       | $\langle 60, 97, L@0.out \rangle$       | $\langle 0, 60, L@0.v \rangle$         |
|                              | $\langle 0, 60.82, L@0.v \rangle$       | $\langle 0, 74, L@0.v \rangle$          | $\langle 0, 74.82, L@0.v \rangle$      |
|                              | $\langle 74.82, 79, L@0.v \rangle$      | $\langle 60.82, 91, L@0.v \rangle$      | $\langle 60.82, 97, L@0.v \rangle$     |
|                              | $\langle 0, 60, L@1 \rangle$            | $\langle 0, 60.82, L@1 \rangle$         | $\langle 0, 74, L@1 \rangle$           |
|                              | $\langle 0, 74.82, L@1 \rangle$         | $\langle 0, 79, L@1 \rangle$            | $\langle 15, 60, L@2 \rangle$          |
|                              | $\langle 0, 60, L@2 \rangle$            | $\langle 60, 60.82, L@2 \rangle$        | $\langle 60, 74, L@2 \rangle$          |
|                              | $\langle 15, 74, L@2 \rangle$           | $\langle 0, 74, L@2 \rangle$            | $\langle 15, 74.82, L@2 \rangle$       |
|                              | $\langle 0, 74.82, L@2 \rangle$         | $\langle 60, 74.82, L@2 \rangle$        | $\langle 60, 79, L@2 \rangle$          |
|                              | $\langle 15, 79, L@2 \rangle$           | $\langle 0, 79, L@2 \rangle$            | $\langle 60, 91, L@2 \rangle$          |
|                              | $\langle 60, 97, L@2 \rangle$           | $\langle 60, 60.82, L@3 \rangle$        | $\langle 60, 74, L@3 \rangle$          |
|                              | $\langle 60, 74.82, L@3 \rangle$        | $\langle 60, 79, L@3 \rangle$           | $\langle 60, 91, L@3 \rangle$          |
|                              | $\langle 60, 97, L@3 \rangle$           | $\langle 74, 74.82, L@4 \rangle$        | $\langle 74, 79, L@4 \rangle$          |
|                              | $\langle 74.82, 79, L@5 \rangle$        | $\langle 60.82, 91, L@5 \rangle$        | $\langle 60.82, 97, L@5 \rangle$       |

está sendo testado e gerenciar mais facilmente os recursos de tempo e custo em função da qualidade dos testes desejada.

A estratégia definida a seguir é fundamentada na relação de inclusão entre os critérios de teste. O objetivo é aplicar um critério de teste menos rigoroso e de menor custo e, à medida que as restrições de tempo e custo permitirem, considerar a aplicação de critérios mais rigorosos visando a evoluir o conjunto de teste e aumentar a confiança de que o software se comporte de acordo com o especificado. Na definição da estratégia, os seguintes conjuntos de teste são considerados:

- $T_f$  correspondente a um teste inicial, em geral desenvolvido com base em algum critério de teste funcional;
- $T_n$  correspondente a um conjunto de teste adequado ao critério Todos-Nós (Todos-Nós<sub>ei</sub>/Todos-Nós<sub>ed</sub>);

- $T_a$  correspondente a um conjunto de teste adequado ao critério Todas-Arestas (Todas-Arestas<sub>ei</sub>/Todas-Arestas<sub>ed</sub>);
- $T_u$  correspondente a um conjunto de teste adequado ao critério Todos-Usos (Todos-Usos<sub>ei</sub>/Todos-Usos<sub>ed</sub>); e
- $T_{pu}$  correspondente a um conjunto de teste adequado ao critério Todos-Pot-Usos (Todos-Pot-Usos<sub>ei</sub>/Todos-Pot-Usos<sub>ed</sub>).

Os nove passos necessários para a aplicação da estratégia incremental são descritos na Tabela 6.7.

Tabela 6.7 – Seqüência de passos da estratégia incremental

| Passo   | Descrição   |
|---------|---|
| Passo 1 | Elaborar um conjunto de teste funcional ( $T_f$ ), utilizando um critério funcional tal como Análise do Valor Limite, Particionamento em Classes de Equivalência, <i>Error-Guessing</i> , ou uma combinação deles [243], considerando a especificação das classes que estão sendo testadas (CST). |
| Passo 2 | Avaliar a cobertura de $T_f$ em relação aos critérios de teste estruturais.   |
| Passo 3 | Evoluir $T_f$ até obter um conjunto de teste ( $T_n$ ) adequado ao critério Todos-Nós (Todos-Nós <sub>ei</sub> /Todos-Nós <sub>ed</sub> ) para as CST.  |
| Passo 4 | Avaliar a cobertura de $T_n$ em relação aos demais critérios de teste estruturais.  |
| Passo 5 | Evoluir $T_n$ até obter um conjunto de teste ( $T_a$ ) adequado ao critério Todas-Arestas (Todas-Arestas <sub>ei</sub> /Todas-Arestas <sub>ed</sub> ) para as CST.  |
| Passo 6 | Avaliar a cobertura de $T_a$ em relação aos demais critérios de teste estruturais.  |
| Passo 7 | Evoluir $T_a$ até obter um conjunto de teste ( $T_u$ ) adequado ao critério Todos-Usos (Todos-Usos <sub>ei</sub> /Todos-Usos <sub>ed</sub> ) para as CST.   |
| Passo 8 | Avaliar a cobertura de $T_u$ em relação aos demais critérios de teste estruturais.  |
| Passo 9 | Evoluir $T_u$ até obter um conjunto de teste ( $T_p$ ), adequado ao critério Todos-Pot-Usos (Todos-Pot-Usos <sub>ei</sub> /Todos-Pot-Usos <sub>ed</sub> ) para as CST.  |

Observa-se que a estratégia considera, inicialmente, o desenvolvimento de um conjunto de teste funcional. Tal conjunto de teste pode ser gerado por meio de algum algoritmo de geração aleatório, com base em algum critério de teste funcional, ou até mesmo com base no conhecimento do testador sobre o programa que está sendo testado. Esses casos de teste servem como ponto de partida para a avaliação de cobertura. Se os resultados obtidos não satisfizerem os objetivos preestabelecidos, casos de teste adicionais podem ser desenvolvidos para alcançar tais objetivos.

A idéia é evoluir o conjunto de teste de modo a satisfazer, inicialmente, os requisitos mínimos do teste estrutural, ou seja, a cobertura de comandos (critério Todos-Nós) e em seguida a cobertura de decisões (critério Todas-Arestas). A seguir, critérios de teste que avaliam a relação entre a definição e uso de variáveis são considerados, objetivando garantir que as associações de fluxo de dados do programa estão corretas. Nesse contexto, sugere-se a utilização do critério Todos-Usos seguido do critério Todos-Pot-Usos.

Além disso, dependendo do tipo de programa que está sendo testado ou até dos objetivos de teste desejados, o testador pode escolher aplicar, em um primeiro momento, somente os critérios independentes de exceção. Posteriormente, conforme as restrições de tempo e custo, os critérios dependentes de exceção também podem ser aplicados, seguindo os passos da estratégia.

Observa-se que, em geral, a obtenção de conjuntos de teste adequados requer a identificação de possíveis requisitos de teste não executáveis, tarefa essa a ser realizada em todas as etapas que exigem a geração de conjuntos de teste adequados com relação a algum critério de teste. Formulários específicos devem ser preenchidos durante e ao final de cada etapa para anotar os dados pertinentes.

Como pode ser observado, os critérios de teste definidos até o momento são destinados ao teste intramétodo, sendo que, nesse nível de teste, o teste de POO não é essencialmente diferente do teste de programas procedimentais. Entretanto, os testes intramétodos são de fundamental importância para assegurar que a lógica de cada método esteja correta. Os demais problemas decorrentes do uso das características de orientação a objetos estão mais relacionados a defeitos de integração e, a seguir, são descritos sucintamente alguns trabalhos nesse contexto.

## Teste estrutural de integração

Souter e Pollock [369] estabeleceram o conceito de associações definição-uso contextuais (*contextual def-use associations – cduas*), as quais agregam informação sobre o contexto de uso dos objetos e exigem determinado contexto na cobertura de determinada associação. A motivação, segundo as autoras, é que mesmo que as abordagens de teste procedimentais possam ser estendidas para lidar com as novas propriedades de linguagens OO, a utilidade dessas extensões não está suficientemente clara.

Com base em um estudo experimental realizado em um trabalho anterior com um conjunto de programas grandes escritos em Java, Souter et al. [371] chegaram à conclusão de que a utilização do projeto de software OO resulta em programas com muitos métodos, cada um com um número limitado de sentenças e com fluxo de controle intramétodo bastante simples.

Um sumário 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 variou 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 revelarem os comportamentos de um programa OO. Além disso, Souter e Pollock [369] argumentam que as abordagens de fluxo de dados, como, por exemplo, a de Harrold e Rothermel [169] discutida anteriormente, não atingem o objetivo principal do teste de fluxo de dados OO de levar em conta as manipulações de objetos. Com isso, as associações de fluxo de dados obtidas a partir dos critérios de Harrold e Rothermel [169] seriam classificadas como sendo livres de contexto (*context-free*), uma vez que podem ser cobertas sem considerar um contexto específico.

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 [432] 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 ser acessados por outras regiões. Souter e Pollock [370] utilizam esse tipo de informação, estendendo o grafo *points-to escape* para um grafo *Annotated Points-to Escape* (APE) a fim de caracterizarem pares Def-Uso contextuais, agregando contexto às definições e usos dos objetos. As definições e os usos de objetos são baseados nas modificações e nos acessos aos atributos dos objetos. Para permitir a geração de *cdus*, informações adicionais sobre a localização e o tipo de manipulação (definição (*store*) ou uso (*load*)) realizada no objeto também são incluídas no grafo. Segundo as autoras, o teste baseado em associações definição-uso contextuais pode melhorar a cobertura dos testes, uma vez que múltiplas associações definição-uso contextuais únicas podem ser geradas para uma mesma associação definição-uso livre de contexto. Ao todo, quatro níveis de contexto foram propostos, identificados por cdu-0, cdu-1, cdu-2 e cdu-3. Quanto maior o nível de contexto, maior o conjunto de associações definição-uso contextuais requeridas e, consequentemente, maiores os requisitos computacionais exigidos para calculá-las.

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 seqüência 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 seqüência interna é o local de chamada em uma seqüência de chamadas que leva do local da chamada original,  $CS_{om}$ , à atribuição (leitura).

Essas seqüências de chamadas é que configuram contextos para cada associação Def-Uso, enquanto as associações definidas nas abordagens anteriores apenas consideravam a definição e o uso em qualquer lugar, sem considerar as seqüências 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. Além disso, 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).

Entretanto, Clarke e Malloy [86] ressaltam que a abordagem proposta não considera associações definição-uso contextuais envolvendo variáveis de tipo primitivo. Além disso, conforme destacado por Souter e Pollock [369, 370], em programas que contenham longas cadeias de chamadas e métodos com múltiplas chamadas para o mesmo método é possível atingir um número de anotações com tamanho exponencial, considerando um APE com anotações precisas. A solução adotada pelas autoras foi, então, limitar o tamanho das anotações

de modo que a construção do APE tivesse uma complexidade da ordem de  $O(n^4)$ , sendo  $n$  o número de nós do APE e, mesmo assim, na condução de alguns experimentos com programas da ordem de 7.500 a 9.700 linhas de código (LOC), a quantidade de memória requerida para computar o conjunto de associações definição-uso contextuais cdu-2 e cdu-3 não foi suficiente.

Independente da fase ou do critério de teste utilizado, a atividade de teste é considerada de alto custo, e reduzir esses custos é de fundamental importância para promover um incentivo na adoção de uma estratégia de teste de baixo custo e alta eficácia em revelar defeitos nos produtos de software. Com a finalidade de reduzir parte dos custos decorrentes da atividade de testes em POO, Harrold e Rothermel propuseram a chamada Estratégia de Teste Incremental Hierárquica (HIT). Tal estratégia é descrita a seguir detalhadamente, pois considera-se que fornece dicas úteis que favorecem a reutilização de casos de testes quando POO fazem uso de herança.

### 6.5.2 Estratégia de teste incremental hierárquica

Harrold et al. [168] definiram uma estratégia para reaproveitar conjuntos de casos de teste adequados ao teste da superclasse para testar características herdadas nas subclasses, reduzindo os custos da atividade de teste.

Como descrito anteriormente, a herança é um mecanismo que permite tanto o compartilhamento da especificação da classe como o do código-fonte para o desenvolvimento de novas classes baseadas em classes já existentes. A definição de uma subclass é dada por um **modificador** que estabelece as diferenças ou alterações nos elementos<sup>3</sup> que compõem a superclasse. Com isso, um modificador e a superclasse são utilizados na criação de uma subclass. A Figura 6.7 ilustra como transformar uma superclasse  $P$  com um modificador  $M$  em uma subclass  $R$ . O operador de composição  $\otimes$  une simbolicamente  $M$  e  $P$ , produzindo  $R$ , sendo  $R = P \otimes M$ .

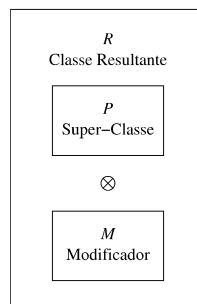


Figura 6.7 – Técnica de modificação incremental [168].

O projetista da subclass especifca o modificador, o qual pode conter diferentes tipos de elementos que alteram a superclasse. Os seis tipos de elementos são [168]:

<sup>3</sup>A palavra elemento é utilizada aqui para denotar tudo aquilo que possa estar presente em uma classe e, eventualmente, sofrer alguma alteração na relação de herança, tal como atributos, métodos ou até mesmo a lista de parâmetros formais de determinado método.

- elemento novo:  $A$  é um elemento definido em  $M$  e não em  $P$  ou  $A$  é um elemento de um método em  $M$  e  $P$ , mas a lista de parâmetros de  $A$  é diferente em  $M$  e  $P$ . Nesse caso,  $A$  é limitado ao elemento definido na classe resultante  $R$  mas não em  $P$ .
- elemento recursivo: o elemento  $A$  é definido em  $P$  mas não em  $M$ . Nesse caso,  $A$  é limitado ao elemento definido em  $P$  e está disponível em  $R$ .
- elemento redefinido:  $A$  é definido em  $P$  e  $M$  e a lista de parâmetros de  $A$  é a mesma em  $P$  e  $M$ . Nesse caso,  $A$  é limitado à definição do elemento em  $M$ , que bloqueia a definição do elemento em  $P$ .
- elemento virtual-novo:  $A$  é especificado em  $M$ , mas sua implementação pode estar incompleta para permitir posterior definição, ou  $A$  é especificado em  $M$  e  $P$ , e sua implementação pode estar incompleta em  $P$  para permitir posterior definição, mas a lista de parâmetros de  $A$  é diferente em  $M$  e  $P$ . Nesse caso,  $A$  é limitado ao elemento definido em  $R$  mas não em  $P$ .
- elemento virtual-recursivo:  $A$  é especificado em  $P$ , mas sua implementação pode estar incompleta em  $P$  para permitir posterior definição, e  $A$  não está definido em  $M$ . Nesse caso,  $A$  é limitado ao elemento definido na classe  $P$  e está disponível na classe resultante  $R$ .
- elemento virtual-redefinido:  $A$  é especificado em  $P$ , mas sua implementação pode estar incompleta para permitir posterior definição, e  $A$  é definido em  $M$ , mantendo algumas características de  $A$  definido em  $P$ . Nesse caso,  $A$  é limitado ao elemento definido em  $M$  que bloqueia os pontos em comum entre as definições.

A Figura 6.8 ilustra alguns dos tipos de elementos. A classe  $P$  tem dois atributos inteiros,  $i$  e  $j$ , e métodos  $A$ ,  $B$  e  $C$ .  $B$  é um método virtual. O modificador para a classe  $R$  contém um atributo do tipo real,  $i$ , e três métodos:  $A$ ,  $B$  e  $C$ . Combinando-se o modificador com a classe  $P$ , a classe  $R$  é obtida. O atributo `float i` é um elemento novo em  $R$ , visto que ele não aparece em  $P$ . O método  $A$ , que está definido em  $M$ , é um elemento novo em  $R$ , uma vez que a lista de atributos não é igual à lista do  $A$  em  $P$ . O método  $A$  de  $P$  é recursivo em  $R$ , pois ele é herdado sem modificações a partir de  $P$ . Assim sendo,  $R$  contém dois métodos  $A$ . O método  $B$  é virtual em  $P$  e, por ser redefinido em  $M$ , ele é virtual-redefinido em  $R$ . O método  $C$  é redefinido em  $R$ , visto que sua implementação foi alterada por  $M$ , sobrescrevendo o método  $C$  de  $P$ . Finalmente, os atributos  $i$  e  $j$  de  $P$  são herdados mas estão ocultos em  $R$ , o que significa que eles não podem ser utilizados pelos métodos definidos pelo modificador.

A proposta consiste em testar inicialmente a superclasse considerando cada método individualmente e então testar a interação entre os métodos. Os casos de teste e as informações da execução dos testes na superclasse são salvos, caracterizando uma história de teste (*testing history*). Quando uma subclasse é definida, a história de teste de sua superclasse bem como o modificador são utilizados para determinar quais elementos devem ser testados (ou retestados) na subclasse e quais casos de testes da superclasse podem ser reaproveitados no teste da subclasse. Segundo Harrold et al., essa estratégia é hierárquica por ser guiada pela ordem da relação de hereditariedade, e é incremental porque usa o resultado obtido em um nível de teste na hierarquia para reduzir o esforço necessário em níveis subsequentes [168].

|  |   |
|--|---|
| <pre> class P { private:     int i;     int j; public:     P(){}     void A(int a, int b)         {i=a;j=a+2*b;}      virtual int B()         {return i;}     int C()         {return j;} };</pre>   | <pre> class R: public P { private:     float i;  public:     R(){}      void A(int A)         {P::A(a,0);}     virtual int B()         {return 3*P::B();}     int C()         {return 2*P::C();} };</pre> |
| <pre> // Elementos de R após o mapeamento  private:     float i;           // novo  public:     void A(int a, int b) // recursivo         {i=a;j=a+2*b;}     void A(int A)      // novo         {P::A(a,0);}     virtual int B()   // virtual redefinido         {return 3*P::B();}     int C()           // redefinido         {return 2*P::C();}</pre> |   |
| <pre> Ocultos     int i;     int j;</pre>  |   |

Figura 6.8 – Classe *P* à esquerda, especificação da subclasse *R* à direita (modificador) e os elementos da subclasse *R* embaixo [168].

## Teste da superclasse ou classe base

O teste da classe base inicia-se com o teste dos métodos (intramétodos), utilizando-se técnicas tradicionais de fluxo de dados. O teste individual de cada método é de fundamental importância, tendo em vista que a subclasse espera que o método herdado funcione corretamente. Nesse sentido, Harrold et al. destacam a importância de se utilizarem critérios funcionais e estruturais para assegurar uma melhor qualidade aos testes intramétodo. A história do teste consiste em uma tripla  $(m, (TF_i, \text{ teste?}), (TE_i, \text{ teste?}))$  em que  $m$  é o método,  $TF_i$  é o conjunto de teste baseado em especificação,  $TE_i$  o conjunto de teste baseado em programa e  $\text{teste?}$  indica se o conjunto de teste deve ou não ser (re)executado. O campo  $\text{teste?}$  pode ter três respostas diferentes,  $S$  indica que todo o conjunto de testes é reusado,  $P$  indica que somente os casos de teste identificados como partes afetadas do teste da subclasse são reusados e  $N$  indica que o conjunto de testes não é reusado.

Além do teste intramétodo também devem ser realizados os testes intraclasse e interclasse. O teste intraclasse procura testar as interações entre os métodos de uma mesma classe, e o teste interclasse, a interação entre métodos de classes diferentes.

Para a realização dos testes intraclasse é necessária a construção do Grafo de Classe (*class graph*) no qual cada nó representa um método e cada arco, uma mensagem entre métodos [168]. Novamente, para o teste intraclasse, critérios funcionais e estruturais devem ser utilizados. A história do teste é representada pela seguinte tripla  $(m_i, (\text{TIE}_i, \text{teste?}), (\text{TIE}_i, \text{teste?}))$ , em que  $m_i$  corresponde ao nó raiz do subgrafo,  $\text{TIE}_i$  é o conjunto de testes de integração baseado em especificação,  $\text{TIE}_i$  é conjunto de testes de integração baseado no programa, e  $\text{teste?}$  indica se o conjunto de teste deve ser executado ou não.

O teste interclasse é guiado pela interação de classes que resulta na troca de mensagens entre métodos de classes diferentes. Isso pode ocorrer, por exemplo, quando um método em uma classe é passado por uma instância de outra classe como um argumento e então envia uma mensagem àquela instância, ou quando uma instância de uma classe é parte da representação de outra classe e manda aquela instância como uma mensagem. A estratégia de teste utilizada nesse caso é a mesma utilizada para interações intraclasse.

O Programa 6.3 ilustra a declaração de uma classe abstrata `Forma` que permite a criação de classes derivadas. O Grafo de Classe resultante é mostrado na Figura 6.9. Retângulos representam métodos e círculos ovais representam instâncias de classes. Setas contínuas representam as mensagens intraclasse, e as setas pontilhadas, as mensagens interclasses.

---

Programa 6.3

---

```
1 class Forma {
2     private:
3         Ponto ponto_referência;
4
5     public:
6         void colocar_ponto_referência(Ponto);
7         point obter_ponto_referência();
8         void mover(Ponto);
9         void apagar();
10        virtual void desenhar() = 0;
11        virtual float área();
12        forma(Ponto);
13        forma();
14 }
```

---

A classe `Forma` é uma classe abstrata que pode ser utilizada na criação de diversas classes que representem o comportamento de figuras. `Forma` inclui duas funções membro virtuais, `área()` e `desenhar()`, contribuindo para o fornecimento de uma interface de acesso comum entre todas as subclasses na hierarquia de herança. A função `desenhar()` é uma função membro virtual definida com um valor inicial 0 e sem implementação. Embora não presente no Programa 6.3, o método `área()` tem uma implementação inicial que pode ser mudada numa subsequente declaração de subclasse.

Os métodos `colocar_ponto_referência()` e `obter_ponto_referência()` fornecem acesso controlado aos dados; os métodos `forma(Ponto)` e `forma()` são construtores da classe; e o método `mover()` muda o ponto de referência da figura. Esse método pode ser implementado em função dos outros métodos já definidos. Finalmente, o método `apagar()` pode ser implementado de várias formas. Uma delas seria chamar o método `desenhar()` no modo “xor”, ou seja, desenhar novamente a figura com a cor do fundo.

A Tabela 6.8 ilustra a história do teste para a `Forma`. Por se tratar da classe base, é necessário testar todos os métodos e todas as interações intraclasse e interclasse. Observa-

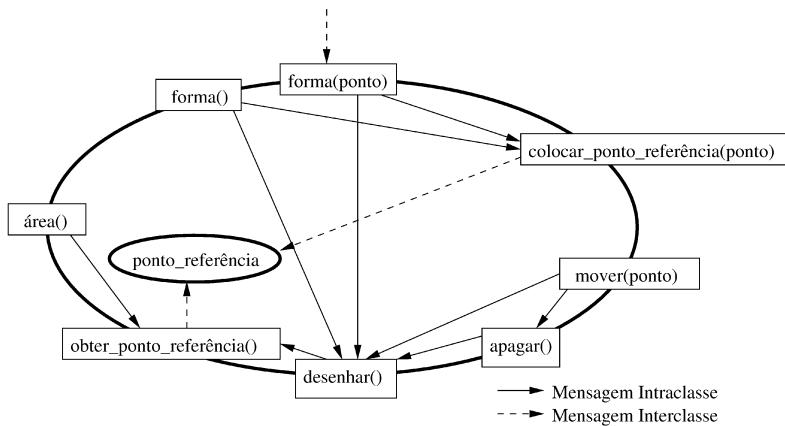


Figura 6.9 – Grafo de classe da classe Forma.

se que o método `desenhar()` só possui o conjunto de teste baseado em especificação e este não pode ser executado, pois não tem nenhuma implementação. Esse conjunto de teste só será utilizado após a criação de uma subclasse que implemente o método `desenhar()`. Se houver uma implementação inicial para o método `área()`, um conjunto de teste baseado em especificação e um conjunto de teste baseado em programa podem ser gerados e executados.

Conjuntos de teste baseados em especificação e em programa também foram desenvolvidos para o teste dos métodos `mover()`, `apagar()` e para os dois construtores de `Forma`. A vantagem de desenvolver conjuntos de teste baseado na especificação para os métodos `desenhar()` e `área()` é que esses conjuntos podem ser herdados na história do teste das subclasses.

Tabela 6.8 – História do teste da classe Forma [168]

| Atributo                              | Conj. de teste baseado em especificação | Conj. de teste baseado em implementação |
|---------------------------------------|---|---|
| Funções individuais                   |   |   |
| <code>colocar_ponto_referência</code> | (TF <sub>1</sub> ,S)                    | (TE <sub>1</sub> ,S)                    |
| <code>obter_ponto_referência</code>   | (TF <sub>2</sub> ,S)                    | (TE <sub>2</sub> ,S)                    |
| <code>mover</code>                    | (TF <sub>3</sub> ,S)                    | (TE <sub>3</sub> ,S)                    |
| <code>apagar</code>                   | (TF <sub>4</sub> ,S)                    | (TE <sub>4</sub> ,S)                    |
| <code>desenhar</code>                 | (TF <sub>5</sub> ,S)                    | —                                       |
| <code>área</code>                     | (TF <sub>6</sub> ,S)                    | (TE <sub>6</sub> ,S)                    |
| <code>Forma</code>                    | (TF <sub>7</sub> ,S)                    | (TE <sub>7</sub> ,S)                    |
| <code>Forma</code>                    | (TF <sub>8</sub> ,S)                    | (TE <sub>8</sub> ,S)                    |
| Interação entre funções               |   |   |
| <code>mover</code>                    | (TIF <sub>9</sub> ,S)                   | (TIE <sub>9</sub> ,S)                   |
| <code>apagar</code>                   | (TIF <sub>10</sub> ,S)                  | (TIE <sub>10</sub> ,S)                  |

Os métodos `mover()`, `apagar()`, `Forma()` e `Forma(Ponto)` chamam outros métodos: o método `mover()` chama os métodos `apagar()` e `desenhar()`; `apagar()` chama o método `desenhar()`; e tanto `Forma()` como `Forma(Ponto)` chamam os métodos `colocar_ponto_referência()` e `desenhar()`.



## Teste da subclasse

Testada a superclasse e armazenada a história do teste, dá-se início ao teste das subclasses. A idéia é reaproveitar tanto quanto possível os casos de teste utilizados no teste da superclasse. Tal reaproveitamento é obtido de acordo com o tipo de elemento herdado pela subclasse.

Dada uma superclasse  $P$ , a história do teste da subclasse é construída em função da história do teste da classe pai ( $H(P)$ ), do grafo da classe pai ( $G(P)$ ) e de um modificador ( $M$ ). O algoritmo *TestSubClass* (apresentado no Programa 6.4) detalha o processo de construção da história do teste da subclasse. Dados  $H(P)$ ,  $G(P)$  e  $M$ , o algoritmo produz  $H(R)$  (história do teste da subclasse  $R$ ) e  $G(R)$  (Grafo de Classe da subclasse  $R$ ).

As ações de *TestSubClass* dependem do tipo do elemento e das modificações que ele recebeu no mapeamento de herança. Para cada tipo de elemento (novo, recursivo, redefinido, virtual-novo, virtual-recursivo e virtual-redefinido) diferentes ações podem ocorrer.

Todo atributo (ou método) NOVO ou VIRTUAL-NOVO deve ser totalmente testado, já que foi definido em  $M$  e ainda não foi testado. Um atributo (ou método) RECURSIVO ou VIRTUAL-RECURSIVO requer um reteste limitado, pois já foi testado em  $P$  e tanto sua especificação quando sua implementação não foram alteradas. É necessário testar, por exemplo, o caso em que um método recursivo acessa um método redefinido. Um atributo (ou método) REDEFINIDO ou VIRTUAL-REDEFINIDO exige reteste, mas os casos de teste baseados em especificação podem ser reusados, se possível.

---

Programa 6.4

---

```
1 Algoritmo TestSubClass (H(P), G(P), M);
2   entrada: H(P): história de teste da classe P;
3           G(P): grafo da classe P;
4           M: modificador que especifica a subclasse R;
5   saída:   H(R): história de teste de R indicando
6           o que deve ser executado novamente;
7           G(R): grafo para a subclasse R;
8 begin
9   H(R) := H(P);      /* inicia a história de R a partir da de P*/
10  G(R) := G(P);     /* inicia o grafo da classe R a partir do de P*/
11  for each A contido em M do
12    case A seja NOVO ou VIRTUAL-NOVO:
13      Gerar TS, TP para A;
14      Adicionar { A, (TS, Y), (TP, Y) } a H(R);
15      Integrar A em G(R);
16      Gerar TIS e TIP para A ;
17      Adicionar {A, (TIS, Y), (TIP, Y)} a H(R);
18    case A seja RECURSIVO ou VIRTUAL-RECURSIVO:
19      if A acessa dados no escopo de R then
20        Identificar testes de interface reusáveis;
21        Adicionar {(TIS, P) e (TIP, P)} a H(R);
22    case A seja REDEFINIDO ou VIRTUAL-REDEFINIDO:
23      Gerar TP para A;
24      Reusar TS de P se ele existe ou Gerar TS para A;
25      Adicionar {A, (TS, Y), (TP, Y)} a H(R);
26      Integrar A em G(R);
27      Gerar TIP para G(R) em relação a A;
28      Reusar TIS de P;
29      Adicionar {A, (TIS, P), (TIP, P)} a H(R);
30 end TestSubClass.
```

---

Para ilustrar o funcionamento do algoritmo *TestSubClass*, considere o exemplo da classe Triângulo (Programa 6.5) que herda características da classe Forma (Programa 6.3).

---

Programa 6.5

---

```
1 class Triângulo: public Forma {
2     private:
3         point Vértice2;
4         point Vértice3;
5
6     public:
7         point obter_vértice1();           // novo
8         point obter_vértice2();           // novo
9         point obter_vértice3();           // novo
10        void colocar_vértice1();          // novo
11        void colocar_vértice2();          // novo
12        void colocar_vértice3();          // novo
13        void desenhar();                // virtual-redefinido
14        float área();                  // virtual-redefinido
15        triângulo();                  // novo
16        triângulo(Ponto, Ponto, Ponto); // novo
17 }
```

---

A história do teste da classe Triângulo, derivada a partir da história da classe Forma, é apresentada na Tabela 6.9.

Observa-se que nenhum dos conjuntos de teste para os métodos `colocar_ponto_referência()` e `obter_ponto_referência()` foi executado novamente por se tratar de métodos recursivos. Os métodos virtuais-redefinidos `desenhar()` e `área()` são retestados, visto que possuem uma nova implementação. Entretanto, os casos de teste baseados em especificação podem ser reutilizados. Para todos os métodos novos na classe Triângulo, conjuntos de teste baseados na especificação e em programa precisam ser desenvolvidos.

Três outros métodos foram adicionados à lista de métodos que interagem entre si: `área()` que chama `obter_vértice1()`, e `obter_vértice1()` e `colocar_vértice1()` que chamam os métodos `obter_ponto_referência()` e `colocar_ponto_referência()`, respectivamente. A interface entre esses pares de métodos deve ser testada e casos de teste adicionais devem ser desenvolvidos para testar individualmente os métodos `obter_ponto_referência()` e `colocar_ponto_referência()`. Na Tabela 6.9, os conjuntos de teste marcados com (‘) contêm somente casos de teste novos e os marcados com (‘‘) contêm tanto casos de teste que foram desenvolvidos quanto casos de teste reaproveitados da superclasse.

Para finalizar, uma nova classe TriânguloEquilátero foi desenvolvida. Essa nova classe herda características da classe Triângulo e não adiciona mais métodos, exceto os novos construtores. Entretanto, TriânguloEquilátero redefine a implementação do método `área()` para melhorar sua eficiência. Assim sendo, somente casos de teste baseados em implementação para o método `área()` foram gerados novamente e executados. Teste de integração entre os métodos novos e redefinidos também são reexecutados. A classe TriânguloEquilátero é mostrada no Programa 6.6 e a história do teste é apresentada na Tabela 6.10.

Tabela 6.9 – História do teste para a classe Triângulo [168]

| Atributo                 | Conj. de teste baseado em especificação | Conj. de teste baseado em implementação |
|--------------------------|---|---|
| Funções individuais      |   |   |
| colocar_ponto_referência | (TF <sub>1</sub> ,N)                    | (TE <sub>1</sub> ,N)                    |
| obter_ponto_referência   | (TF <sub>2</sub> ,N)                    | (TE <sub>2</sub> ,N)                    |
| mover                    | (TF <sub>3</sub> ,N)                    | (TE <sub>3</sub> ,N)                    |
| apagar                   | (TF <sub>4</sub> ,N)                    | (TE <sub>4</sub> ,N)                    |
| desenhar                 | (TF <sub>5</sub> ,S)                    | (TE <sub>4</sub> ,S)                    |
| área                     | (TF <sub>6</sub> ,S)                    | (TE <sub>5</sub> ,S)                    |
| Forma                    | (TF <sub>7</sub> ,N)                    | (TE <sub>7</sub> ,N)                    |
| Forma                    | (TF <sub>8</sub> ,N)                    | (TE <sub>8</sub> ,N)                    |
| obter_vértice1           | (TF' <sub>11</sub> ,S)                  | (TE' <sub>11</sub> ,S)                  |
| obter_vértice2           | (TF' <sub>12</sub> ,S)                  | (TE' <sub>12</sub> ,S)                  |
| obter_vértice3           | (TF' <sub>13</sub> ,S)                  | (TE' <sub>13</sub> ,S)                  |
| colocar_vértice1         | (TF' <sub>14</sub> ,S)                  | (TE' <sub>14</sub> ,S)                  |
| colocar_vértice2         | (TF' <sub>15</sub> ,S)                  | (TE' <sub>15</sub> ,S)                  |
| colocar_vértice3         | (TF' <sub>16</sub> ,S)                  | (TE' <sub>16</sub> ,S)                  |
| Triângulo                | (TF' <sub>17</sub> ,S)                  | (TE' <sub>17</sub> ,S)                  |
| Triângulo                | (TF' <sub>18</sub> ,S)                  | (TE' <sub>18</sub> ,S)                  |
| Interação entre funções  |   |   |
| mover                    | (TIF'' <sub>9</sub> ,P)                 | (TIE'' <sub>9</sub> ,P)                 |
| apagar                   | (TIF'' <sub>10</sub> ,P)                | (TIE'' <sub>10</sub> ,P)                |
| área                     | (TIF' <sub>19</sub> ,S)                 | (TIE' <sub>19</sub> ,S)                 |
| obter_vértice1           | (TIF' <sub>20</sub> ,S)                 | (TIE' <sub>20</sub> ,S)                 |
| colocar_vértice1         | (TIF' <sub>21</sub> ,S)                 | (TIE' <sub>21</sub> ,S)                 |

---



---

```

1   class TriânguloEquilátero: public Triângulo {
2       public:
3           float área();                                // redefinido
4           TriânguloEquilátero();                      // novo
5           TriânguloEquilátero(Ponto,Ponto,Ponto); // novo
6   }

```

---



---

## Avaliação da estratégia: um estudo de caso

Para avaliar os resultados obtidos com a utilização da estratégia incremental hierárquica, Harrold et al. apresentam um estudo de caso que utiliza a hierarquia de classes de uma biblioteca de interfaces gráfica (*Interviews*) implementada em C++. Essa hierarquia é composta pela superclasse *Interactor* e suas subclasses *Scene*, *MonoScene* e *Dialog*, sendo que: *Scene* é uma subclasse de *Interactor*, *MonoScene* é uma subclasse de *Scene* e *Dialog* é uma subclasse de *MonoScene*.

Com base nessa hierarquia de classes foi avaliada a porcentagem de reaproveitamento de casos de teste obtida se, em vez de sempre se retestarem todas as classes e subclasses, fosse utilizada a estratégia proposta. Os resultados são apresentados na Tabela 6.11.

Conforme destacado por Harrold et al.[168], o uso da Estratégia Incremental Hierárquica reduz significativamente o esforço de teste. Muitos métodos que devem ser testados de novo reutilizam os casos de teste baseados em especificação que foram desenvolvidos para suas superclasses. A reutilização de conjuntos de testes da superclasse resulta em uma economia

Tabela 6.10 – História do teste para a classe TriânguloEquilátero [168]

| Atributo                 | Conj. de teste baseado em especificação | Conj. de teste baseado em implementação |
|--------------------------|---|---|
| Funções individuais      |   |   |
| colocar_ponto_referência | (TF <sub>1</sub> ,N)                    | (TE <sub>1</sub> ,N)                    |
| obter_ponto_referência   | (TF <sub>2</sub> ,N)                    | (TE <sub>2</sub> ,N)                    |
| mover                    | (TF <sub>3</sub> ,N)                    | (TE <sub>3</sub> ,N)                    |
| apagar                   | (TF <sub>4</sub> ,N)                    | (TE <sub>4</sub> ,N)                    |
| desenhar                 | (TF <sub>5</sub> ,N)                    | (TE <sub>4</sub> ,N)                    |
| área                     | (TF <sub>6</sub> ,S)                    | (TE <sub>5</sub> ,S)                    |
| Forma                    | (TF <sub>7</sub> ,N)                    | (TE <sub>7</sub> ,N)                    |
| Forma                    | (TF <sub>8</sub> ,N)                    | (TE <sub>8</sub> ,N)                    |
| obter_vértice1           | (TF <sub>11</sub> ,N)                   | (TE <sub>11</sub> ,N)                   |
| obter_vértice2           | (TF <sub>12</sub> ,N)                   | (TE <sub>12</sub> ,N)                   |
| obter_vértice3           | (TF <sub>13</sub> ,N)                   | (TE <sub>13</sub> ,N)                   |
| colocar_vértice1         | (TF <sub>14</sub> ,N)                   | (TE <sub>14</sub> ,N)                   |
| colocar_vértice2         | (TF <sub>15</sub> ,N)                   | (TE <sub>15</sub> ,N)                   |
| colocar_vértice3         | (TF <sub>16</sub> ,N)                   | (TE <sub>16</sub> ,N)                   |
| Triângulo                | (TF <sub>17</sub> ,N)                   | (TE <sub>17</sub> ,N)                   |
| Triângulo                | (TF <sub>18</sub> ,N)                   | (TE <sub>18</sub> ,N)                   |
| Triângulo_equilátero     | (TF <sub>22</sub> ,S)                   | (TE <sub>22</sub> ,S)                   |
| Triângulo_equilátero     | (TF <sub>23</sub> ,S)                   | (TE <sub>23</sub> ,S)                   |
| Interação entre funções  |   |   |
| mover                    | (TIF'' <sub>9</sub> ,P)                 | (TIE'' <sub>9</sub> ,P)                 |
| apagar                   | (TIF'' <sub>10</sub> ,P)                | (TIE'' <sub>10</sub> ,P)                |
| área                     | (TIF'' <sub>19</sub> ,P)                | (TIE'' <sub>19</sub> ,P)                |

Tabela 6.11 – Reaproveitamento obtido no teste intraclasse [168]

| Classe     | Reteste de Todos os Métodos | Técnica Incremental | Razão entre as duas Técnicas |
|------------|-----------------------------|---------------------|------------------------------|
| Interactor | 93                          | 93                  | 100%                         |
| Scene      | 96                          | 36                  | 38%                          |
| MonoScene  | 99                          | 9                   | 9%                           |
| Dialog     | 103                         | 6                   | 6%                           |

tanto no tempo para analisar a classe e determinar quais métodos devem ser testados como no tempo gasto para executar os casos de testes. Com a estratégia proposta, o testador pode estimar os custos necessários para a realização dos teste e, com base nessa estimativa, concentrar os esforços nos pontos considerados mais problemáticos ou com maior probabilidade de conter defeitos.

### 6.5.3 Teste de mutação OO

Da mesma forma que os critérios de fluxo de dados, o teste de mutação também foi proposto inicialmente para o teste de unidade de programas procedimentais [115], mas já foram propostas extensões desse critério para o teste de integração de programas C [110] (vide Capítulo 5) e também para o teste de especificações em Máquinas de Estados Finitos [132, 133], Redes de Petri [134, 360], Statecharts [130], Estelle [331, 374], SDL [380] e Especificações Algébricas [442]. É importante ressaltar: a essência do critério é a mesma, ou seja, realizar alterações sintáticas no produto que está sendo testado produzindo-se um conjunto de produtos mutantes e desenvolver um conjunto de teste que mostre que o conjunto de mutantes não

se comporta corretamente em relação à especificação. Feito isso, evidencia-se que o produto em teste não apresenta os defeitos representados pelo conjunto de mutantes.

A flexibilidade de extensão do critério vem do fato de que para se aplicar o teste de mutação é necessária a existência de um modelo executável e que aceite uma entrada e produza uma saída possível de ser comparada com a saída do mutante. Além disso, é necessária a definição de um conjunto de operadores de mutação responsável pela representação do modelo de defeitos correspondente à entidade executável em questão.

Atualmente, vários pesquisadores têm trabalhado na definição de operadores de mutação para programas OO. Entre eles destacam-se os trabalhos de Kim et al. [212, 214], que definem um conjunto de operadores de mutação de classes, mas que não exploram todas as características de um programa OO; de Bierman et al. [34], que definem um conjunto de operadores de mutação específico para o teste de determinadas classes da API (*Application Program Interface*) de Java; e de Ma et al. [257], os quais propõem um conjunto de operadores de mutação de classe para Java que incluem os operadores propostos por Kim et al. [212, 214]. Entretanto, todos os conjuntos de mutação propostos para OO foram desenvolvidos para a linguagem Java e focam somente o teste de classe, não levando em consideração o teste intramétodo e intermétodo da forma como é realizado pelos critérios Análise de Mutantes e Mutação de Interface em programas procedimentais.

Os operadores de mutação são dependentes da linguagem de programação; no contexto de programas OO, as linguagens Java e C++ são as que concentram maior número de trabalhos relacionados. Dada a semelhança de algumas construções sintáticas dessas linguagens com a linguagem C, Vincenzi [413] avaliou a aplicabilidade de operadores de mutação desenvolvidos para C no contexto de Java e C++, ou seja, o objetivo foi avaliar quais defeitos modelados pelos operadores de mutação desenvolvidos para C também modelavam defeitos que poderiam ocorrer em programas OO.

A seguir, os trabalhos citados são descritos de forma sucinta. Inicialmente, são identificados operadores de mutação que podem ser utilizados para o teste intramétodo. Em seguida, são identificados operadores de mutação que podem ser utilizados no teste intermétodo e, posteriormente, são identificados os operadores de mutação destinados ao teste interclasses.

## Teste intramétodo

O teste intramétodo visa avaliar se a funcionalidade de um método está implementada corretamente, identificando defeitos de lógica e programação. Observa-se que Java e C++ possuem diversas construções sintáticas comuns à linguagem C, principalmente no nível de unidade. Com isso, o conjunto de operadores de mutação definido para a linguagem C [7] foi avaliado de modo a identificar quais deles poderiam ser utilizados no contexto de programas OO escritos em C++ e Java. A análise realizada resultou nos dados mostrados na Tabela 6.12.

Na primeira coluna da Tabela 6.12 estão listados os nomes dos 80 operadores de mutação para o teste de unidade de programas C, extraídos de Agrawal et al. [7]. Esses operadores estão divididos em quatro classes: mutação de constantes (5 operadores – Tabela 6.12(a)), mutação de comandos (16 operadores – Tabela 6.12(b)), mutação de variáveis (12 operadores – Tabela 6.12(c)) e mutação de operadores (47 operadores – Tabela 6.12(d)).

Cada conjunto descreve operadores que modelam diferentes tipos de defeitos em um mesmo tipo de estrutura, e seus nomes são descritos por quatro letras; a primeira (em maiús-



pelos mesmos operadores, bem como uma descrição detalhada do significado do nome de cada um deles, podem ser consultados os trabalhos de Agrawal et al. [7] e Vincenzi [413].

Tabela 6.13 – Significado do nome de alguns operadores de mutação de unidade para o teste intramétodo [413]

| Operador | Significado                              | Descrição   |
|----------|--|---|
| u-CGCR   | Constant for Global Constant Replacement | Aplica-se a cada constante global do programa, trocando-a por todas as outras constantes globais que aparecem na mesma função.                        |
| u-CLCR   | Constant for Local Constant Replacement  | Aplica-se a cada constante local do programa, trocando-a por todas as outras constantes locais que aparecem na mesma função.                          |
| u-CGSR   | Constant for Global Scalar Replacement   | Aplica-se a cada referência escalar global do programa, trocando-a por todas as constantes globais que aparecem na mesma função.                      |
| u-CLSR   | Constant for Local Scalar Replacement    | Aplica-se a cada referência escalar local do programa, trocando-a por todas as constantes locais que aparecem na mesma função.                        |
| u-VDTR   | Domain Traps                             | Aplica-se a cada variável escalar, envolvendo a mesma por uma “função armadilha” que testa se a variável assume os valores negativo, zero e positivo. |
| u-VTWD   | Twiddle Mutations                        | Aplica-se a cada variável escalar, alterando o seu valor uma unidade a mais e a menos.  |
| u-OLBN   | Logical Operator by Bitwise Operator     | Substitui um operador lógico por um operador <i>bitwise</i> sem atribuição.   |
| u-ORRN   | Relational Operator Mutation             | Substitui um operador relacional por outro operador relacional.   |
| u-SMTC   | n-trip continue                          | Introduz no início de cada laço uma chamada à função que, quando executada 2 vezes, interrompe a execução do laço.                                    |
| u-SSDL   | Statement Deletion                       | Apaga comandos sistematicamente, um a um, mas mantém o ponto-e-vírgula no final do comando para manter a validade sintática do programa.              |
| u-SWDD   | while Replacement by do-while            | Substitui comandos <code>while</code> por comandos <code>do-while</code>  |

O símbolo ( $\checkmark$ ) é utilizado para indicar quando um operador de mutação é aplicável para a linguagem em questão, o símbolo ( $\times$ ), quando o operador não é aplicável, e o símbolo ( $\odot$ ) indica os operadores que podem ser aplicados com alguma modificação semântica. O símbolo (\*) antes do nome do operador indica os operadores novos que foram definidos por Vincenzi [413]. Tendo em vista que no caso da linguagem C existem operadores de mutação específicos para realizar mutações em variáveis locais e globais, uma observação é necessária para definir como esses conceitos de variáveis globais e locais foram mapeados para o teste intramétodo de linguagens OO.

No caso de C++, por ser um superconjunto de C, o conjunto de variáveis globais é composto por variáveis globais tradicionais (como conhecidas em C) e pelos atributos de uma classe, uma vez que esses atributos podem ser compartilhados entre os diversos métodos de uma mesma classe como se fossem variáveis globais da classe. Já o conjunto de variáveis locais é definido como as variáveis declaradas dentro do método, da mesma forma como as variáveis locais em C que são aquelas declaradas dentro de uma função.

No caso de Java, não existe o conceito de variável global como na linguagem C, ou seja, em Java existem somente atributos e variáveis locais. Toda variável em Java deve obrigatoriamente ser declarada dentro de uma classe. Assim sendo, o conjunto de variáveis globais para

a linguagem Java é composto apenas dos atributos de uma classe e o conjunto de variáveis locais é composto das variáveis declaradas dentro de um método.

Feitas tais considerações, como pode ser observado na Tabela 6.12, todos os 80 operadores de unidade de C são aplicáveis no teste intramétodo de programas C++, o que é natural, uma vez que C++ é, na verdade, um superconjunto de C.

Para o teste de programas Java, observa-se que a maioria dos operadores também é diretamente aplicável (59 dos 80 operadores de mutação). Os demais 21 operadores, analisados em seqüência, são pertencentes às seguintes classes de mutação:

**Mutação de comandos:** 1 operador (u-SGLR);

**Mutação de variáveis:** 5 operadores (u-VGPR, u-VGTR, u-VLPR, u-VLTR e u-VSCR);

**Mutação de operadores:** 15 operadores (u-OALN, u-OARN, u-OBLN, u-OBRN, u-OIPM, u-OLAN, u-OLBN, u-OLRN, u-OLSN, u-ORAN, u-ORB, u-ORLN, u-ORSN, u-OSLN, u-OSRN).

- Operadores de Mutação de Comandos

O operador de mutação de comando u-SGLR é responsável pela troca de rótulos de comandos `goto`. Como Java não possui comandos `goto`, esse operador não é diretamente aplicável a programas Java.

Entretanto, Java possui comandos `break` e `continue` com rótulos, comandos estes não presentes em C e C++. Desse modo, é possível mapear o conceito desse operador para o teste intramétodo de programas Java. Com isso, quatro outros operadores foram definidos:

- u-SBLR – `break Label Replacement`: tem por objetivo substituir comandos `break` e comandos `break` rotulados (*labeled break*) por todos os demais comandos `break` e `break` rotulados válidos dentro de um bloco;
- u-SCLR – `continue Label Replacement`: tem por objetivo substituir comandos `continue` e comandos `continue` rotulados (*labeled continue*) por todos os demais comandos `continue` e `continue` rotulados válidos dentro de um bloco;
- u-SLBC – `Labeled break By continue's Label Replacement`: troca o rótulo de cada comando `break` rotulado por todos os demais rótulos de comandos `continue` válidos dentro de um bloco;
- u-SLCB – `Labeled continue By break's Label Replacement`: troca o rótulo de cada comando `continue` rotulado por todos os demais rótulos de comandos `break` válidos dentro de um bloco.

Como pode ser observado na Tabela 6.12, esses quatro novos operadores são específicos para o teste de programas Java, não sendo utilizados no teste de programas C++.

- Operadores de Mutação de Variáveis

Considerando os operadores de mutação de variáveis, observa-se que cinco (u-VGPR, u-VGTR, u-VLPR, u-VLTR e u-VSCR) não são diretamente aplicáveis a Java. A justificativa é que esses operadores são específicos para a realização de mutação de ponteiros e registros (*struct*) existentes em C e C++, mas não presentes em Java.

Entretanto, o conceito desses operadores pode ser mapeado para Java. No caso dos operadores u-VGPR e u-VLPR, embora Java não tenha variáveis tipo ponteiro, ela apresenta variáveis tipo referência; portanto, operadores de mutação específicos para realizar mutações em variáveis tipo referência podem ser desenvolvidos. Já no caso dos operadores u-VGTR, u-VLTR e u-VSCR, os quais se referem a `struct`, embora Java não apresente variáveis tipo estrutura, ela apresenta o conceito de classe (a qual pode ser vista como uma estrutura que contém os dados e funções de acesso a esses dados) de modo que o conceito desses operadores também pode ser mapeado para Java.

A partir desses operadores, como pode ser observado na Tabela 6.12, três novos foram definidos:

- u-VGCR – Mutate Global Class References: foi definido a partir dos operadores u-VGPR e u-VGTR e é aplicado em variáveis de referência globais. Nesse operador, os tipos das variáveis são preservados de modo que somente variáveis de referência globais de tipos compatíveis são mutadas.
- u-VLCR – Mutate Local Class References: foi definido a partir dos operadores u-VLPR e u-VLTR e é aplicado em variáveis de referência locais. Nesse operador, os tipos da variáveis são preservados de modo que somente variáveis de referência locais de tipos compatíveis são mutadas.
- u-VCAR – Class Attributes Replacement: foi definido tendo como base o operador u-VSCR e é responsável pela mutação de referências a atributos de uma classe que são substituídas por referências aos demais atributos da mesma classe, respeitando os tipos dos atributos.

Observa-se que tais operadores podem ser também utilizados no contexto de C++, uma vez que as estruturas sintáticas exigidas por esses operadores também estão presentes nessa linguagem.

- Operadores de Mutação de Operadores

Observa-se que os 15 operadores de mutação de unidade de C que não são aplicáveis a Java pertencem à classe de mutação de operadores. A razão para isso é que, ao contrário de C e C++, toda expressão lógica em Java é do tipo booleana. Essa característica reduz a ocorrência de enganos, tal como a utilização do símbolo “=” em vez de “==” no teste de igualdade, limitando o número de operadores de mutação que podem ser utilizados em expressões lógicas.

Com a definição desses novos operadores, considera-se que os desvios sintáticos mais comuns cometidos durante a codificação de métodos em Java e C++ estão sendo modelados. Java passa a ter  $59 + 7 = 66$  operadores e C++ passa a ter  $80 + 3 = 83$  operadores destinados ao teste intramétodo.

Obviamente, quanto maior o número de operadores de mutação, mais mutantes são gerados e, consequentemente, maior é o custo para a aplicação do critério no que se refere ao tempo de execução e análise dos mutantes vivos. Como foi mencionado anteriormente, além de estudos que buscam a redução do custo de aplicação do teste de mutação por meio da utilização de critérios alternativos como a Mutação Aleatória (*Randomly Selected X% Mutation*) [3], a Mutação Restrita (*Constrained Mutation*) [272] e a Mutação Seletiva (*Selective*

*Mutation*) [313], destacam-se também os estudos que objetivam determinar Conjuntos Essenciais de Operadores de Mutação [310, 438, 25] e os estudos que objetivam auxiliar na determinação de mutantes equivalentes [164, 182, 306, 311, 421].

Um ponto importante a ser observado é que, entre os operadores de mutação considerados essenciais para o teste de unidade em programas C (u-SWDD, u-SMTC, u-SSDL, u-OLBN, u-ORRN, u-VTWD, u-VDTR, u-CGCR, u-CLCR, u-CGSR e u-CLSR), todos eles, exceto o u-OLBN no caso de Java, são também aplicáveis em programas Java e C++, o que motiva a investigar se a mesma relação de inclusão entre esses operadores se mantém no teste de programas OO [413].

## Teste intermétodo

Ao contrário do teste intramétodo, o teste intermétodo deseja descobrir defeitos nas interfaces de comunicação entre os métodos. Ele pode ser considerado similar ao teste de integração em programas procedimentais. Conforme apresentado no Capítulo 5, Delamaro et al. [110] propuseram o critério Mutação de Interface que pode ser visto como uma extensão da Análise de Mutantes e preocupa-se em assegurar que as interações entre unidades sejam testadas. Assim, o objetivo do critério Mutação de Interface é inserir perturbações nas conexões entre duas unidades.

Observa-se que os conceitos do critério Mutação de Interface podem ser facilmente mapeados para o contexto OO, considerando o teste intermétodos. Basicamente, uma vez que, para realizar as mutações, a grande maioria dos operadores de interface baseia-se nos conjuntos de parâmetros, variáveis e constantes relacionados à conexão de duas unidades  $f-g$ , conforme apresentado na Seção 5.6 do Capítulo 5, basta redefinir tais conjuntos considerando as linguagens OO em questão, Java e C++.

Novamente, no caso de C++, tudo o que é válido em C continua sendo válido e resta apenas adicionar ao conjunto de variáveis globais  $G(g)$  os atributos da classe utilizados pela função (método)  $g$ ; no caso do conjunto de variáveis externas  $E(g)$ , devem-se adicionar os atributos da classe não utilizados pela função (método)  $g$ .

Já para a linguagem Java, as variáveis globais podem ser vistas como os atributos de uma classe ou atributos estáticos e, desse modo, supondo que a mutação será aplicada na conexão entre dois métodos  $f$  e  $g$ , a definição dos conjuntos para Java ficaria sendo:

$P(g)$ : é o conjunto dos parâmetros formais de  $g$ .

$G(g)$ : é o conjunto de variáveis globais utilizadas no método  $g$ .

$L(g)$ : é o conjunto de variáveis declaradas em  $g$  (variáveis locais).

$E(g)$ : é o conjunto de variáveis globais não utilizadas em  $g$ .

$C(g)$ : é o conjunto de constantes utilizadas em  $g$ .

Além disso, o conjunto  $R$  de “constantes requeridas” também é alterado, como mostrado na Tabela 6.14.

Apresentadas tais definições, da mesma forma como no teste intramétodo, fazendo-se uma análise dos operadores de mutação definidos por Delamaro et al. [110], a

Tabela 6.15 indica qual é a possibilidade de aplicação de cada um considerando as linguagens Java e C++.

Como pode ser observado, por apresentar um nível de abstração maior, todos os 33 operadores de mutação de interface podem ser utilizados para viabilizar o teste intermétodo em programas OO, considerando as linguagens C++ e Java. Um ponto importante destacado por Binder [35] e Ma et al. [257] é que, em geral, a complexidade dos métodos em programas OO não é muito grande. A idéia é desenvolver métodos simples que, pela interação com outros métodos da mesma ou de outras classes, implementem a funcionalidade desejada. Nesse sentido, o teste de intermétodos é de fundamental importância no teste de programas OO. Nesse sentido, considera-se que tal conjunto de operadores de mutação venha a trazer importantes contribuições no teste de programas OO, restando agora a realização de estudos empíricos para verificar os aspectos de custo, eficácia e dificuldade de satisfação desses operadores.

Tabela 6.14 – Conjunto de constantes requeridas para Java

| Tipo de variável | Constantes requeridas  |
|------------------|--|
| byte             | -1, 1, 0, Byte.MAX_VALUE, Byte.MIN_VALUE                     |
| short            | -1, 1, 0, Short.MAX_VALUE, Short.MIN_VALUE                   |
| int              | -1, 1, 0, Integer.MAX_VALUE, Integer.MIN_VALUE               |
| long             | -1L, 1L, 0L, Long.MAX_VALUE, Long.MIN_VALUE                  |
| float            | -1.0f, 1.0f, 0.0f, -0.0f, Float.MAX_VALUE, Float.MIN_VALUE   |
| double           | -1.0d, 1.0d, 0.0d, -0.0d, Double.MAX_VALUE, Double.MIN_VALUE |
| char             | 1, Character.MAX_VALUE, Character.MIN_VALUE                  |
| boolean          | true, false  |
| String           | "", null   |
| Outros objetos   | null   |

<tipo>.MAX\_VALUE, <tipo>.MIN\_VALUE correspondem respectivamente ao maior valor positivo e ao menor valor negativo do tipo de dado representado por <tipo>. Ao contrário de C e C++, em Java esses valores não são dependentes da plataforma.

Outro ponto importante a ser observado é que, da mesma forma como foi determinado o conjunto de operadores essenciais para o teste de unidade, Vincenzi et al. [417, 418] realizaram um experimento aplicando o procedimento *Essencial* [25] ao conjunto de operadores de mutação de interface. Para isso, consideraram os dados coletados para cinco programas C utilitários do UNIX. Desse experimento, Vincenzi et al. [418] chegaram a um subconjunto de oito operadores (I-CovAllNod, I-DirVarBitNeg, I-IndVarBitNeg, I-IndVarRepExt, I-IndVarRepGlo, I-IndVarRepLoc, I-IndVarRepReq e II-ArgAriNeg), os quais compõem o conjunto essencial de operadores de mutação de interface. Para o experimento em questão, a utilização desse subconjunto resultou em uma redução de custo do critério Mutação de Interface superior a 73%, mantendo-se um escore de mutação em relação ao conjunto total de operadores de interface na ordem de 0,998 [418]. Tais resultados motivam reavaliar se os mesmos resultados se confirmam no teste de programas OO. A Tabela 6.16 apresenta a descrição dos oito operadores essenciais. Para mais informações sobre os tipos de mutações realizadas pelos mesmos operadores, bem como uma descrição detalhada do significado do nome de cada um deles, pode ser consultado o trabalho de Delamaro [108].

Tabela 6.15 – Operadores de mutação de interface para o teste intermétodo

| Grupo-I        |       |       |
|----------------|-------|-------|
| Operador       | Java  | C++   |
| I-CovAllEdg    | ✓     | ✓     |
| I-CovAllNod    | ✓     | ✓     |
| I-DirVarAriNeg | ✓     | ✓     |
| I-DirVarBitNeg | ✓     | ✓     |
| I-DirVarIncDec | ✓     | ✓     |
| I-DirVarLogNeg | ✓     | ✓     |
| I-DirVarRepCon | ✓     | ✓     |
| I-DirVarRepExt | ✓     | ✓     |
| I-DirVarRepGlo | ✓     | ✓     |
| I-DirVarRepLoc | ✓     | ✓     |
| I-DirVarRepPar | ✓     | ✓     |
| I-DirVarRepReq | ✓     | ✓     |
| I-IndVarAriNeg | ✓     | ✓     |
| I-IndVarBitNeg | ✓     | ✓     |
| I-IndVarIncDec | ✓     | ✓     |
| I-IndVarLogNeg | ✓     | ✓     |
| I-IndVarRepCon | ✓     | ✓     |
| I-IndVarRepExt | ✓     | ✓     |
| I-IndVarRepGlo | ✓     | ✓     |
| I-IndVarRepLoc | ✓     | ✓     |
| I-IndVarRepPar | ✓     | ✓     |
| I-IndVarRepReq | ✓     | ✓     |
| I-RetStaDel    | ✓     | ✓     |
| I-RetStaRep    | ✓     | ✓     |
| Total          | 24/24 | 24/24 |

(a)

| Grupo-II     |      |     |
|--------------|------|-----|
| Operador     | Java | C++ |
| II-ArgAriNeg | ✓    | ✓   |
| II-ArgBitNeg | ✓    | ✓   |
| II-ArgDel    | ✓    | ✓   |
| II-ArgIncDec | ✓    | ✓   |
| II-ArgLogNeg | ✓    | ✓   |
| II-ArgRepReq | ✓    | ✓   |
| II-ArgSwiAli | ✓    | ✓   |
| II-ArgSwiDif | ✓    | ✓   |
| II-FunCalDel | ✓    | ✓   |
| Total        | 9/9  | 9/9 |

(b)

| Legenda                                   |
|---|
| ✓: operadores aplicáveis                  |
| ✗: operadores não-aplicáveis              |
| ◎: operadores aplicáveis com modificações |
| *: operadores definidos                   |

Tabela 6.16 – Significado do nome de alguns operadores de mutação de unidade para o teste intermétodo [413]

| Operador       | Significado  | Descrição  |
|----------------|--|--|
| I-CovAllNod    | Coverage of Nodes                                      | Garante cobertura de nós de determinada função.              |
| I-DirVarBitNeg | Inserts Bit Negation at Interface Variables            | Acrescenta negação de bit em variáveis de interface.         |
| I-IndVarBitNeg | Inserts Bit Negation at Interface Variables            | Acrescenta negação de bit em variáveis não de interface.     |
| I-IndVarRepExt | Replaces Non Interface Variables by Extern Global      | Troca variáveis não de interface por elementos de <i>E</i> . |
| I-IndVarRepGlo | Replaces Non Interface Variables by Global Variables   | Troca variáveis não de interface por elementos de <i>G</i> . |
| I-IndVarRepLoc | Replaces Non Interface Variables by Local Variables    | Troca variáveis não de interface por elementos de <i>L</i> . |
| I-IndVarRepReq | Replaces Non Interface Variables by Required Constants | Troca variáveis não de interface por elementos de <i>R</i> . |
| II-ArgAriNeg   | Insert Arithmetic Negation on Argument                 | Acrescenta negação aritmética antes de argumento.            |

## Teste interclasse

Conforme definido por Ma et al. [257], no teste interclasse o objetivo é identificar defeitos relacionados a características específicas de programas OO, tais como encapsulamento, herança, polimorfismo e acoplamento dinâmico.

O principal problema nesse contexto é a identificação de um modelo de defeitos adequado a partir do qual os operadores de mutação possam ser derivados. Ma et al. [257] propuseram 24 operadores de mutação para o teste interclasse de programas Java com base nos modelos de defeitos propostos por Offutt et al. [316], Chevalley [75] e Kim et al. [212, 214]. Juntos, esses três modelos incluem os seguintes tipos de defeitos [257]:

- Defeitos identificadas por Offutt et al. [316]:
  - Anomalia na visibilidade do estado
  - Inconsistência na definição do estado (devido à ocultação de variável de estado)
  - Anomalia na definição do estado (devido a sobrecarga)
  - Definição indireta de estado inconsistente
  - Comportamento de construção anômala
  - Construção incompleta
  - Uso de tipo inconsistente
- Defeitos identificados por Kim et al. [212, 214]:
  - Uso incorreto de sobrecarga de métodos
  - Uso incorreto de modificador de acesso
  - Uso incorreto do modificador `static`
- Defeitos identificados por Chevalley [75]:
  - Implementação de métodos sobre carregados incorreta
  - Uso incorreto da palavra reservada `super`
  - Uso incorreto da palavra reservada `this`
  - Defeitos provenientes de enganos de programação comuns

Os operadores de mutação interclasse de Ma et al. [257] estão divididos em seis grupos:

1. Ocultação de informação (*Information Hiding – Access Control*) – 1 operador;
2. Herança (*Inheritance*) – 7 operadores;
3. Polimorfismo (*Polymorphism*) – 4 operadores;
4. Sobrecarga (*Overloading*) – 4 operadores;
5. Características específicas de Java (*Java-Specific Features*) – 4 operadores;
6. Enganos comuns de programação (*Common Programming Mistakes*) – 4 operadores.

Os quatro primeiros estão baseados em características comuns à maioria das linguagens OO. O quinto grupo inclui características específicas da linguagem Java e o último grupo é formado por operadores que representam enganos comuns quando se programa utilizando o paradigma OO. A Tabela 6.17 mostra o conjunto completo de operadores e a possibilidade de

aplicação deles também na linguagem C++. A primeira letra do nome do operador identifica a qual grupo ele pertence: A – Access Control, I – Inheritance, P – Polymorphism, O – Overloading, J – Java-Specific Features e E – Common Programming Mistakes.

Como observado por Vincenzi [413], os operadores específicos para a linguagem Java não são aplicados a C++, mas todos os demais operadores podem ser mapeados para C++. Um ponto a ser observado é que C++ é muito mais flexível do que Java e um conjunto de operadores de mutação adequado ao teste de programas C++ deve ser muito mais abrangente do que esse. Por exemplo, entre algumas características de C++ não contempladas por esses operadores estão: herança múltipla, sobrecarga de operadores, funções *friends* e *templates* [94].

Tabela 6.17 – Operadores de mutação para o teste interclasse

| Operador | Java | C++ |
|----------|------|-----|
| AMC      | ✓    | ✓   |
| IHD      | ✓    | ✓   |
| IHI      | ✓    | ✓   |
| IOD      | ✓    | ✓   |
| IOP      | ✓    | ✓   |
| IOR      | ✓    | ✓   |
| ISK      | ✓    | ✓   |
| IPC      | ✓    | ✓   |
| PNC      | ✓    | ✓   |
| PMD      | ✓    | ✓   |
| PPD      | ✓    | ✓   |
| PRV      | ✓    | ✓   |
| OMR      | ✓    | ✓   |
| OMD      | ✓    | ✓   |
| OAO      | ✓    | ✓   |
| OAN      | ✓    | ✓   |
| JTD      | ✓    | ✗   |
| JSC      | ✓    | ✗   |
| JID      | ✓    | ✗   |
| JDC      | ✓    | ✗   |
| EOA      | ✓    | ✓   |
| EOC      | ✓    | ✓   |
| EAM      | ✓    | ✓   |
| EMM      | ✓    | ✓   |

| Legenda                                    |
|--|
| ✓ – operadores aplicáveis                  |
| ✗ – operadores não-aplicáveis              |
| ○ – operadores aplicáveis com modificações |
| * – operadores definidos                   |

Ma et al. [257] denominam o conjunto de operadores de mutação apresentado na Tabela 6.17 como sendo um conjunto abrangente (*comprehensive set*) em oposição aos definidos por Kim et al. [212, 214], que visaram à definição de um conjunto seletivo (ou essencial). A definição de um conjunto abrangente, embora leve a um aumento no custo de aplicação do critério, reduz a chance de que tipos de defeitos importantes deixem de ser contemplados, aumentando a eficácia do critério. Uma vez definido o conjunto mais abrangente, estudos empíricos podem ser realizados objetivando a determinação de um conjunto essencial. Essa é a idéia utilizada no desenvolvimento do procedimento *Essencial* [25] e motiva a investigar sua aplicação, buscando a determinação do conjunto essencial de operadores de mutação

de classe. Na Tabela 6.18 é apresentada uma breve descrição de alguns dos operadores de mutação de classe da Tabela 6.17. Para obter mais informações sobre os tipos de mutações realizadas, bem como uma descrição detalhada do significado do nome de cada um deles, pode ser consultado o trabalho de Ma et al. [257].

Tabela 6.18 – Significado do nome de alguns operadores de mutação de classe [257]

| Operador | Significado   | Descrição  |
|----------|---|--|
| AMC      | Access modifier change                                  | Substitui um modificador de acesso por outros modificadores.   |
| IHI      | Hiding variable insertion                               | Adiciona a declaração de um atributo de mesmo nome e tipo da superclasse.                                  |
| PMD      | Instance variable declaration with parent class type    | Troca a declaração de tipo de um objeto pela declaração de tipo da superclasse.                            |
| OMD      | Overloading method deletion                             | Remove a declaração de um método sobrecarregado.   |
| OAO      | Argument order change                                   | Troca a ordem dos argumentos nas expressões de invocação dos métodos.                                      |
| JTD      | this keyword deletion                                   | Remove palavra reservada <code>this</code> .   |
| EOA      | Reference assignment and content assignment replacement | Substitui uma atribuição por referência por uma atribuição a uma referência “clonada”.                     |
| EOC      | Reference comparison and content comparison replacement | Troca uma comparação de referências por uma comparação de conteúdo e vice-versa.                           |
| EAM      | Accessor method change                                  | Troca nome de método de acesso ( <code>get</code> ) por outro nome de método de acesso de tipo compatível. |

Além dos operadores para o teste interclasse de programas Java apresentados na Tabela 6.17, outros conjuntos de operadores de mutação para o teste de características mais específicas da linguagem vêm sendo propostos, como, por exemplo, os operadores para o teste de classes que implementam as interfaces `Iterator`, `Collection` e a classe abstrata `InputStream` propostos por Alexander et al. [12].

## 6.6 Teste de componentes

Como definido por Szyperski [381], um componente de software é uma unidade de composição de sistemas com especificações contratuais de interfaces e explícita dependência de contexto. Um componente de software pode ser desenvolvido independente e ser utilizado por terceiros para composição. Componentes de software existem em diferentes formas. Em geral, um componente de software pode ser tão simples como uma única classe ou tão sofisticado como um JavaBean, Enterprise JavaBean [271] ou objetos COM [283]. Pode ser observado que componentes de software herdam muitas das características do paradigma de programação OO, mas a noção de componentes transcende a noção de objetos. Reuso em programação OO, em geral, representa reuso de bibliotecas de classes considerando uma linguagem de programação específica. Componentes de software podem ser reutilizados sem conhecimento da linguagem de programação ou ambiente no qual foram desenvolvidos, caracterizando uma forma mais genérica de reuso [153, 428]. Assim sendo, o desenvolvimento de sistemas baseados em componentes, os quais, em geral, fazem uso extensivo das características de programação OO, tem motivado uma reavaliação das estratégias de teste quanto à adequação nesse novo contexto.

Conforme será descrito nesta seção, o teste de componentes e de sistemas baseados em componentes envolve uma série de questões. Como destacam Harrold et al. [167], é possível

analisar a questão do teste de sistemas baseados em componentes de duas perspectivas: a perspectiva do cliente e a perspectiva do desenvolvedor.

## Perspectiva do cliente

Os clientes são aqueles que desenvolvem sistemas, integrando em suas aplicações componentes desenvolvidos independentemente. Para auxiliá-los, existem diversas iniciativas de se adaptarem técnicas de análise e teste destinadas a programas tradicionais para o contexto de sistemas baseados em componentes. Entretanto, existem algumas questões que dificultam a adaptação de tais técnicas.

Primeiro, o código-fonte dos componentes em geral não é disponibilizado para os seus clientes. Técnicas e critérios de teste baseados na implementação, tais como critérios baseados em análise de fluxo de dados e critérios baseados em mutação necessitam do código-fonte para derivar os requisitos de teste. Quando o código-fonte do componente não está disponível para o cliente, as técnicas de teste tradicionais não podem ser aplicadas no teste de sistemas baseados em componentes ou pelo menos será requerido um esquema alternativo estabelecido entre as partes interessadas.

Segundo, em sistemas baseados em componentes, mesmo que o código-fonte esteja disponível, os componentes e a aplicação do cliente podem ter sido implementados em diferentes linguagens de programação. Desse modo, uma ferramenta de análise ou de teste baseada em uma linguagem de implementação específica não seria adequada para testar um programa que envolve outra linguagem, além daquela apoiada pela ferramenta.

Terceiro, um componente de software freqüentemente oferece mais funcionalidades do que a aplicação do cliente necessita. Com isso, sem a identificação da parte da funcionalidade que é utilizada pela aplicação, uma ferramenta de teste irá fornecer relatórios imprecisos. Por exemplo, critérios de teste estruturais avaliam o quanto determinado conjunto de teste é adequado em cobrir os requisitos de teste exigidos pelo critério (elementos estruturais do programa). Quando se deseja avaliar a adequação de um determinado conjunto de teste em relação a um sistema baseado em componentes, os elementos estruturais que compreendem a parte não utilizada do componente devem ser excluídos da avaliação. Caso contrário, uma ferramenta de teste irá produzir relatórios indicando baixa cobertura para o conjunto de teste, mesmo se tal conjunto teste exaustivamente a parte do código que está sendo utilizada [348].

## Perspectiva do desenvolvedor

O desenvolvedor implementa e testa o componente de software independentemente da aplicação que fará uso de tal componente. Ao contrário do cliente, o desenvolvedor tem acesso ao código-fonte. Assim, testar o componente para o desenvolvedor é similar ao teste de unidade/integração tradicional. Entretanto, critérios tradicionais, tais como cobertura de comandos e de desvios condicionais, podem não ser suficientes para o teste de componentes devido à baixa capacidade de detecção de defeitos desses critérios [400]. Corrigir um defeito em um componente depois que esse já está no mercado, em geral, envolve um custo de correção muitas vezes maior do que se o mesmo defeito fosse descoberto durante o teste de integração de um sistema não baseado em componentes, pois um componente pode estar sendo utilizado por muitas aplicações.

A seguir são descritos alguns critérios de teste funcional e algumas estratégias que podem ser utilizadas no teste de componentes. Em seguida, são discutidas algumas alternativas que tentam viabilizar o uso de critérios estruturais no teste de componentes.

### 6.6.1 Estratégias e critérios de teste

Como pode ser observado, critérios funcionais [31] são aplicados diretamente no teste de programas procedimentais, orientados a objetos e componentes de software, visto que derivam os seus requisitos de teste somente com base na especificação do programa/componente. Entretanto, o maior problema com os critérios da técnica funcional é que, por serem baseados na especificação, eles não são capazes de garantir que partes essenciais ou críticas da implementação tenham sido cobertas pelo conjunto de teste.

Uma vez que os critérios da técnica estrutural e os critérios baseados em defeitos, em geral, requerem a disponibilidade do código-fonte para serem aplicados, critérios alternativos, que não apresentam tal restrição vêm sendo propostos. Tais critérios são baseados em reflexão computacional [126], polimorfismo [368], metadados (*metadata*) [127] e metaconteúdo (*metacontent*) [317], teste baseado em estados [32] e autoteste (*built-in testing*) [126, 424].

Reflexão computacional permite acesso à estrutura interna de um programa e inspeção do seu comportamento, sendo utilizada na área de teste para automatizar a execução dos testes, criando instâncias de classes e invocando métodos em diferentes sequências. No teste de componentes, a reflexão computacional é usada para “carregar” uma determinada classe em uma ferramenta de teste que identifica os métodos e os respectivos parâmetros, viabilizando sua execução. Com isso, a reflexão viabiliza o desenvolvimento de casos de teste, inspecionando o que é necessário para invocar os métodos de cada uma das classes [127]. Rosa e Martins [346] propõem o uso de uma arquitetura reflexiva para validar o comportamento de POO.

Outra solução, similar a um empacotador (*wrapper*), é proposta por Soundarajan e Tyler [368] usando polimorfismo. Dada a especificação formal de um componente, contendo as pré e pós-condições que devem ser satisfeitas na invocação de cada método, métodos polimórficos são criados de modo que, antes da invocação do método real (o que é implementado pelo componente), a versão polimórfica do método verifica se a precondição é satisfeita, coleta informações usadas na invocação do método (valor dos parâmetros, por exemplo), e constata se as pós-condições foram satisfeitas. A desvantagem dessa abordagem é que ela requer uma especificação formal do componente e não garante a cobertura de código. Além disso, para viabilizar a sua utilização, é necessário o desenvolvimento de um gerador automático de empacotadores, uma vez que alterações na especificação do componente (as pré ou pós-condições, por exemplo) podem requerer que os métodos polimórficos sejam gerados/avaliados novamente.

Metadados são, também, utilizados pelos modelos de desenvolvimento de componentes para fornecer informações genéricas sobre o componente, tais como o nome de suas classes, o nome de seus métodos, bem como informações para o teste [127, 317]. Pelo uso de metadados, aspectos estáticos e dinâmicos do componente podem ser consultados pelo cliente para a realização de diversas tarefas. O problema com essa abordagem é que ainda não há um consenso sobre o conjunto de informações que deve ser disponibilizado nem sobre a forma como disponibilizá-lo. Além do mais, essa abordagem requer um trabalho adicional por parte do desenvolvedor do componente.

Outras estratégias propõem uma abordagem integrada para a geração de dados de teste para o teste de componentes de software. Elas combinam informações funcionais e estruturais obtidas a partir de uma especificação formal ou semiformal do componente. A idéia é construir um Grafo Def-Uso [337] do componente com base na sua especificação, a fim de que critérios de teste de fluxo de dados e de controle possam ser utilizados para a geração de dados de teste. O problema é que como as informações estruturais são derivadas da especificação, satisfazer tais critérios não garante a cobertura do código do componente, mas sim de sua especificação. Além disso, uma especificação formal ou semiformal do componente tem de estar disponível [32, 125].

O conceito de componentes autotestáveis também vem sendo explorado. A idéia é disponibilizar componentes de software com capacidades de teste embutidas e que possam ser habilitadas ou desabilitadas, dependendo se o componente encontra-se em operação normal ou em manutenção, por exemplo [127, 269, 424]. Edwards [127] discute como diferentes tipos de informações podem ser embutidas em componentes de software por meio de um empacotador de metadados reflexivo (*reflexive metadata wrapper*). Ele sugere que informações como a especificação do componente, sua documentação, histórico de verificação, serviços de verificação de violação de pré e pós-condições, serviços de autoteste, entre outras, podem ser embutidas utilizando-se tal mecanismo. O problema com essa abordagem, embora muito útil, é que ela exige tempo e recursos adicionais por parte do desenvolvedor do componente, pois é dele que depende a coleta e a inclusão de tais informações. Além disso, quais informações realmente deveriam ser fornecidas e como deveriam ser fornecidas ainda não dispõem de um padrão, tornando mais difícil o trabalho de se desenvolverem ferramentas de teste que necessitam de tais informações.

Como pode ser observado pelos trabalhos descritos, todas as abordagens propostas tentam minimizar o problema do teste de componentes do lado dos clientes, quando o código-fonte do componente não se encontra disponível. Isso implica que, exceto se o componente for dotado de autoteste ou metadados que permitam a coleta de informações de seu estado interno, nenhuma informação sobre cobertura de características estruturais do componente pode ser obtida por parte do cliente.

No caso de componentes Java, tais como JavaBeans, esse problema pode ser superado realizando os testes diretamente no programa objeto (*bytecode*) sem depender do código-fonte Java. O arquivo *bytecode* é uma representação binária independente de plataforma que contém informações de alto nível sobre uma classe, tais como seu nome, o nome de sua superclasse, informações sobre métodos, variáveis e constantes utilizadas, além das instruções de cada um de seus métodos.

Instruções de *bytecode* lembram instruções em linguagem *assembly*, mas armazenam informações de alto nível sobre um programa, de modo que é possível extrair informações de fluxo de controle e de dados a partir delas [414]. Trabalhando diretamente com *bytecode* Java, tanto o desenvolvedor do componente quanto seus clientes podem utilizar a mesma representação e os mesmos critérios para testar componentes Java. Além disso, o cliente será capaz de avaliar qual porcentagem do componente foi coberta por sua massa de teste, ou seja, ele será capaz de avaliar a qualidade de seu conjunto de teste funcional em relação a critérios de teste estruturais.

Chambers et al. [65] e Zhao [457] descrevem duas abordagens diferentes para realizar análise de dependência em *bytecode* Java. Os critérios de fluxo de dados apresentados anteriormente utilizam uma técnica de análise similar à desenvolvida por Zhao [457]. No trabalho

de Zhao [457], a análise de dependência de fluxo de controle e de dados é também realizada diretamente no *bytecode*. A principal diferença entre abordagem utilizada por Zhao [457] e a utilizada por Vincenzi et al. [420] está no modelo de fluxo de dados utilizado para identificar os conjuntos de variáveis que são definidas e usadas. O modelo de fluxo de dados proposto por Zhao [457] considera somente o conjunto de variáveis locais (aqueles definidas dentro de um método) na identificação das dependências de fluxo de dados, e o modelo proposto por Vincenzi et al. [420] considera não somente as variáveis locais, mas também os atributos de instância e de classe, e variáveis agregadas. Assim sendo, acredita-se que o modelo de fluxo de dados proposto por Vincenzi et al. [420] represente mais precisamente as interações de fluxo de dados intramétodo que possam ocorrer em um programa Java.

### 6.6.2 Problemas no teste de integração

O papel do teste é verificar se o componente está sendo utilizado de maneira coerente e que sua integração respeita a especificação fornecida pelo produtor. Algumas questões, porém, dificultam a integração do componente. A mais importante delas é que o código-fonte dos componentes não está disponível, o que impossibilita a utilização de técnicas como a cobertura de elementos estruturais.

Beydeda e Gruhn [33] identificam diversos problemas relacionados à utilização de componentes por parte dos clientes, como: desenvolvimento do componente dependente de um contexto, documentação insuficiente e dependência do cliente em relação ao produtor. Grande parte desses problemas está relacionada com a falta de informação, principalmente do cliente, em relação à maneira como o componente foi desenvolvido. Para lidar com tais problemas e facilitar a execução de algumas tarefas por parte do cliente, em particular a condução de testes de maneira sistemática, algumas abordagens foram propostas.

Beydeda e Gruhn dividem essas abordagens em duas classes: as que procuram lidar com essa falta de informação e as que procuram facilitar a troca de informação entre produtor e cliente. Nessa segunda classe destacam-se alguns trabalhos, como os de Orso et al. [318], Liu e Richardson [246] e Edwards [127], que procuram estabelecer formas de encapsular no componente informações que possam ser úteis ao cliente na condução de algumas tarefas, incluindo o teste de integração do componente. Um problema com esses trabalhos é que eles identificam como a informação deve ser fornecida ao cliente, mas não quais são essas informações.

## 6.7 Ferramentas

Na prática, a aplicação de um critério de teste está fortemente condicionada à sua automatização. O desenvolvimento de ferramentas de teste é de fundamental importância, uma vez que a atividade de teste é muito propensa a erros, além de improdutiva, se aplicada manualmente. Além disso, ferramentas de teste facilitam a condução de estudos experimentais que buscam avaliar e comparar os diversos critérios de teste. Assim sendo, a disponibilidade de ferramentas de teste propicia maior qualidade e produtividade para a atividade de teste. A seguir é apresentada a descrição, parcialmente extraída do trabalho de Domingues [120], de uma série de ferramentas comerciais e não comerciais que se encontram disponíveis para o teste de POO e algumas também destinadas ao teste de componentes, principalmente para

o teste de programas escritos em C++ e Java. Essas ferramentas foram obtidas por meio de pesquisa na *World Wide Web* e a descrição apresentada é baseada na documentação existente de cada uma, bem como na execução de tais ferramentas, utilizando versões de demonstração quando essas encontravam-se disponíveis.

A ferramenta de teste PiSCES (*Coverage Tracker for Java*) é uma das primeiras ferramentas de teste desenvolvidas para o teste de *applets* Java. PiSCES é uma ferramenta de medida de cobertura (comandos e decisões) que identifica quais partes do código-fonte já foram exercitadas durante os testes e quais ainda precisam ser. Embora tenha sido projetada para o teste de *applets* Java, a ferramenta requer o código-fonte para derivar os requisitos de teste a serem cobertos [36].

Outra iniciativa no desenvolvimento de um conjunto de ferramentas de teste foi iniciado pela Sun Microsystems em meados dos anos 90 [367]. *SunTest (Java Testing Tools from Sun)* incluía uma ferramenta de captura-reprodução (*capture-replay*), denominada *JavaStar*, destinada ao teste de programas com interface gráfica; uma ferramenta de medida de cobertura de código (comandos e decisões), denominada *JavaScope*; uma ferramenta para a geração de drivers para o teste de métodos, denominada *JavaSpec*; uma ferramenta para o teste de carga, denominada *JavaLoad*; uma ferramenta destinada à análise estática de *threads*, denominada *JavaLoom*; e uma ferramenta para verificar a portabilidade de programas Java, denominada *JavaPureCheck*. Infelizmente, em 1999 a *Sun* tomou uma decisão estratégica de descontinuar o desenvolvimento desse conjunto de ferramentas de teste.

*JProbe Suite* [333] é um conjunto de três ferramentas, composto por: *JProbe Profiler and Memory Debugger* que ajuda a eliminar gargalos de execução causados por algoritmos inefficientes em códigos Java e aponta as causas de perdas de memória nessas aplicações, rastreando quais objetos seguram referências para outros; *JProbe Threadalyzer*, que monitora interações entre *threads* e avisa o testador quando essa interação representar perigo, bem como identifica potenciais perigos de concorrências e *deadlocks*; e *JProbe Coverage*, que localiza códigos não testados e mede quanto do código está sendo exercitado, permitindo ao testador estimar a confiança dos testes executados.

TCAT/Java [342] e JCover [267] são duas ferramentas de teste que implementam os critérios de cobertura de comandos e decisões para o teste de programas e *applets* Java. Tais ferramentas também exigem o código-fonte para conduzir a atividade de teste.

*Parasoft C++ Test* [323] é uma ferramenta de teste de unidade para códigos C/C++ que executa os seguintes tipos de teste: 1) teste funcional; 2) teste estrutural (cobertura de comandos e decisões); e 3) teste de regressão. Essa ferramenta permite aos desenvolvedores testarem suas classes imediatamente após elas terem sido escritas e compiladas por meio da automatização da criação de *driver* e de quaisquer *stubs* necessários, no qual o testador pode personalizar os respectivos valores de retorno ou, ainda, entrar com os próprios *stubs*. Essa ferramenta automatiza o teste funcional com a geração automática dos casos de teste e documentação dos resultados esperados, os quais são comparados com os resultados reais. Além disso, o testador pode incluir seus casos de teste e obter relatórios personalizados. No teste estrutural, essa ferramenta gera e executa automaticamente casos de teste projetados para testar uma classe especificada. Qualquer problema encontrado é assinalado e apresentado em uma estrutura gráfica. Esses casos de teste são automaticamente salvos, de forma que possam ser usados facilmente no teste de regressão para se ter certeza de que modificações nas aplicações não introduziram novos defeitos.

*Parasoft Insure++* [324] é uma ferramenta de detecção de defeitos em tempo de execução para códigos C/C++ que acelera as tarefas de depuração. Pela visualização e execução de um programa e manipulação simultânea de dados, essa ferramenta ajuda desenvolvedores a entenderem como o código funciona. Essa ferramenta provê um completo diagnóstico de cada problema, incluindo a descrição do erro, a linha do código-fonte que contém o erro e informações da pilha de execução. Os relatórios de defeitos providos pela ferramenta ajudam desenvolvedores a encontrarem defeitos mais facilmente do que no teste manual. Apesar de não modificar o código-fonte, essa ferramenta provê informações completas para a correção dos defeitos.

*ProLint Advanced Graphical Lint* [332] verifica os módulos de uma aplicação com o objetivo de encontrar defeitos e inconsistências para mais de 600 tipos de problemas, proporcionando uma maior confiabilidade e portabilidade de códigos C/C++.

*Rational PureCoverage* [339] é uma ferramenta de análise de cobertura (comandos e decisões) para códigos C++ e Java que aponta as áreas do código que foram ou não exercitadas durante os testes. Ela expõe o código não testado em todas as partes da aplicação. Essa ferramenta possui diversas opções de apresentação que auxiliam o testador na compreensão de quais partes do código foram ou não testadas utilizando informação de cobertura de código em arquivos, módulos e linhas de código, permitindo ainda ao testador escolher o nível de cobertura de código por módulos e, então, focalizar as partes da aplicação que mais lhe interessam.

*Rational Purify* [340] é uma ferramenta de detecção de defeitos em tempo de execução para códigos C/C++ que ajuda o testador a encontrar a raiz do problema em qualquer parte da aplicação. Como a *Rational PureCoverage*, essa ferramenta permite ao testador escolher o nível de verificação por módulos.

JUnit é um *framework* de teste que vem sendo muito utilizado e viabiliza a documentação e a execução automática de casos de teste. O framework JUnit é de código aberto e pode ser utilizado para escrever e executar de forma automática um conjunto de teste, fornecendo relatórios sobre quais casos de teste não se comportaram de acordo com o que foi especificado. A idéia básica é implementar algumas classes específicas que armazenam informações sobre os dados de entrada e a respectiva saída esperada para cada caso de teste. Após a execução de um caso de teste, a saída obtida é comparada com a saída esperada e quaisquer discrepâncias são reportadas. O principal problema do JUnit é que ele não fornece informação a respeito da cobertura obtida pelos casos de teste e também não apóia a aplicação de um critério de teste. Entretanto, JUnit pode ser utilizado mesmo que somente o *bytecode* e a especificação do programa estejam disponíveis [28].

Com o intuito de oferecer ao testador uma abordagem para verificar a cobertura dos casos de testes JUnit em relação critérios estruturais de fluxo de controle, foi desenvolvida a ferramenta Cobertura [119, 165]. Essa ferramenta apóia a aplicação dos critérios Todos-Nós e Todas-Decisões no teste de programas Java e tem sido bastante utilizada por programadores que trabalham nessa linguagem devido a sua simplicidade, facilidade de uso e integração com a ferramenta JUnit.

Outra ferramenta que apóia a aplicação de critérios de teste estruturais é a ferramenta JaBUTi [422]. O diferencial da JaBUTi em relação a outras ferramentas de teste é a não-exigência da disponibilidade do código-fonte para a realização dos testes. A ferramenta trabalha diretamente com *bytecode* Java e apóia a aplicação dos critérios Todos-Nós, Todas-

Arestas, Todos-Usos e Todos-Potenciais-Usos no teste intramétodos. Teoricamente, qualquer linguagem de programação que produza como resultado um código objeto compatível com a especificação da máquina virtual Java poderia ser testado pela ferramenta JaBUTi. Observa-se que, por requerer apenas o *bytecode* para a realização dos testes, essa ferramenta tem sido estendida para o teste de programas orientados a aspectos [237] (vide Capítulo 7), teste de aplicações com conexão com banco de dados [297], teste de programas implementado(s) utilizando agentes móveis [112] e teste de aplicações J2ME [113].

Uma outra ferramenta, denominada *Component Test Bench* (CTB), pode ser utilizada no teste de componentes de software [54]. A ferramenta fornece um padrão genérico que permite ao desenvolvedor do componente especificar o conjunto de teste utilizado no teste de um dado componente. O conjunto de teste é armazenado em um arquivo XML (*eXtensible Markup Language*) e executado por meio de uma ferramenta que compõe o CTB, denominada IRTB (*Instrumented runtime system*), ou eles podem ser executados invocando-se a execução de uma máquina virtual Java ou, ainda, compilando e executando programas em C ou C++. Caso o conjunto de teste seja disponibilizado juntamente com o componente, o cliente poderá repetir os testes e verificar se o componente comporta-se como esperado no contexto do sistema baseado em componentes. A ferramenta não fornece medidas de cobertura de código e é utilizada somente para a realização dos teste de conformidade, ou seja, avaliar se o componente comporta-se como esperado.

Uma abordagem similar é utilizada pela *Parasoft JTest* [95], uma ferramenta de teste de classes para códigos Java que executa os seguintes tipos de teste: 1) análise estática; 2) teste funcional; 3) teste estrutural (cobertura de comandos e decisões); e 4) teste de regressão. Essa ferramenta pode executar todos esses tipos de teste em uma simples classe ou em um conjunto de classes. No teste funcional, a ferramenta gera automaticamente um conjunto essencial de casos de teste, estrategicamente projetado para alcançar uma cobertura tão completa quanto possível. O testador pode aumentar esse conjunto de casos de teste com os próprios casos de teste. Em seguida, a ferramenta executa automaticamente todos esses casos de teste e mostra seus resultados em uma representação de árvore. O testador pode ainda visualizar esses resultados e validá-los e, quando a ferramenta executar testes subsequentes nessa classe, a ferramenta será capaz de notificar ao testador quando ocorrerem erros nos testes de regressão e funcional. A principal diferença entre JTest e CTB é que JTest utiliza o framework JUnit para armazenar e executar os casos de teste automaticamente, enquanto CTB armazena os casos de teste em arquivos XML. JTest não detecta somente erros, como também pode prevenir erros e assegurar que defeitos não serão adicionados no código quando ele é modificado de maneira automatizada, reexecutando os casos de teste documentados com o JUnit. A análise estática previne enganos padronizando códigos e, por consequência, reduz a possibilidade de defeitos serem inseridos no código.

*Glass JAR Toolkit* (GJTK) é uma ferramenta de teste de cobertura que opera diretamente no *bytecode* Java e não requer o código-fonte para aplicar critérios de teste de fluxo de controle (cobertura de comandos e decisões) em *bytecode* Java. Ela pode ser utilizada para arquivos .class diretamente ou arquivos .class empacotados em arquivos .jar [124].

Para apoiar o teste de mutação em POO, algumas ferramentas também vêm sendo desenvolvidas. Alexander et al. [12] propuseram a arquitetura de uma ferramenta, denominada *Object Mutation Engine – OME*, que implementa um subconjunto dos operadores de mutação propostos por Bieman [34] no teste de classes Java.

Tanto Chevalley [75] quanto Ma et al. [257] desenvolveram também ferramentas de teste baseadas em mutação para apoiar a aplicação dos conjuntos de operadores de mutação que ambos desenvolveram/estenderam. A característica comum entre essas ferramentas é que elas utilizam o conceito de reflexão computacional para implementar as mutações. Ambas as ferramentas utilizam OpenJava [387], que é um sistema reflexivo em tempo de compilação, o qual utiliza macros (metaprogramas) para manipular metaobjetos que representam entidades lógicas de um programa, facilitando a realização das mutações.

Recentemente, Bybro [56] desenvolveu uma ferramenta de teste, denominada *Mutation Testing System*, que implementa parte do conjunto de operadores de mutação proposto por Ma et al. [257]. 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.

A Tabela 6.19 apresenta comparações das atividades apoiadas por essas ferramentas de teste para programas orientados a objetos, tais como: 1) fases do teste; 2) critérios de teste; 3) linguagem suportada; 4) exigência do código-fonte; 5) atividade de depuração; e 6) teste de regressão.

Quando se analisa a Tabela 6.19 observa-se que o framework JUnit [28] é uma das ferramentas que pode ser utilizada tanto para o programa quanto para o teste de componentes de software desenvolvidos em Java, mas tal ferramenta apóia apenas a execução automática de casos de teste e realização de testes funcionais, sem apoio de nenhum critério de teste específico e não fornecendo informação sobre a cobertura de código obtida por determinado conjunto de teste. Outra ferramenta que também apóia somente o teste funcional é a CTB [54].

Considerando as ferramentas que permitem a avaliação de cobertura de código por meio de critérios estruturais, observa-se que das ferramentas analisadas todas apóiam somente o teste de fluxo de controle (cobertura de comandos e decisão) em POO. Nenhuma delas apóia a aplicação de algum critério de fluxo de dados seja para o teste de unidade, integração ou sistema. Além disso, exceto pela ferramenta GlassJAR [124] e pelas que apóiam o teste funcional, todas as demais necessitam do código-fonte para a aplicação dos critérios, dificultando sua utilização no teste estrutural de componentes de software por parte dos clientes, os quais, em geral, não têm acesso ao código-fonte.

Ao se levarem em conta as ferramentas que apóiam o teste de mutação, observa-se que todas fornecem suporte ao teste de integração interclasse. Mesmo que o teste de integração seja de grande importância no contexto de POO, tendo em vista o grande número de chamadas de métodos intra e interclasses existentes, considera-se que, ainda assim, os testes de unidade devam ser realizados, principalmente, para eliminar os defeitos de lógica e de programação antes de as unidades individuais serem integradas.

## 6.8 Considerações finais

Este capítulo apresentou o estado atual da atividade de teste no contexto de teste de programas orientados a objetos e daqueles desenvolvidos utilizando componentes de software. No que se refere ao teste de programas orientados a objetos, observou-se que, embora as características de orientação a objetos possam trazer vantagens para o desenvolvedor em termos de proximidade com o mundo real e reutilização de código, elas também implicam trazer

Tabela 6.19 – Características apresentadas pelas ferramentas de teste OO

| Ferramentas de teste de software     | Teste de unidade | Teste de integração | Teste de sistema | Critérios funcionais | Critérios de fluxo de controle | Critérios de fluxo de dados | Critérios baseados em mutação | Linguagem suportada | Exigência de código-fonte | Atividade de depuração | Teste de regressão |
|--------------------------------------|------------------|---------------------|------------------|----------------------|--------------------------------|-----------------------------|-------------------------------|---------------------|---------------------------|------------------------|--------------------|
| <i>PiSCES</i>                        | ✓                |                     |                  | ✓                    |                                |                             |                               | Java                | ✓                         |                        |                    |
| <i>SunTest</i>                       | ✓                | ✓                   |                  | ✓                    |                                |                             |                               | Java                | ✓                         |                        |                    |
| <i>xSuds Toolsuite</i>               | ✓                | ✓                   |                  | ✓                    | ✓                              |                             |                               | C/C++               | ✓                         | ✓                      | ✓                  |
| <i>JProbe Developer Suite</i>        | ✓                | ✓                   |                  | ✓                    |                                |                             |                               | Java                | ✓                         | ✓                      |                    |
| <i>TCAT/Java</i>                     | ✓                |                     |                  | ✓                    |                                |                             |                               | Java                | ✓                         |                        |                    |
| <i>JCover</i>                        | ✓                |                     |                  | ✓                    |                                |                             |                               | Java                | ✓                         |                        |                    |
| <i>Parasoft C++ Test</i>             | ✓                | ✓                   | ✓                | ✓                    |                                |                             |                               | C/C++               | ✓                         |                        | ✓                  |
| <i>Parasoft Insure++</i>             |                  |                     |                  |                      |                                |                             |                               | C/C++               | ✓                         | ✓                      |                    |
| <i>ProLint</i>                       |                  |                     |                  |                      |                                |                             |                               | C/C++               | ✓                         |                        |                    |
| <i>Rational PureCoverage</i>         | ✓                | ✓                   |                  | ✓                    |                                |                             |                               | C++/Java            | ✓                         |                        |                    |
| <i>Rational Purify</i>               |                  |                     | ✓                |                      |                                |                             |                               | C/C++               | ✓                         | ✓                      |                    |
| <i>JUnit</i>                         | ✓                |                     |                  | ✓                    |                                |                             |                               | Java                |                           |                        |                    |
| <i>Cobertura</i>                     | ✓                |                     |                  |                      | ✓                              |                             |                               | Java                |                           |                        |                    |
| <i>JaBUTi</i>                        | ✓                | ✓                   |                  | ✓                    | ✓                              | ✓                           |                               | Java                |                           | ✓                      |                    |
| <i>CTB</i>                           | ✓                | ✓                   | ✓                | ✓                    |                                |                             |                               | Java/C/C++          |                           |                        |                    |
| <i>Parasoft JTest</i>                | ✓                | ✓                   | ✓                | ✓                    | ✓                              |                             |                               | Java                | ✓                         |                        | ✓                  |
| <i>Glass JAR Toolkit</i>             | ✓                | ✓                   | ✓                | ✓                    | ✓                              |                             |                               | Java                |                           |                        |                    |
| <i>Object Mutation Engine</i>        |                  | ✓                   |                  |                      |                                |                             | ✓                             | Java                | ✓                         |                        |                    |
| <i>Ferramenta de Chevalley [75]</i>  |                  | ✓                   |                  |                      |                                |                             | ✓                             | Java                | ✓                         |                        |                    |
| <i>Ferramenta de Ma et al. [257]</i> | ✓                |                     |                  |                      |                                |                             | ✓                             | Java                | ✓                         |                        |                    |
| <i>Mutation Testing System</i>       |                  | ✓                   |                  |                      |                                |                             | ✓                             | Java                | ✓                         |                        |                    |

algumas dificuldades adicionais para a atividade de teste. Mesmo com essas dificuldades, critérios de teste estão sendo desenvolvidos ou adaptados a partir daqueles já existentes para o teste de programas procedimentais e vêm sendo utilizados no teste de programas orientados a objetos.

Apesar da popularização trazida pelas vantagens do desenvolvimento baseado em componentes, nota-se que alguns problemas relacionados a esse paradigma continuam sem uma solução geral. É o caso da atividade de teste, que introduz alguns problemas importantes no desenvolvimento baseado em componentes, principalmente considerando a perspectiva do cliente, que faz uso do componente desenvolvido e, em geral, não possui acesso a todas as informações que dispõe o desenvolvedor do mesmo. Diversos trabalhos na literatura têm identificado um ponto em comum, como fonte de tais problemas: a falta de dados a respeito do desenvolvimento do componente, de modo a facilitar sua utilização como parte de um sistema baseado em componentes. Esses mesmos trabalhos propõem algumas formas de prover tal informação ao cliente do componente, tendo por fim minimizar as dificuldades em algumas tarefas do processo de desenvolvimento, em particular, na condução da atividade de teste; entretanto, ainda não há consenso sobre o conjunto de informações que deve ser disponibilizado.

# Capítulo 7

## Teste de Aspectos

*Otávio Augusto Lazzarini Lemos (ICMC/USP)*

*Reginaldo Ré (UTFPR)*

*Paulo César Masiero (ICMC/USP)*

### 7.1 Introdução

A programação orientada a objetos (POO) revolucionou as técnicas de programação, auxiliando no tratamento de complexidades inerentes ao software [58] e possibilitando uma melhor separação de interesses na implementação e no projeto de sistemas. Apesar disso, alguns problemas não foram completamente resolvidos por suas construções. Especificamente, a orientação a objetos não permite a clara separação de certos tipos de requisitos – geralmente não funcionais – que têm sido chamados de interesses transversais (*crosscutting concerns*), pois sua implementação tende a se espalhar por diversos módulos do sistema, em vez de ficar localizada em unidades isoladas.

Como uma proposta para resolver esse problema, no final da década de 1990 [209] surgiu a programação orientada a aspectos (POA), que oferece mecanismos para a construção de programas em que os interesses transversais ficam separados dos interesses básicos, em vez de espalhados pelo sistema. A POA não é um novo paradigma de programação, mas sim uma técnica para ser utilizada em conjunto com os diversos paradigmas de programação existentes [128], possibilitando a construção de sistemas com maior qualidade.

A diferença principal apresentada pela POA com relação às outras técnicas de programação é possibilitar a implementação de módulos isolados – os aspectos –, que têm a capacidade de afetar vários outros módulos do sistema de forma transversal. Em POA, um único aspecto pode contribuir para a implementação de diversos outros módulos que implementam as funcionalidades básicas (chamado de código-base). Esse mecanismo que permite a um aspecto afetar vários pontos do sistema é chamado de quantificação.

A POA é uma abordagem recente. Apesar disso, já existem iniciativas do seu uso para o apoio ao teste de software, utilizando principalmente a linguagem AspectJ (que será apresentada na próxima seção). A separação entre código-base e código de teste auxilia o manuseio de ambos, e a facilidade de inserir e remover aspectos na aplicação pode levar à criação de ce-

nários de teste com maior rapidez. Além disso, os aspectos podem ser facilmente removidos quando a atividade de teste é finalizada, conservando intacto o programa original.

Por outro lado, a POA apresenta novos desafios em cada fase do ciclo de vida de desenvolvimento de software porque apresenta peculiaridades com relação a outras técnicas de programação. Por exemplo, por causa da existência do aspecto como uma entidade nova, técnicas de modelagem propostas para projetar programas orientados a objetos não podem ser utilizadas diretamente e, portanto, necessitam de adaptações – ou novas abordagens – para serem adequadamente utilizadas.

Nesse sentido, também as técnicas de teste de software devem ser revisitadas para serem aplicadas nesse novo contexto. Por exemplo, para aplicar o teste estrutural a programas orientados a aspectos é necessário adaptar as representações estruturais dos programas – os grafos de fluxo de controle e de dados – para que os critérios possam ser adequadamente utilizados. Além disso, pode ser necessário definir novos tipos de critérios, independentemente da técnica utilizada, que irão auxiliar a descobrir defeitos em programas orientados a aspectos.

Neste capítulo são abordadas duas perspectivas relacionadas a POA e ao teste de software: 1) como a POA pode auxiliar o teste de software OO e 2) como é possível testar programas que utilizam a POA. Na primeira perspectiva são apresentadas alternativas que utilizam a POA em atividades relacionadas ao teste de software. Na segunda perspectiva são apresentadas abordagens de teste aplicadas a programas orientados a aspectos.

### 7.1.1 Definições

Aspectos podem adicionar comportamento em outros módulos por meio do mecanismo de quantificação, e, para isso, é necessário que se possam identificar pontos específicos da execução do programa, chamados de pontos de junção (*join points*), para o comportamento adicional ser definido. Dessa forma, uma linguagem orientada a aspectos deve fornecer um modelo por meio do qual possam ser identificados os pontos na execução do programa (chamado modelo de pontos de junção). Por exemplo, em AspectJ – uma extensão de Java que apóia a POA –, o modelo de pontos de junção é baseado na semântica de execução do programa e os pontos são identificados por padrões do tipo: chamada ou execução de um método com um certo nome ou padrão de nome, leitura/escrita de atributos com um certo nome ou padrão de nome, etc.

Um conjunto de pontos de junção – ou somente conjunto de junção (*pointcut*) – identifica diversos pontos de junção em um sistema. O conjunto de junção é utilizado pelo comportamento transversal, que é implementado por uma construção similar a um método chamada de adendo (*advice*), que executa antes, depois ou no lugar dos pontos de junção identificados. Esses adendos são chamados de adendos anteriores, posteriores e de contorno. Por exemplo, para implementar um interesse de registro que imprime uma mensagem toda vez que um método é chamado, poderia ser utilizado um aspecto com um adendo anterior. Esse adendo executaria toda vez que qualquer método do sistema fosse chamado.

Depois do código-base e de os aspectos serem codificados, é necessário combinar (*weave*) os módulos em uma aplicação executável. Assim, toda linguagem de programação orientada a aspectos deve contar também com um combinador (*weaver*) responsável por essa tarefa. A combinação pode ser feita em diversos momentos, dependendo da decisão de implementação tomada para cada linguagem de programação específica: combinação dinâmica é o processo

de combinação que ocorre em tempo de execução, enquanto a combinação estática ocorre em tempo de compilação.

De maneira geral, os aspectos podem servir para dois propósitos: auxiliar atividades de desenvolvimento (aspectos de *desenvolvimento*) ou implementar interesses que estarão na aplicação final (aspectos de *produção*). No primeiro caso os aspectos são utilizados para ajudar em atividades como depuração, garantia de que padrões de código estejam sendo utilizados e teste de software. Nesses casos, os aspectos não estarão presentes na aplicação final, mas serão apenas utilizados durante o desenvolvimento, sendo retirados antes da entrega do produto de software. No segundo caso os aspectos cumprem papéis relacionados à funcionalidade do software e estarão presentes na aplicação final como, por exemplo, aspectos que implementam regras de negócio.

## 7.1.2 AspectJ

A linguagem de programação orientada a aspectos mais estável e utilizada no presente momento é a linguagem AspectJ [391], uma extensão de Java para apoiar a POA, utilizada como base neste capítulo. Por ser uma extensão de Java, AspectJ contém todas as construções dessa linguagem e, portanto, é construída sobre o paradigma OO.

Na Figura 7.1 é mostrada parte do diagrama de classes/aspectos de uma aplicação orientada a aspectos simples implementada em AspectJ, que será utilizada para explicar os conceitos da linguagem. O exemplo refere-se a um sistema que modela conexões telefônicas para as quais os interesses de temporização, faturamento e transferência de chamadas são implementados utilizando aspectos. No sistema de telefonia, clientes fazem, aceitam, juntam e concluem ligações de longa distância ou locais. As classes-base oferecem funcionalidade para simular clientes, chamadas e conexões.

Algumas classes do sistema são:

- `Cliente`, que modela clientes e possui atributos para nome, telefone, código de área do cliente e senha para acessar o sistema da empresa de telefonia (em um sítio na *Web* em que ele pode alterar seus dados cadastrais, por exemplo);
- a classe abstrata `Conexao` com suas duas classes concretas `Local` e `LongaDistancia`, que modelam conexões locais e de longa distância;
- `Chamada`, que modela chamadas;
- `Temporizador`, que modela temporizadores;
- `FabricaConexao`, que é responsável por criar as conexões.

Os aspectos do sistema são:

- `Temporizacao`, que implementa o interesse de temporização e se encarrega de registrar os tempos das conexões por cliente, iniciando e parando um temporizador associado a cada conexão;

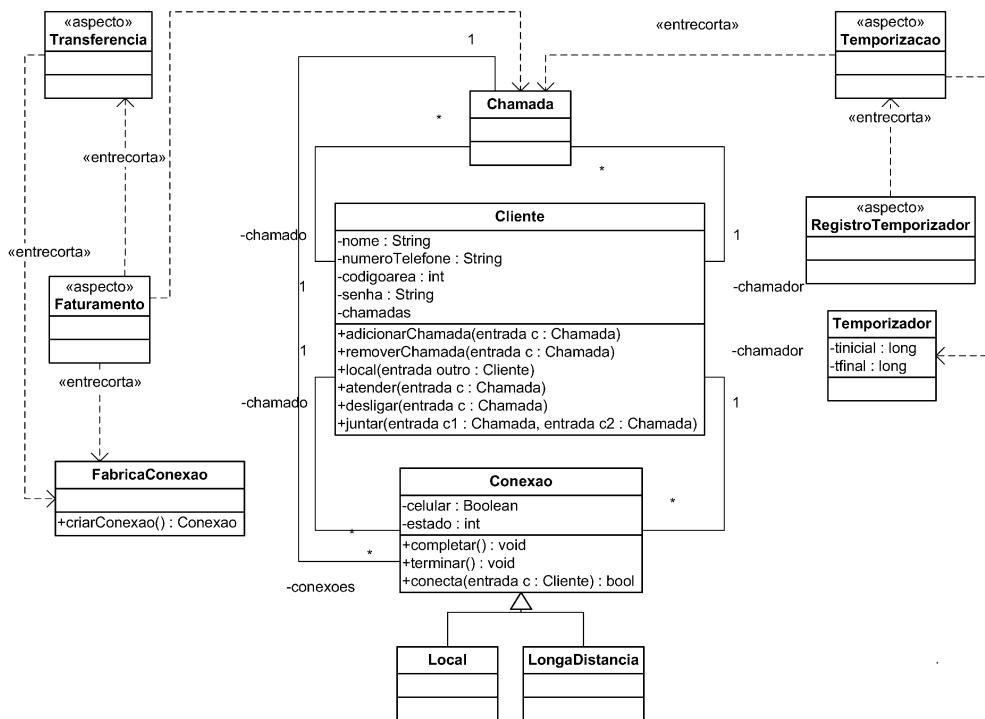


Figura 7.1 – Parte do diagrama de classes/aspectos de uma aplicação de telefonia.

- **Faturamento**, que implementa o interesse de faturamento e se encarrega de declarar o seguinte: cada conexão tem um cliente pagador e, também, as chamadas locais e de longa distância devem ser cobradas com taxas diferentes;
- **RegistroTemporizador**, que implementa um registro, o qual imprime na tela os horários em que um temporizador inicia e pára;
- **Transferencia**, que implementa o interesse de transferência de chamadas ocorrido quando o cliente informa que outro cliente deve receber suas chamadas (por exemplo quando ele viaja).

No Programa 7.1 são mostrados os códigos da classe **Chamada** e do aspecto **Temporizacao**. Essa aplicação também será utilizada ao longo do capítulo e, quando necessárias, outras partes do código e do diagrama de classes serão apresentadas.

---

Programa 7.1

```

1  public class Chamada {
2      private Cliente chamador, chamado;
3      private Vector conexoes = new Vector();
4
5      public Chamada(Cliente chamador, Cliente chamado, boolean cel) {
6          this.chamador = chamador;
7          this.chamado = chamado;

```

---

```
8     Conexao c;
9     c =
10    FabricaConexao.criarConexao(chamador, chamado, cel);
11    conexoes.addElement(c);
12 }
13
14 public void atende() {
15     Conexao conexao = (Conexao)conexoes.lastElement();
16     conexao.completar();
17 }
18 public boolean conectado(){
19     return ((Conexao)conexoes.lastElement()).getEstado()
20         == Conexao.INICIADA;
21 }
22
23 public void desligar(Cliente c) {
24     for(Enumeration e = conexoes.elements(); e.hasMoreElements();) {
25         ((Conexao)e.nextElement()).desconectar();
26     }
27 }
28
29 public boolean inclui(Cliente c){
30     boolean resultado = false;
31     for(Enumeration e = conexoes.elements(); e.hasMoreElements();) {
32         resultado = resultado || ((Conexao)e.nextElement()).conecta(c);
33     }
34     return resultado;
35 }
36
37 public void juntar(Chamada outra){
38     for(Enumeration e=outra.conexoes.elements(); e.hasMoreElements();) {
39         Conexao con = (Conexao)e.nextElement();
40         outra.conexoes.removeElement(con);
41         conexoes.addElement(con);
42     }
43 }
44 }
45
46 public aspect Temporizacao {
47
48     public long Cliente.tempototalConexao = 0;
49
50     public long getTempoTotalConexao(Cliente cli) {
51         return cli.tempototalConexao;
52     }
53
54     private Temporizador Conexao.temporizador = new Temporizador();
55     public Temporizador getTemporizador(Conexao con) {
56         return con.temporizador;
57     }
58
59     after (Conexao c) returning () :
60         target(c) && call(void Conexao.inicia()) {
61             getTemporizador(c).iniciar();
62         }
63
64     pointcut finalizacao(Conexao c) : target(c) &&
65         call(void Conexao.desconecta());
66
67     after(Conexao c) returning () : finalizacao(c) {
68         getTemporizador(c).parar();
```

```
69      c.getChamador().tempoTotalConexao+=getTemporizador(c).getTempo();  
70      c.getChamado().tempoTotalConexao+=getTemporizador(c).getTempo();  
71  }  
72 }
```

---

Além das classes Java comuns, em AspectJ podem-se codificar aspectos por meio da palavra reservada `aspect` (como, por exemplo, o aspecto `Temporizacao` – linhas 46-72 do Programa 7.1). Aspectos são módulos que combinam: especificações de pontos de junção utilizando conjuntos de junção (`pointcut`); adendos anteriores (`before`), posteriores (`after`) e de contorno (`around`); e declarações intertipos, utilizadas para introduzir atributos, métodos e heranças em outras classes ou interfaces, para os propósitos do aspecto. Versões recentes de AspectJ também apóiam declarações de avisos e erros de compilação quando certos pontos de junção são identificados [391].

O modelo de pontos de junção de AspectJ é baseado nas construções sintáticas da linguagem: chamada e execução de métodos/construtores (`call` e `execution`), leitura e escrita de atributos (`get` e `set`), execução de determinado tratador de exceção (`handler`), e outros. Um conjunto de junção é definido com base nos designadores de conjuntos de junção, que são os `call`, `execution`, `get`, etc., também chamados de designadores de conjuntos de junção primitivos. Eles podem ser compostos a outros tipos de designadores, utilizando operadores ‘e’ (`&&`) e ‘ou’ (`||`) para formar um único conjunto de junção. Além disso, pode ser utilizado o operador de negação (`!`) para evitar que determinados pontos de junção sejam selecionados. Por questões de simplificação, será usado o termo conjunto de junção para se referir tanto ao próprio conjunto quanto aos designadores.

Os outros tipos de conjuntos de junção 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`), verificação de alguma condição dinâmica ou estática (`if`), tipo do objeto corrente no ponto de junção (`this`), tipo do objeto do qual o método está sendo chamado 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 do qual o método está sendo chamado e argumentos do método do ponto de junção, respectivamente). Um exemplo de conjunto de junção no Programa 7.1 é o `finalizacao` declarado na linha 64, que identifica todas as chamadas ao método `Conexao.desconecta` e, além disso, obtém o objeto do tipo `Conexao` do qual o método está sendo chamado.

Em AspectJ, existem três tipos de adendos posteriores: `after returning`, `after throwing` e simplesmente `after`. O primeiro é executado apenas quando o ponto de junção retorna normalmente; o segundo, apenas quando uma exceção é lançada; e o terceiro executa sempre. O aspecto `Temporizacao` listado no Programa 7.1 possui dois adendos posteriores para iniciar e parar o temporizador de uma conexão, atribuindo a hora atual ao atributo do tipo `Temporizador` introduzido na classe `Conexao` por meio de uma declaração intertipos (linha 54). Os adendos definem comportamento nos seguintes conjuntos de junção: 1) todas as chamadas ao método `Conexao.inicia()` – que completa uma ligação – iniciando o temporizador (linhas 59-62) e 2) todas as chamadas ao método `Conexao.desconecta()` – que termina uma conexão – parando o temporizador (linhas 64-71).



Os adendos de contorno (não presentes no exemplo) podem fazer com que o ponto de junção alcançado prossiga com a sua execução por meio do método `proceed()`, que também pode ser utilizado para obter o valor de retorno do ponto de junção. Dentro dos adendos também é possível capturar dados do contexto dos pontos de junção, como nos dois adendos do aspecto `Temporizacao` listados no Programa 7.1. São duas as maneiras de se obtiverem os dados do contexto dos pontos de junção: utilizando os conjuntos de junção do tipo `args` (para obter os argumentos), `this` (para obter o objeto executando no ponto de junção) ou `target` (para obter o objeto alvo em um ponto de junção, como no exemplo); ou por meio da palavra reservada `thisJoinPoint`, que funciona como um objeto que encapsula o ponto de junção alcançado.

Do ponto de vista da implementação, para ser compatível com qualquer máquina virtual Java, o compilador de AspectJ gera, a partir do código-fonte do código-base e código-fonte de aspectos, *bytecode* Java comum, obtido por meio da combinação de código-base e aspectos. Para que isso seja possível, 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 é o mesmo do adendo, com exceção do tratamento especial que é dado para os adendos de contorno quando o método `proceed()` é explicitamente chamado [183].

A implementação da execução dos adendos é feita da seguinte maneira: primeiramente o compilador identifica os possíveis pontos de junção no programa e, de maneira geral, insere uma chamada ao método correspondente ao adendo antes, depois ou em substituição ao ponto de junção, 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 estes casos são adicionados alguns resíduos no *bytecode* para fazer as verificações dinâmicas necessárias. Um exemplo de resíduo pode ser uma instrução condicional `if` que checa o tipo de determinado parâmetro, inserida antes da chamada ao método correspondente a um adendo, quando a execução do adendo depende do tipo do parâmetro [183].

É necessário tomar cuidado para não confundir detalhes de implementação da linguagem AspectJ com conceitos essenciais de POA. Deve ficar claro para o leitor que os métodos em que os adendos definem comportamento adicional são sintaticamente inconscientes (*oblivious*) da existência dos aspectos, apesar da estratégia utilizada pelo AspectJ de inserir chamadas nos pontos de junção identificados no *bytecode* para representar as execuções implícitas. O conceito de inconsciência (*obliviousness*) [138] está fortemente ligado à POA e é considerado uma propriedade interessante do software, pois contribui para um menor acoplamento dos módulos do sistema.

## 7.2 Teste de programas OO apoiado por aspectos

A separação entre código-base funcional e código transversal permitida pela POA é bastante útil para apoiar eficientemente o teste de código escrito apenas na linguagem base. Por exemplo: com AspectJ pode-se apoiar o teste de programas Java e com AspectC pode-se apoiar o teste de programas escritos em C. O princípio básico é que o código de apoio, como o de rastreamento (*trace*) e o de registro (*log*), pode ser codificado como um ou mais aspectos e, depois de concluídos os testes, pode ser facilmente retirado do código, uma vez que não deve fazer parte do código a ser colocado em produção. Portanto, os aspectos que apóiam o

teste são aspectos de desenvolvimento, conforme definido na Seção 7.1.1. Nesta seção são apresentados alguns exemplos de uso de aspectos para apoiar o teste de programas orientadas a objetos. Deve-se notar, entretanto, que as técnicas apresentadas nesta seção podem também ser aplicadas a programas orientados a aspectos.

### 7.2.1 Imitadores virtuais de objetos

Os imitadores virtuais de objetos [285] foram propostos como uma alternativa aos imitadores de objetos (*mock objects*) usados principalmente em métodos ágeis de desenvolvimento de software. Objetos imitadores são utilizados para simular o comportamento de outras unidades das quais uma unidade em teste depende e, assim, executar isoladamente seu teste, sem interferências. As implementações dos imitadores de objetos são simples e devem retornar valores fixos, como os módulos pseudocontrolados (stubs) usados em POO.

Para ilustrar o uso de objetos imitadores, considere um exemplo simples relacionado à autenticação de usuários. Na Figura 7.2(a) é mostrado um exemplo de teste de uma unidade orientada a objetos para a operação `validar()` [285]. Nesse sistema, a autenticação dos usuários é iniciada quando os atributos `idUsuario` e `senha` são fornecidos pelo usuário e a operação que autentica esses dois atributos é invocada. Essa operação delega a responsabilidade de autenticar o usuário para o método `autenticar()`, que retorna um dos códigos de estado: `USUARIO_VALIDO`, `NENHUMA_SENHA`, `NENHUM_IDUSUARIO` e `USUARIO_INVALIDO`. O método `validar()` interpreta o código de estado e atualiza o atributo `situacao` com a mensagem adequada para o usuário.

Na Figura 7.2(a) é apresentado o conjunto de classes de autenticação de usuário e também uma classe de teste, usada no teste de unidade com apoio de uma ferramenta como o JUnit. O diagrama de classes possui uma unidade em teste: o método `validar()` da classe `DialogoAcesso`; as unidades das quais ela depende, no caso o método `autenticar()` da classe `ControladorAcesso`; e casos de teste, como os métodos da classe `TestarDialogoAcesso`. O Programa 7.2 ilustra a aplicação de um caso de teste, realizada em alguns passos: i) instanciação da classe que contém a unidade a ser testada; ii) definição de valores dos atributos que a unidade precisa para ser testada; iii) invocação do método a ser testado; e, iv) verificação do resultado da execução do método por meio de uma assertiva.

---

Programa 7.2

```
1  public void testarValidarUsuarioValido() {
2      DialogoAcesso dialogo = new DialogoAcesso();
3      dialogo.setIdUsuario("João");
4      dialogo.setSenha("senhaJoão");
5
6      dialogo.validar();
7
8      assertEquals("Usuário autenticado", dialogo.getSituacao());
9  }
```

---

Ao executar o teste do método `validar()`, o método `autenticar()` também está sendo testado, já que `validar()` delega a responsabilidade de autenticação para o método `autenticar()`, o que não é desejável porque o método `autenticar()` possui o próprio conjunto de casos de teste. Nota-se, então, que, para efetuar os testes na unidade é necessário isolá-la das outras unidades, o que pode ser feito por meio dos imitadores de

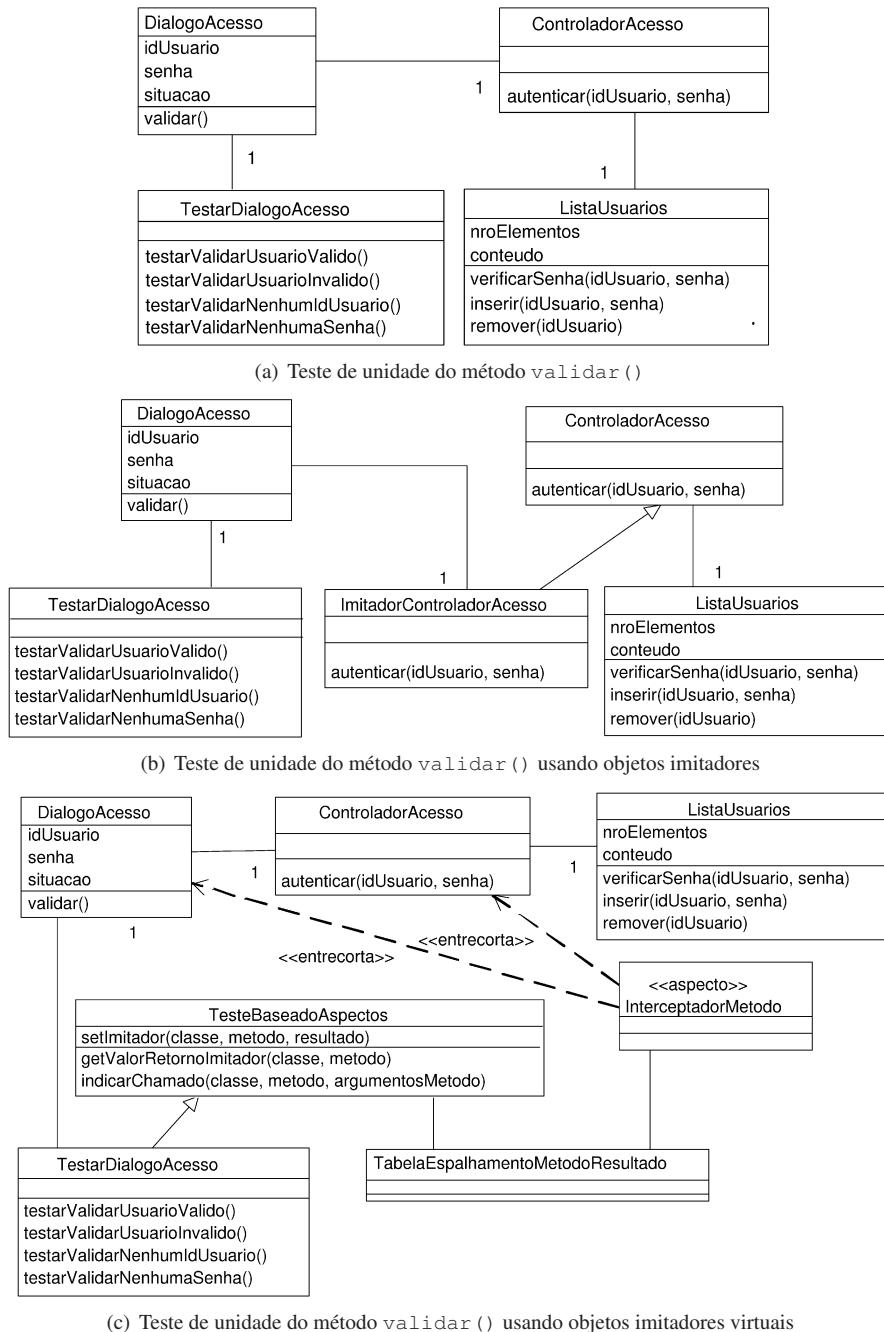


Figura 7.2 – Diferentes formas de teste de unidade de um sistema de autenticação de usuário.

objetos. A Figura 7.2(b) ilustra o uso dos imitadores de objetos para isolar a unidade em teste. Os mesmos casos de teste são utilizados para testar o método `validar()`, porém a

classe `ImitadorControladorAcesso`, que implementa um imitador, é criada para simular o comportamento do método `autenticar()` ao reescrevê-lo. Conforme ilustrado no Programa 7.3 e na Figura 7.2(b), o método é implementado para retornar um valor fixo, `USUARIO_VALIDO`, que é usado pela classe `DialogoAcesso` durante a execução do caso de teste implementado no método `testarValidarUsuarioValido()`, garantindo que apenas o método `validar` está sendo testado.

---

```
1  public int autenticar(String IdUsuario, String senha) {           Programa 7.3
2      return USUARIO_VALIDO;
3  }
```

---

Pode-se notar que o uso de objetos imitadores pode causar dois problemas: i) a classe `DialogoAcesso` deve ser alterada para que seu método `validar()` invoque o método `autenticar()` da classe `ImitadorControladorDeAcesso`, em vez do método da classe `ControladorDeAcesso`, o que exige alterações temporárias nas classes envolvidas; e, ii) o método `autenticar()` é alterado para retornar diferentes valores, dependendo do usuário e da senha usados em cada um dos casos de teste, tornando o método mais complexo.

A idéia dos imitadores virtuais de objetos é aproveitar a característica não invasiva dos aspectos para simular o comportamento desejado para o teste, em vez de alterar o projeto de classes do sistema e a implementação de seus métodos [285]. Conforme é mostrado na Figura 7.2(c), o método da classe `TesteBaseadoAspectos` usa uma tabela de espalhamento (*hashing*) para armazenar associações entre os métodos, dos quais é desejado um comportamento simulado, e seus resultados. O aspecto `InterceptadorMetodo`, Programa 7.4, é implementado para interceptar chamadas aos métodos invocados pela operação/método em teste e, com o auxílio da tabela de espalhamento, identificar quando um método simulado é invocado para retornar o valor imitador e não permitir que o método original seja executado. É interessante observar que o conjunto de junção `todasAsChamadas` especifica que devem ser selecionadas as execuções de todos os métodos do sistema, exceto aqueles dentro da classe `TesteBaseadoAspectos`.

---

```
1  aspect InterceptadorMetodo{           Programa 7.4
2      pointcut todasAsChamadas(): execution(* *.*(..)) &&
3          !within(TesteBaseadoAspectos);
4
5      Object around() : todasAsChamadas() {
6          String nomeDaClasse;
7          nomeDaClasse =
8              thisJoinPoint.getSignature().getDeclaringType().getName();
9          Object recebedor = thisJoinPoint.getThis();
10         if (recebedor != null)
11             nomeDaClasse = recebedor.getClass().getName();
12         String nomeDoMetodo = thisJoinPoint.getSignature().getName();
13         Object valorDeRetorno;
14         valorDeRetorno =
15             TesteBaseadoAspectos.getValorRetornoImitador(nomeDaClasse,
16                                                 nomeDoMetodo);
17         if (valorDeRetorno != null){
18             TesteBaseadoAspectos.indicarChamado(nomeDaClasse, nomeDoMetodo,
19                                                 getArgumentos(thisJoinPoint));
19     }
```

```

20         return valorDeRetorno;
21     }
22     else{
23         return proceed();
24     }
25 }
26 private Hashtable getArgumentos(JoinPoint jp){
27     Hashtable argumentos = new Hashtable();
28     Object[] valoresDeArgumentos = jp.getArgs();
29     String[] nomesDeArgumentos;
30     nomesDeArgumentos =
31         ((CodeSignature)jp.getSignature()).getParameterNames();
32     for (int i = 0; i < valoresDeArgumentos.length; i++){
33         if (valoresDeArgumentos[i] != null)
34             argumentos.put(nomesDeArgumentos[i], valoresDeArgumentos[i]);
35     }
36     return argumentos;
37 }
38 }

```

---

Apresenta-se no Programa 7.5 um caso de teste com a utilização de imitadores virtuais de objetos, no qual se pode notar a configuração do método interceptado e simulado – setImitador("Controlador Acesso", "autenticar", resultadoImitador). Os imitadores virtuais de objetos permitem que diferentes casos de teste de uma mesma unidade em teste definam valores de retorno distintos para métodos simulados, não existindo a criação de objetos imitadores. Assim, todo o código necessário ao teste se encontra no caso de teste, fazendo com que apenas a unidade em teste seja realmente testada.

---

Programa 7.5

```

1 public void testarValidarUsuarioValido() {
2     DialogoAcesso dialogo = new DialogoAcesso();
3
4     Integer resultadoImitador =
5         new Integer(ControladorAcesso.USUARIO_VALIDO);
6     setImitador("ControladorAcesso", "autenticar", resultadoImitador);
7
8     dialogo.setIdUsuario("João");
9     dialogo.setSenha("senhaJoão");
10    dialogo.validar();
11
12    assertEquals("Usuário autenticado", dialogo.getSituacao());
13 }

```

---

## 7.2.2 Teste embutido em componentes

O objetivo do teste embutido em componentes de software é criar interfaces para auxiliar o teste de software em componentes que possuem responsabilidades definidas por contratos e usáveis por intermédio de interfaces definidas [23]. Isso é útil para os desenvolvedores do componente, que podem efetuar os testes individuais de cada componente durante seu desenvolvimento, e também para os integradores, que podem testar os componentes em seu ambiente de uso, na presença de outras partes do sistema. Já que testabilidade pode ser vista como um interesse não funcional [48], a idéia é implementar aspectos que contenham casos de teste. O uso de aspectos faz com que o projeto e a implementação dos componentes fiquem

livres de alterações durante a atividade de teste, isto é, porque não é necessário introduzir código de teste no componente.

No teste embutido em componente auxiliado por aspectos [48], os casos de teste são implementados pelos aspectos e podem ser aplicados depois de combinados com as classes-base. No exemplo apresentado no Programa 7.6, um aspecto é criado para implementar os casos de teste do componente `ListaUsuarios`, presente na Figura 7.2(a). Com o objetivo de auxiliar o teste, o aspecto introduz um atributo para monitorar o estado do componente. Os casos de teste são implementados por meio de métodos introduzidos na classe em teste, os quais complementam a interface do componente com operações genéricas e específicas de teste, como, no exemplo, testar a inserção de um usuário na lista de usuários ou testar os três estados possíveis da `ListaUsuarios`, no Programa 7.6. Por exemplo, o caso de teste implementado no método inserido pelo aspecto `testarInserirUsuario()` testa se o estado da classe `ListaUsuarios` é valido após a inserção de um usuário no sistema de controle de acesso.

---

Programa 7.6

---

```

1  public aspect AspectoLista{
2      private String ListaUsuarios.estado = "Vazia";
3
4      public void ListaUsuarios.testarInserirUsuario(){
5          String idUsuario = "João";
6          String senha = "senhaJoão";
7          inserir(idUsuario, senha);
8          if (!apenasUm())
9              System.out.println("Problemas na inserção!");
10     }
11
12
13     public boolean ListaUsuarios.vazia(){
14         return nroElementos == 0;
15     }
16     public boolean ListaUsuarios.apenasUm(){
17         return nroElementos == 1;
18     }
19     public boolean ListaUsuarios.maisDeUm(){
20         return nroElementos > 1;
21     }
22
23     pointcut crescendo(ListaUsuarios lu) :
24             target(lu) && call(public void ListaUsuarios.inserir(..));
25
26     after(ListaUsuarios lu) : crescendo(lu) {
27         if (lu.estado.equals("Vazia"))
28             lu.estado = "Apenas um";
29         else if (lu.estado.equals("Apenas um"))
30             lu.estado = "Mais de um";
31     }

```

---

Transições de estado podem ser capturadas por meio de conjuntos de junção e tratadas nas execuções dos adendos. No exemplo, o interesse de teste relacionado à transição de estados é saber se o número de itens da `ListaUsuarios` cresce. Isso é feito por meio do conjunto de junção `crescendo(ListaUsuario lu)` (linhas 23 e 24) e do adendo `after(ListaUsuario lu):crescendo(lu)` (linhas 26-31), que ajustam o estado da lista. Assim, é possível detectar se a inserção de elementos na lista está alterando de

maneira correta o contador de elementos ou se realmente os elementos estão sendo inseridos. Nesse caso, pode-se observar que o uso dos aspectos no teste embutido em componentes acrescenta testabilidade em componentes de software, permitindo a captura e a alteração do estado do objeto, tarefa muitas vezes difícil se utilizada apenas a programação orientada a objetos.

### 7.2.3 Verificação de padrões de código

Além de fornecer mecanismos para auxiliar o teste de software, a POA pode ser usada para verificar a utilização de regras de codificação, boas práticas de programação e o uso adequado de padrões de projeto [197, 270]. As verificações no código podem ser feitas tanto em tempo de compilação quanto em tempo de execução e são muito úteis para a detecção de erros de programação ou o não-cumprimento de padrões de codificação.

No Programa 7.7 é ilustrado o uso da POA para verificar uma regra de projeto [197]. Qualquer tentativa de verificação de usuário e senha por meio do método `verificarSenha(idUsuario, senha)` da classe `ListaUsuario` – Figura 7.2(a) – deve ser feita apenas pela classe `ControladorAcesso` ou, eventualmente, por qualquer uma de suas subclasses. Para garantir essa regra, o aspecto ilustrado no Programa 7.7 exibirá um erro se esse método for invocado por um método de qualquer outra classe.

---

Programa 7.7

```
1 public aspect AspectoRegras{
2     declare error: !withincode(* ControladorAcesso.*(..)) &&
3             call(public int ListaUsuarios.verificarSenha(..));
4             "Use o método ControladorAcesso.autenticar() .";
5 }
```

---

Outro exemplo de uso de aspectos pode ser observado no Programa 7.8, em que é feita a verificação de um padrão de codificação tradicional da programação orientada a objetos: atributos devem possuir métodos de acesso que permitam sua leitura ou escrita para manter o encapsulamento [197]. Qualquer tentativa de escrita nos atributos sem a utilização de um método iniciado com ‘set’ – método padrão para a escrita de atributos – resulta em um aviso emitido pelo AspectJ.

---

Programa 7.8

```
1 public aspect AspectoPadroes{
2     declare warning: set(!public * *) &&
3             !withincode(* set*(..)):
4             "Escrevendo atributo fora do método correspondente.";
5 }
```

---

### 7.2.4 Teste de invariantes, pré e pós-condições

A técnica de projeto por contrato [281] consiste na definição de invariantes, pré e pós-condições para definir responsabilidade e pode ser aplicada com o auxílio da programação orientada a objetos. A utilização da técnica nem sempre é simples, já que, para garantir os invariantes, pré e pós-condições, pode ser necessário fazer verificações em tempo de execução. Além disso, sem a POA, o código para a verificação fica entrelaçado com o código funcional

de várias classes. O uso de pré e pós-condições também auxilia a atividade de teste, pois auxilia a revelar alguns tipos de defeitos baseados no estado do sistema antes e depois da execução.

É importante lembrar: satisfazer uma precondição não implica diretamente a invocação de um método, apenas indica que o método pode ser invocado a partir de um estado válido e que vai resultar em um estado que atenda às pós-condições especificadas, permanecendo em estado válido. Um exemplo de verificação de precondições é mostrado no Programa 7.9. O método autenticar() da classe ControladorAcesso() – Figura 7.2(a) – tem como precondição não receber como argumento valores nulos, verificada por meio do uso do adendo anterior, que lança uma exceção caso ela não seja atendida. Com a precondição atendida, a pós-condição do método é retornar um valor de autenticação válido: USUARIO\_VALIDO, NENHUMA\_SENHA, NENHUM\_IDUSUARIO e USUARIO\_INVALIDO; caso contrário, o adendo posterior lança uma exceção que indica violação da pós-condição.

---

Programa 7.9

---

```

1  public aspect AspectoChecagemPrePos{
2      pointcut autenticar(String idUsuario, String senha):
3          call(int ControladorAcesso.autenticar(String, String))
4          && args(idUsuario, senha);
5
6      before(String idUsuario, String senha):autenticar(idUsuario, senha) {
7          if ((idUsuario==null) || (senha==null))
8              throw new NullPointerException("Parâmetro inválido.");
9      }
10     after (String idUsuario, String senha) returning (int resultado):
11         autenticar(idUsuario, senha){
12             if ((resultado!=ControladorAcesso.USUARIO_VALIDO) &&
13                 (resultado!=ControladorAcesso.NENHUMA_SENHA) &&
14                 (resultado!=ControladorAcesso.NENHUM_IDUSUARIO) &&
15                 (resultado!=ControladorAcesso.USUARIO_INVALIDO))
16                 throw new RuntimeException("Resultado de autenticação inválido.");
17     }

```

---

O invariante de uma classe deve estabelecer uma regra que deve ser verdadeira durante todo o ciclo de vida das instâncias da classe, e qualquer violação da regra leva a um estado inválido. No Programa 7.10 é apresentado um exemplo de invariantes, que verifica se o atributo ‘situacao’ possui um valor permitido.

---

Programa 7.10

---

```

1  public aspect AspectoChecagemInvariante{
2      pointcut checarSituacao(String s):
3          set(private String DialogoAcesso.situacao) && args(s);
4
5      after (String s):    checarSituacao(s){
6          if ((s!="Usuário autenticado") &&
7              (s!="Usuário não autenticado") &&
8              (s!="Usuário não fornecido") &&
9              (s!="Usuário não autenticado") &&
10             (s!=null))
11             throw new RuntimeException("Situação inválida.");
12     }
13 }
```

---

### 7.2.5 Instrumentação de código

Durante as atividades de teste, é muito útil coletar informações em tempo de execução para efetuar, por exemplo, rastreamento, registro e análise de cobertura, freqüentemente instrumentando o código-fonte, uma aplicação típica da POA. Por exemplo, podem-se rastrear todas as chamadas ao método autenticar() da classe ControladorAcesso, registrando seus parâmetros, ou monitorar alterações no atributo situacao da classe DialogoAcesso, Programa 7.11. O monitoramento do atributo pode ser usado durante a depuração pelo programador para descobrir onde e por que o valor do atributo foi alterado, por exemplo, enquanto o método que monitora a autenticação pode ser usado para auxiliar em testes que visem a melhorar a segurança do sistema.

---

Programa 7.11

```
1 public aspect AspectoRastrear{
2     pointcut monitorarAutenticar(String idUsuario, String senha):
3         call(int ControladorAcesso.autenticar(String, String)) &&
4             args(idUsuario, senha);
5
6     pointcut monitorarAtributo(String s, DialogoAcesso da):
7         set(private String DialogoAcesso.idUsuario) &&
8             args(s) &&
9                 target(da);
10
11    before(String idUsuario, String senha):
12        monitorarAutenticar(idUsuario, senha){
13            System.out.print("O método autenticar foi invocado com: ");
14            System.out.println(idUsuario + " senha:" + senha + ".");
15        }
16
17    before(String s, DialogoAcesso da): monitorarAtributo(s, da){
18        System.out.print("O atributo idUsuario alterado de: ");
19        System.out.print(da.getIdUsuario());
20        System.out.print(" para ");
21        System.out.println(s + ".");
22    }
23 }
```

---

Para exemplificar o resultado do aspecto de rastreamento, considere o método do Programa 7.2. Nele, o atributo idUsuario é alterado e o método autenticar é invocado, ambos capturados pelos conjuntos de junção do aspecto de rastreamento. O resultado da execução do método do Programa 7.2 conforme a alteração do atributo e os parâmetros passados é mostrado no Programa 7.12.

---

Programa 7.12

```
1 O atributo idUsuario alterado de: para João.
2 O método autenticar foi invocado com: João senha:senhaJoão.
```

---

## 7.3 Teste de programas orientados a aspectos

Antes de definir abordagens de teste para programas OA, é interessante identificar quais tipos de defeitos acontecem em programas desse tipo. Alexander et al. [11] definiram um modelo de defeitos para programas orientados a aspectos baseado nos seguintes tipos:

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;
6. mudanças incorretas em dependências de controle.

A primeira classe de defeitos explora enganos possíveis de ser cometidos na definição de conjuntos de junção, pois o desenvolvedor pode criar conjuntos de junção que são muito (pouco) restritivos, ou seja, capturam menos (mais) pontos de junção do que deveriam. Por exemplo, no código do Programa 7.1, no aspecto de `Temporizacao` existe o conjunto de junção `finalizacao`, que designa todas as chamadas ao método que encerra conexões. Se existissem outros pontos na execução do sistema que encerrassem conexões (por exemplo, em algum tipo de exceção), esse conjunto de junção estaria sendo muito restritivo, ou seja, outros pontos de junção que deveriam ser identificados estariam sendo negligenciados.

A segunda classe de defeitos explora enganos que o desenvolvedor pode cometer ao definir a precedência entre aspectos que afetam pontos de junção coincidentes. O problema é que diferentes conjuntos de junção utilizados por diferentes adendos de diferentes aspectos (por exemplo, dois adendos posteriores de dois aspectos diferentes) podem identificar pontos de junção coincidentes, e, nesse caso, é necessário definir uma precedência entre os aspectos, o que pode ser feito erroneamente do ponto de vista semântico da aplicação. Por exemplo, na aplicação parcialmente mostrada na seção anterior, Programa 7.1, os aspectos de faturamento e de temporização devem afetar pontos de junção em comum, quando uma conexão é encerrada, de maneira que se registre o tempo final e se calcule o custo total da ligação. Para isso, o aspecto de temporização deve ter precedência sobre o de faturamento, pois o faturamento necessita do tempo total para o cálculo. Se a precedência for incorretamente definida, o sistema poderá apresentar uma falha.

A terceira e a quarta classes de defeitos exploram os enganos cometidos nos aspectos que alteram dados dos componentes que afetam, violando pós-condições e invariantes de estado previamente definidas. Por exemplo, pode haver um adendo que define comportamento em um método e altera os atributos de algum objeto, violando uma pós-condição definida para o método.

A quinta classe de defeitos explora enganos cometidos em conjuntos de junção do tipo `cflow` da linguagem AspectJ, que identifica pontos de junção dentro de fluxos de controle determinados. Pode ser que pontos de junção sejam negligenciados ou tomados por engano quando esse tipo de conjunto de junção é utilizado.

Por fim, a sexta classe de defeitos explora mudanças errôneas nas dependências de fluxo de controle, que podem ocorrer a partir do mau uso de aspectos.

### 7.3.1 Teste estrutural

Neste texto considera-se que em programas OA que são extensões de programas OO (como acontece nos escritos em 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 conjuntos de junção. Assim sendo, considerando uma única classe ou um único aspecto, já é possível pensar em teste de integração. Métodos da mesma classe ou do 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.

Se levadas em conta tais considerações, a atividade de teste de programas OA poderia ser dividida nas seguintes fases:

1. Teste de unidade, ou teste intramétodo e intraadendo, que objetiva identificar defeitos na lógica e na implementação de cada método, método introduzido ou adendo.

Uma questão importante é a execução dos adendos em isolamento, já que, de maneira geral e segundo o modelo da POA, esses são executados implicitamente, por meio de um ponto de junção. Assim, da mesma forma como Harrold e Rothermel [169] consideram a classe o driver de um método, pode-se considerar o aspecto somado a um ponto de junção que faça um adendo ser executado como o driver desse adendo, pois sem eles não é possível executá-lo (a não ser que haja alguma infra-estrutura especial para permitir a execução do adendo como se fosse um método, mas isso 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 em isolamento. Essas questões são dependentes da linguagem OA utilizada e dos mecanismos que ela apóia e serão abordadas quando necessário.

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

- Teste de módulos (classes e aspectos), que procura 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) Seqüencial, que consiste no teste das interações das unidades quando são executadas em diferentes seqüências, alterando o estado do módulo. Auxilia a aumentar a confiança de que as seqüências de execuções das unidades de um mesmo módulo interagem de maneira apropriada.
- Teste de componentes, que procura 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) Seqüencial, que consiste no teste das interações das unidades acessíveis de diferentes módulos quando são executadas em diferentes seqüências. Auxilia a aumentar a confiança de que as seqüências de execuções de unidades de módulos diferentes interagem de maneira apropriada.

O teste de programas OA interunidades tanto de módulos (no caso dos aspectos) quanto de componentes, quando considerado par a par, pode ser classificado em: intermétodo, método-adendo, adendo-método e interadendo, de acordo com os tipos das unidades que interagem entre si.

No que diz respeito ao teste de unidade, a representação mais conhecida para o estabelecimento de critérios de fluxo de controle é o Grafo de Fluxo de Controle (GFC). Para critérios de fluxo de dados é utilizado o Grafo Def-Usa, uma extensão do GFC com informação adicional sobre definições e usos de variáveis em cada nó e aresta do grafo.

Vincenzi et al. [415] definiram modelos de fluxo de dados baseados em *bytecode* Java para o teste estrutural de unidade de programas orientados a objetos implementados em Java. Nesses modelos, a informação sobre o fluxo de dados é obtida baseando-se em uma classificação das instruções de *bytecode*. A partir desse trabalho, Lemos et al. definiram um modelo para testar unidades de programas orientados a aspectos a partir do *bytecode* [235, 237]. O Grafo Def-Usa orientado a aspectos (AODU) é o modelo de fluxo de dados definido com o propósito de aplicar critérios de teste estrutural de unidade para programas orientados a aspectos escritos em AspectJ.

Para o teste de unidade desses programas, um grafo desse tipo deve ser gerado para cada método, método intertipo declarado e adendo [237], que são as unidades a serem testadas. O grafo AODU é construído com base em um Grafo de Instrução GI. Informalmente o Grafo de Instrução é um grafo no qual os nós contêm uma única instrução de *bytecode* e as arestas conectam instruções que podem ser executadas uma após a outra (inclusive quando há desvios). Informações de fluxo de dados também são obtidas para cada nó de GI [237, 415]. A partir daí, um grafo AODU de uma dada unidade  $u$  é definido como um grafo dirigido  $AODU(u) = (N, E, s, T, C)$ :

- $N$  representa o conjunto de nós de um grafo  $AODU : N = \{n \mid n$  corresponde a um bloco de instruções de *bytecode* de  $u\}$ . Isto é,  $N$  é o conjunto não-vazio de nós, que representa as instruções de *bytecode* de  $u$ .  $I_n$  é a n-tupla ordenada de instruções agrupadas no nó  $n$ ;
- $E = E_r \cup E_e$  é o conjunto completo de arestas do grafo AODU:
  - $E_r$  é o conjunto de arestas regulares definido como  $E_r = \{(n_i, n_j) \mid$  a primeira instrução de  $I_{n_j}$  pode ser executada depois da última instrução de  $I_{n_i}\}$ ;
  - $E_e$  é o conjunto de arestas de exceção definido como  $E_e = \{(n_i, n_j) \mid$  pode ocorrer uma exceção em alguma instrução de  $I_{n_i}$  e existe um tratador para essa exceção que inicia no primeiro elemento de  $I_{n_j}\}$ ;
  - $E_c \subseteq E_r$  é o conjunto de arestas transversais (*crosscutting edges*) definido como  $E_c = \{(x, y) \in E \mid (x \in C) \vee (y \in C)\}$  (componente  $C$  definida abaixo);
- $s \in N \mid IN(s) = 0$  é o nó de entrada de  $u$ .  $IN(n)$  é o número de arestas de entrada do nó  $n$ ;
- $T \subseteq N$  é o conjunto (possivelmente vazio) de nós de saída, isto é,  $T = \{n \in N \mid OUT(n) = 0\}$ .  $OUT(n)$  é o número de arestas de saída do nó  $n$ ;
- $C \subseteq N$  é o conjunto (possivelmente vazio) de nós transversais. Nesse caso um nó transversal corresponde a um bloco de instruções no qual uma das instruções representa uma invocação de um método correspondente a um adendo de um dado aspecto.

Na Figura 7.3 é mostrado o grafo AODU para representar o fluxo de controle e de dados do adendo de contorno do aspecto de transferência. O código é apresentado no Programa 7.13. O adendo de contorno tem a função de verificar se o cliente chamado transferiu para algum outro cliente o recebimento de suas chamadas, e se esse último também o fez, e assim por diante. O cliente chamado é então mudado para o transferido, se não ocorrer nenhum ciclo de transferências (ou seja, o cliente chamador ser o chamado). No fim do adendo é criada uma conexão entre o chamador e o transferido, se este existir, instanciando uma conexão local ou de longa distância de acordo com o código de área dos clientes. Como o adendo do aspecto de faturamento define comportamento no momento da criação de uma conexão, o adendo de contorno é afetado por ele, como pode ser verificado a partir dos nós transversais 82 e 159.

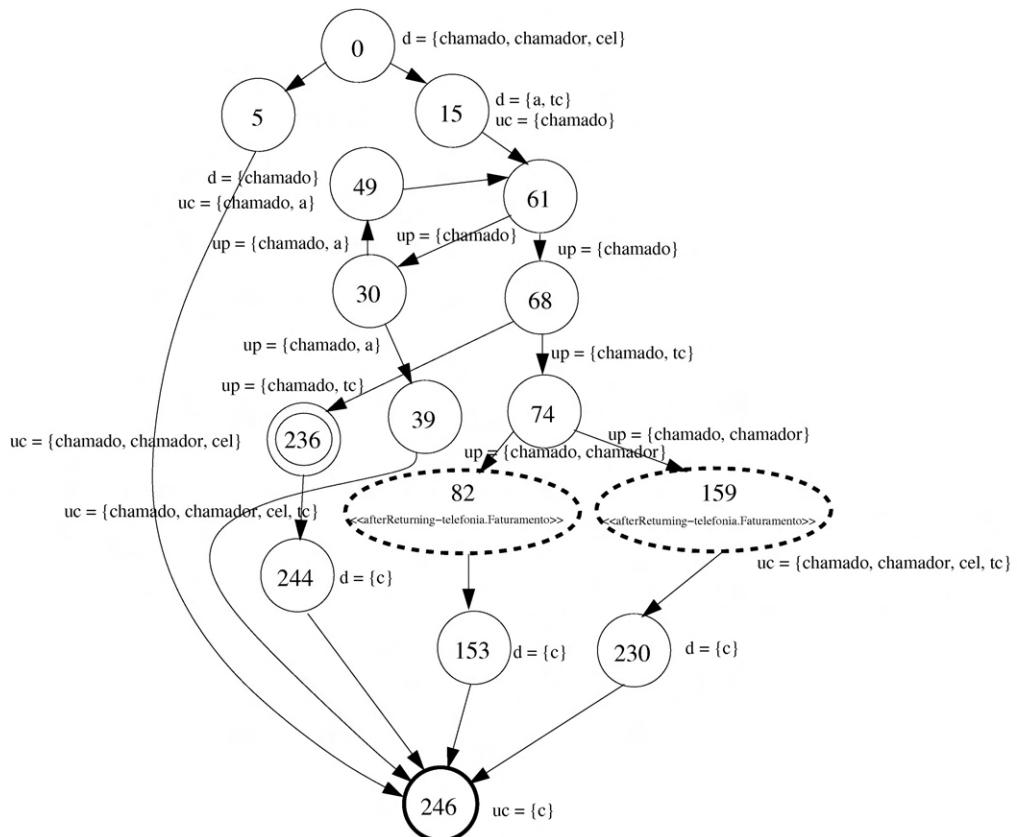


Figura 7.3 – AODU do adendo de contorno do aspecto Transferencia.

---

```

1 public aspect Transferencia {
2
3     ...
4     Conexao around (Cliente chamador, Cliente chamado, boolean cel) :
5         criaConexao(chamador, chamado, cel) {
6

```

```
7     Conexao c;
8
9     if (chamador == chamado)
10        throw new RuntimeException("Chamador não pode ser o chamado!");
11
12    ArrayList a = new ArrayList();
13
14    Cliente tc = chamado;
15
16    while (chamado.temClienteTransferencia()) {
17        if(a.indexOf(chamado) >= 0)
18            throw new RuntimeException("Ciclo de transferências!");
19        else {
20            a.add(chamado);
21            chamado = chamado.getClienteTransferencia();
22        }
23    }
24
25    if(tc != chamado) {
26        if (chamado.local(chamador)) {
27            System.out.println("Transferindo chamada de " + tc + " para "
28                + chamado);
29            c = new Local(chamador, chamado, cel);
30        }
31        else {
32            System.out.println("Transferindo chamada de " + tc + " para "
33                + chamado);
34            c = new LongaDistancia(chamador, chamado, cel);
35        }
36    }
37    else c = proceed(chamador, chamado, cel);
38
39    return c;
40 }
```

\*

O número no rótulo dos nós representa o deslocamento da primeira instrução de *bytecode* do bloco correspondente. Nós tracejados representam execuções de adendos (nós transversais) e têm informação adicional sobre o tipo do adendo que afeta aquele ponto, e o nome do aspecto ao qual pertence (por exemplo `<<afterReturning - telefonia.Faturamento >>` corresponde a um adendo posterior do aspecto Faturamento do pacote telefonia). Nós negritados representam nós de saída, nós duplos representam chamadas a métodos, e arestas tracejadas representam fluxo de controle anormal quando exceções são lançadas.

A idéia de definir novos critérios de teste é estabelecer requisitos de teste relacionados a caminhos ou elementos de um dado programa, de maneira que a execução desses caminhos/elementos aumente a probabilidade de revelar a presença de falhas na implementação. No caso de programas OA, são de interesse os elementos estruturais específicos de programas desse tipo. Como mostrado anteriormente, no modelo definido identifica-se a execução implícita de um adendo como um elemento estrutural especial, por meio dos nós transversais. Esses pontos especiais representam os pontos de junção nos quais um dado adendo executa. Identificam-se também outros elementos relacionados aos nós transversais como especiais a esse contexto: arestas que tenham nós transversais como nó início ou fim e pares definição-uso nos quais os usos estão em nós transversais.

Com relação ao grafo AODU, os seguintes conceitos são necessários para a definição dos critérios de teste:

- conjunto de nós predicativos: considere  $OUT_r(i)$  o número de arestas regulares de saída de um nó  $i$ , formalmente:  $OUT_r(i) = |\{(i, j) \in E_r\}|$ . O conjunto de nós predicativos é o conjunto  $N_{pred} = \{n \in N \mid OUT_r(n) > 1\}$ , isto é, o conjunto de todos os nós do grafo AODU que contêm mais de uma aresta regular de saída;
- caminhos livres de exceção: o conjunto de caminhos livres de exceção é o conjunto  $\{\pi \mid \forall (n_i, n_j) \in \pi \Rightarrow (n_i, n_j) \text{ é alcançável por um caminho que não contém nenhuma aresta de exceção}\}$ ;
- nós dependentes e independentes de exceção: o conjunto de nós dependentes de exceção é definido como  $N_{ed} = \{n \in N \mid \text{não existe um caminho livre de exceção } \pi \text{ tal que } n \in \pi\}$ . O conjunto de nós independentes de exceção é o conjunto definido como  $N_{ei} = N - N_{ed}$ ;
- arestas dependentes e independentes de exceção: as arestas dependentes de exceção formam o conjunto  $E_{ed} = \{e \in E \mid \text{não existe um caminho livre de exceção } \pi \text{ tal que } e \in \pi\}$ . As arestas independentes de exceção formam o conjunto  $E_{ei} = E - E_{ed}$ ;
- arestas transversais: o conjunto de arestas transversais é definido como  $E_c = \{(x, y) \in E_r \mid (x \in C) \vee (y \in C)\}$ ;
- c-usos globais e locais: um c-uso de uma variável  $x$  em um nó  $j$  é um c-uso global se não há nenhuma definição de  $x$  no mesmo nó  $j$ , em uma instrução anterior ao c-uso. Caso contrário, é um c-uso local;
- os conjuntos def, c-uso, p-uso, dcu e dpu: na implementação dos critérios de fluxo de dados, todos os usos de uma variável em um nó predutivo  $i \in N_{pred}$  são considerados p-usos. Essa decisão foi feita no trabalho de Vincenzi [415] porque é computacionalmente caro distinguir c-usos de p-usos a partir das instruções de bytecode Java, considerando a estrutura orientada a pilhas da máquina virtual Java. Assim, para um nó  $i$  e uma variável  $x$  do grafo AODU, são definidos:  
 $\text{def}(i) = \{\text{variáveis definidas no nó } i\}$   
 $\text{c-uso}(i) = \begin{cases} \text{variáveis com uso global em } i \text{ se } i \text{ é computacional} \\ \emptyset \quad \text{caso contrário} \end{cases}$   
 $\text{p-uso}(i) = \begin{cases} \text{variáveis com uso local ou global em } i \text{ se } i \text{ é predutivo} \\ \emptyset \quad \text{caso contrário} \end{cases}$   
 $\text{dcu}(x, i) = \{ \text{nós } j \text{ de um grafo AODU tal que } x \in \text{c-uso}(j) \text{ e existe um caminho livre de definição para } x \text{ de } i \text{ a } j \}$   
 $\text{dpu}(x, i) = \{ \text{arestas } (j, k) \text{ de um grafo AODU tal que } x \in \text{p-uso}(j) \text{ e existe um caminho livre de definição para } x \text{ de } i \text{ à aresta } (j, k) \}$ ;
- pares def-c-uso e def-p-uso: um par def-c-uso é uma tripla  $(i, j, x)$  em que  $i$  é um nó que contém uma definição de  $x$  e  $j \in \text{dcu}(x, i)$ . Um par def-p-uso é uma tripla  $(i, (j, k), x)$  em que  $i$  é um nó que contém uma definição de  $x$  e  $(j, k) \in \text{dpu}(x, i)$ . Um par Def-Usa é um par def-c-uso ou def-p-uso.

Dois critérios de fluxo de controle – Todos-Nós e Todas-Arestas – foram revisitados para serem aplicados nesse contexto. Considere  $\mathbf{T}$  um conjunto de casos de teste para um programa  $\mathbf{P}$  (sendo AODU o grafo de fluxo correspondente de  $\mathbf{P}$ ), e  $\Pi$  o conjunto de caminhos percorridos em  $\mathbf{P}$  a partir da execução dos casos de teste de  $\mathbf{T}$ .

O critério Todos-Nós pode ser definido no contexto de programas OA da seguinte maneira:

- Todos-Nós:  $\Pi$  satisfaz o critério Todos-Nós se cada nó  $n \in N$  do grafo AODU está incluído em  $\Pi$ . Em outras palavras, esse critério garante que todas as instruções (ou comandos) de uma dada unidade são executadas ao menos uma vez por algum caso de teste de  $\mathbf{T}$ .

Na aplicação do critério Todos-Nós em programas orientados a aspectos, uma propriedade interessante é a habilidade de saber quais dos nós cobertos são específicos desse contexto, para que seja possível o testador focar esses pontos na atividade de teste. Como observado na seção anterior, um novo tipo de interação ocorre em unidades de programas orientados a aspectos, representada pelos nós transversais.

Dessa maneira, é interessante haver um critério de teste particular que requeira a cobertura dos nós transversais, para auxiliar a revelar defeitos particulares a esses pontos. Conseqüentemente, a partir da análise de cobertura baseada nesse critério, o testador teria conhecimento sobre quando os casos de teste estariam – ou não – sensibilizando os aspectos.

A partir dessa motivação, define-se o seguinte critério:

- Todos-Nós-Transversais (Todos-Nós<sub>c</sub>)
  - $\Pi$  satisfaz o critério Todos-Nós-Transversais se cada nó  $n_i \in C$  está incluído em  $\Pi$ . Em outras palavras, esse 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  $\mathbf{T}$ .

No exemplo de grafo mostrado na Figura 7.3, os requisitos de teste referem-se aos nós transversais 82 e 159. Assim, seriam necessários casos de teste que passassem por cada um deles, para que o critério fosse satisfeito.

O critério Todas-Arestas pode ser definido no contexto de programas OA da seguinte maneira:

- Todas-Arestas:  $\Pi$  satisfaz o critério Todas-Arestas se cada aresta  $e \in E$  de um grafo AODU está incluída em  $\Pi$ . Em outras palavras, esse critério garante que cada aresta do grafo AODU de uma dada unidade é executada ao menos uma vez por algum caso de teste de  $\mathbf{T}$ .

Da mesma maneira que existem nós especiais no AODU – os nós transversais –, pode-se considerar também a existência de arestas especiais, que conectam os nós transversais. Do ponto de vista do teste, é interessante também a informação de quando tais arestas são exercitadas, seguindo a mesma idéia do critério Todos-Nós-Transversais. Essa informação é interessante porque um defeito poderia ser revelado somente quando uma aresta transversal em particular fosse escolhida. Portanto, define-se o seguinte critério:

- Todos-Arestas-Transversais (Todos-Arestas<sub>c</sub>)

- $\Pi$  satisfaz o critério Todos-Arestas-Transversais se cada aresta  $e_c \in E_c$  está incluída em  $\Pi$ . Em outras palavras, esse critério requer que cada aresta do grafo AODU que tem um nó transversal como nó início ou destino seja exercitada por algum caso de teste de  $\mathbf{T}$ .

No exemplo de grafo mostrado na Figura 7.3, os requisitos de teste desse critério referem-se às arestas transversais (74, 82), (74, 159), (82, 153) e (159, 230). Assim, seriam necessários casos de teste que passassem por cada uma delas, para que o critério fosse satisfeito.

No que diz respeito ao fluxo de dados, o critério Todos-Usos foi revistado para ser aplicado nesse contexto:

- Todos-Usos:  $\Pi$  satisfaz o critério Todos-Usos se para cada  $i \in \text{def}(i)$ ,  $\Pi$  inclui um caminho livre de definição para  $x$  de  $i$  a cada elemento de  $\text{dcu}(x, i)$  e a cada elemento  $\text{dpu}(x, i)$ . Em outras palavras, esse critério requer que cada par def-c-uso  $(i, j, x)$ , em que  $j \in \text{dcu}(x, i)$  e cada par def-p-uso  $(i, (j, k), x)$  em que  $(j, k) \in \text{dpu}(x, i)$  seja exercitado ao menos uma vez para algum caso de teste de  $\mathbf{T}$ .

Um ponto interessante a ser notado sobre o fluxo de dados de um programa orientado a aspectos é que um aspecto e uma unidade afetada por seus adendos podem trocar dados (por exemplo, quando variáveis de contexto são passadas para algum adendo). Dessa forma, outro critério para programas orientados a aspectos é exercitar os pares Def-Uso cujos usos estão em nós transversais – porque esses usos são evidências de troca de dados entre classes e aspectos. Esse critério é interessante porque interações de dados errôneas entre classes e aspectos são uma possível fonte de defeitos, como, por exemplo, os indicados no modelo de falhas mostrado anteriormente, quando pré e pós-condições e invariantes de estado são violadas. Esses tipos de violação podem estar relacionados com dados alterados a partir dos aspectos.

Quando variáveis de contexto são passadas a algum adendo, em *bytecode*, essas variáveis são adicionadas à pilha de execução (e de maneira geral isso é feito no mesmo bloco da execução do adendo, ou seja, em um nó transversal). Essas instruções de manipulação da pilha caracterizam o uso das variáveis empilhadas, de acordo com o modelo definido por Vincenzi [415]. Assim, quando variáveis de contexto – e qualquer outra informação – são passadas para adendos, no modelo utilizado é caracterizado um uso da variável em um nó transversal.

A partir daí define-se o critério Todos-Usos-Transversais:

- Todos-Usos-Transversais (Todos-Usos<sub>c</sub>)

- $\Pi$  satisfaz o critério Todos-Usos-Transversais se para cada nó  $i \in \text{def}(i)$ ,  $\Pi$  inclui um caminho livre de definição para  $x$  de  $i$  a cada elemento de  $\text{dcu}(x, i)$  que é um nó transversal e a todos elementos de  $\text{dpu}(x, i)$  nos quais o nó origem da aresta é um nó transversal. Em outras palavras, esse critério requer que cada par def-c-uso  $(i, j, x)$  em que  $j \in \text{dcu}(x, i)$  e  $j \in C$  e cada par def-p-uso  $(i, (j, k), x)$  em que  $(j, k) \in \text{dpu}(x, i)$  e  $j \in C$  seja exercitado pelo menos uma vez por algum caso de teste de  $\mathbf{T}$ .

Na Tabela 7.1 são mostrados os requisitos de teste derivados para o exemplo de grafo AODU mostrado anteriormente, que corresponde ao adendo de contorno do aspecto de Transferencia (Programa 7.13).

Tabela 7.1 – Requisitos de teste gerados a partir do grafo AODU do adendo de contorno do aspecto Transferencia

| Critério de teste          | Conjunto de requisitos   |
|----------------------------|--|
| Todos-Nós-Transversais     | {159, 82}  |
| Todas-Arestas-Transversais | {(159, 230), (82, 153), (74, 159), (74, 82)}   |
| Todos-Usos-Transversais    | {(tc, 15, 159), (chamado, 49, 159), (cel, 0, 82),<br>(chamado, 0, 82), (chamador, 0, 82), (cel, 0, 159),<br>(chamado, 0, 159), (chamador, 0, 159), (tc, 15, 82),<br>(chamado, 49, 82)} |

### 7.3.2 JaBUTi/AJ: uma ferramenta de teste estrutural

A atividade de teste, se realizada manualmente, é geralmente propensa a erros e limitada a aplicações de pequeno porte. 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.

A ferramenta JaBUTi/AJ é uma extensão da ferramenta *Java Bytecode Understanding and Testing* (JaBUTi) proposta para o teste de programas OO escritos em Java. JaBUTi/AJ foi estendida para testar a estrutura das unidades de programas OA escritos em AspectJ, utilizando os modelos e critérios descritos na seção anterior.

JaBUTi/AJ permite o uso dos critérios especiais orientados a aspectos: Todos-Nós-Transversais, Todas-Arestas-Transversais e Todos-Usos-Transversais (identificados na ferramenta como All-Nodes-c, All-Edges-c e All-Uses-c). Para cada unidade, isto é, cada método e adendo, a ferramenta permite a visualização do bytecode, código-fonte (se estiver disponível) e grafo AODU, e os requisitos de teste derivados podem ser visualizados em cada um desses meios.

Para o teste estrutural da aplicação de telefonia apresentada anteriormente, por exemplo, o testador deve primeiramente utilizar o gerenciador de projetos para criar um projeto de teste. Com esse gerenciador o testador pode escolher quais classes e aspectos deseja instrumentar e testar e um nome para o projeto de teste, entre outras configurações.

Uma vez que o conjunto de requisitos de teste de cada critério tenha sido determinado, tais requisitos podem ser utilizados para avaliar a qualidade de um conjunto de teste existente e/ou para desenvolver novos casos de teste para melhorar a cobertura dos requisitos. Por exemplo, o testador pode decidir criar um conjunto de teste com base em critérios de teste funcionais ou mesmo gerar um conjunto de teste sem nenhum método predefinido e avaliar a cobertura desse conjunto de teste em relação a cada um dos critérios de teste estruturais da ferramenta. Por outro lado, o testador pode visualizar o conjunto de requisitos de teste de cada critério, gerados para cada um dos métodos/adendos das classes/aspectos em teste,

verificar quais deles ainda não foram cobertos por algum caso de teste e então desenvolver um novo caso de teste que cubra tais requisitos. Nas Figuras 7.4(a), 7.4(b) e 7.4(c) são ilustrados os requisitos de teste do método `LineSegment.distance` apresentado no Programa 7.1, gerados pelos critérios All-Nodes-c, All-Edges-c, e All-Uses-c, respectivamente.



Figura 7.4 – Requisitos de teste do adendo de contorno do aspecto Transferencia.

A partir dos requisitos dos critérios OA, o testador pode então criar conjuntos de casos de teste para adequá-los aos critérios. Por exemplo, no caso do adendo mostrado anteriormente, para cobrir todos os nós transversais deveriam ser criados dois casos de teste. Esses casos deveriam executar o método da classe que constrói os objetos de conexão `FabricaConexao.criarConexao`, sendo que o cliente chamado deveria ter escolhido outro cliente

para receber suas ligações. Um dos casos de teste teria um cliente transferido em uma área diferente da do chamador – longa distância – e outro na mesma área – local.

A partir daí, o testador poderia continuar verificando cada um dos requisitos restantes gerados a partir dos critérios oferecidos pela ferramenta. Outra característica interessante da ferramenta JaBUTi é que os casos de teste podem ser escritos em JUnit e importados para o projeto de teste.

O teste estrutural deve ser aplicado tanto nos adendos quanto nos métodos, para que seja analisada a cobertura tanto para o código-base quanto para os aspectos. A ferramenta ainda compõe sumários dos requisitos de teste de diferentes tipos: por classe/aspecto, por método/adendo e por critério.

JaBUTi/AJ é uma ferramenta com protótipo operacional implementada em meio acadêmico; entretanto, poderia ser utilizada para aumentar a qualidade dos sistemas de software orientados a aspectos desenvolvidos também no meio industrial.

### 7.3.3 Teste baseado em estados

O teste baseado em estados para programas orientados a aspectos [449] consiste em derivar casos de teste a partir de modelos de estados. A proposta estende o método de teste de programas OO denominado FREE (*Flattened Regular Expression*) para representar não apenas classes mas também aspectos, por meio de um modelo aspectual de estados – ASM (*Aspectual State Model*). Tanto o FREE como o ASM usam a interpretação de *Statecharts* da UML – Unified Modeling Language – com extensões específicas para a atividade de teste.

O ASM permite que, primeiramente, as classes-base sejam modeladas usando a notação do FREE e, posteriormente, os aspectos sejam modelados usando a própria notação. Dessa forma, a proposta do teste baseado em estados apóia tanto o teste incremental, por meio do FREE e do ASM, como o teste conjunto de classes e aspectos.

Os objetos são entidades que enviam mensagens para outros objetos e recebem mensagens de outros objetos. O seu comportamento é determinado pelas interações, dependências e limitações nas seqüências das mensagens descritas no ASM que, portanto, descreve o estado e o comportamento dinâmico dos objetos.

A Figura 7.5(a) apresenta um modelo FREE para um objeto que representa uma conta bancária, cujos estados são: Aberta, Bloqueada, Inativa e Fechada e cujas transições são: abrir, bloquear, debitlar, creditar, obterSaldo, desativar e fechar. As relações de estado-transição determinam as mudanças de estado possíveis dos objetos da classe Conta.

A Figura 7.5(b) mostra o ASM para a classe Conta, com alterações decorrentes das seguintes mudanças nos requisitos, implementadas com o uso de aspectos:

- o proprietário da conta pode efetuar transações com saldo negativo, desde que o saldo permaneça dentro de um limite preestabelecido. Nesse estado, denominado Devedora, o cliente deve pagar juros sobre o saldo devedor.
- os débitos na conta devem ser monitorados para verificar se as transações com saldo devedor estão dentro do limite e, caso contrário, levar ao estado Bloqueada.

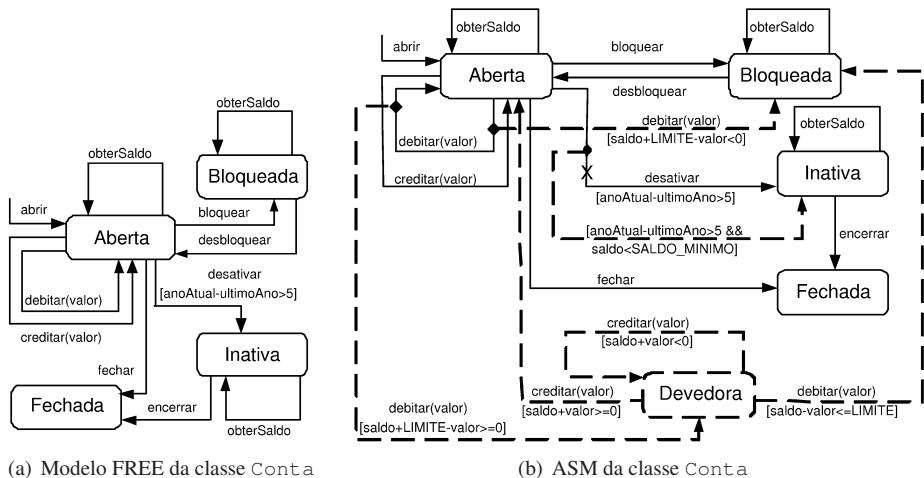


Figura 7.5 – Exemplo de modelo de estados com FREE e ASM.

- na classe Conta original, o estado Inativa é alcançado quando uma conta permanece sem movimentações por um prazo de cinco anos, porém uma nova condição estabelece que a conta se torna inativa quando permanece sem movimentações durante cinco anos e quando o saldo é menor que um valor mínimo, o SALDO\_MINIMO.

Conforme se pode observar na Figura 7.5, as classes-base são modeladas usando FREE, e os aspectos são modelados por meio de extensões da notação (ASM) que representam conjuntos de junção, pontos de junção e adendos. Os elementos gráficos da Figura 7.5(b) constituídos de linhas tracejadas são estados e transições implementados com construções sintáticas da POA. Os conjuntos de junção afetam as transições de três diferentes formas: pontos de junção de entrada, pontos de junção de saída e pontos de junção de contorno. Uma entrada em um ponto de junção é representada por uma linha tracejada, que inicia com um losango em uma transição e termina em um estado. Um exemplo de ponto de junção de entrada pode ser observado na transição debitlar(valor), que leva do estado Aberta ao estado Aberta, Figura 7.5(b). Nessa transição existe um ponto de junção de entrada com a condição  $[saldo+LIMITE-valor<0]$  que, caso satisfeita, leva ao estado Bloqueada. Naquela transição também existe um ponto de junção de saída, que leva ao estado Devedora caso a condição de guarda  $[saldo+LIMITE-valor>=0]$  seja atendida.

Um ponto de junção de contorno é representado por uma linha tracejada, que inicia com um círculo e leva a um estado. Adicionalmente, um X marca a transição que é interceptada por esse tipo de conjunto de junção para indicar um fluxo condicional entre a execução da transição e a execução do adendo relacionado ao conjunto de junção. O ponto de junção de contorno pode ser observado na transição desativar, que leva do estado Aberta para o estado Inativa caso a condição de guarda  $[anoAtual-ultimoAno>5 \& saldo < SALDO_MINIMO]$  seja atendida.

É apresentada na Figura 7.6(a) a árvore de transição de estados do ASM da Figura 7.5(b), na qual é possível observar que a raiz da árvore é o estado inicial Aberta. Essa árvore é gerada percorrendo-se todos os caminhos possíveis que partem do estado inicial do ASM da

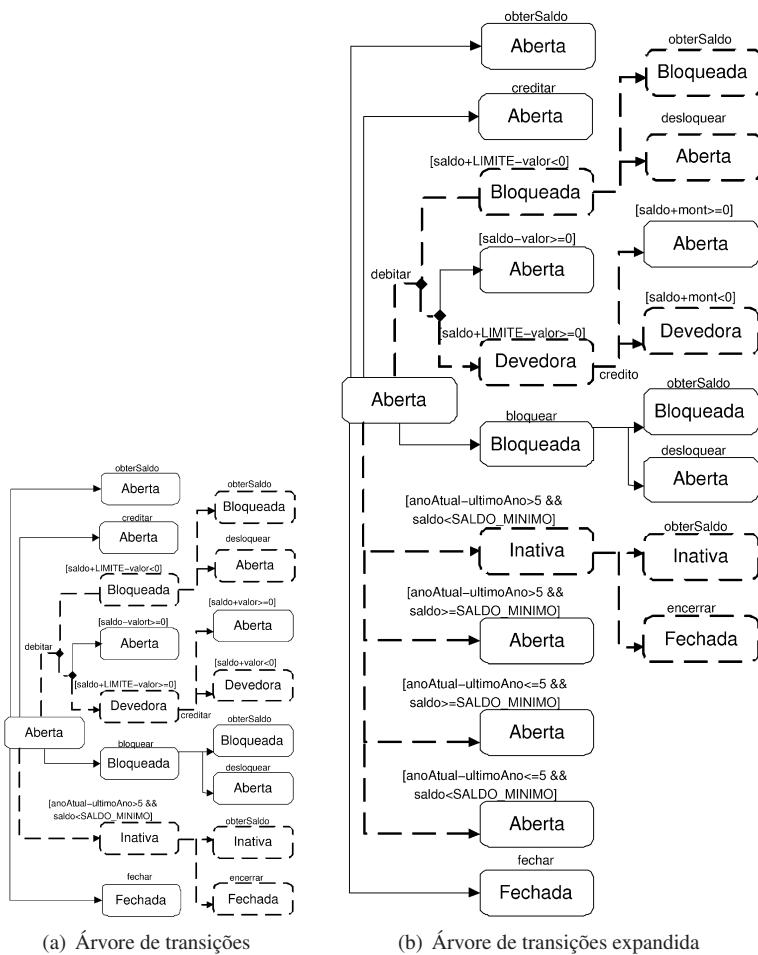


Figura 7.6 – Árvores de transições para o ASM da Figura 7.5(b).

Figura 7.5(b) e que chegam a um nó folha ou retornam ao nó inicial. O nó é considerado terminal quando ele representa um estado que está presente na árvore ou quando é um estado final.

Os caminhos representados por linhas tracejadas na árvore são gerados apenas pela presença dos aspectos e, portanto, mostram requisitos de teste específicos para os interesses transversais. Casos de teste usados para testar classes e aspectos são gerados automaticamente a partir de uma árvore de transição derivada do ASM. Cada seqüência de transições, que são os diferentes caminhos da raiz até os nós-folha da árvore de transições, constitui um requisito de teste para testar o comportamento do objeto. Por exemplo, para testar o caminho *Aberta* → *Devedora* → *Devedora* é necessário um caso de teste que execute a seqüência de mensagens *abrir()*, *debitar(valor1)*, *creditar(valor2)*, em que as condições de guarda  $[\text{saldo} + \text{LIMITE} - \text{valor} \geq 0]$  e  $[\text{saldo} + \text{valor} < 0]$  são satisfeitas e

valores adequados sejam atribuídos ao saldo, que é o saldo inicial, e aos parâmetros `valor1` e `valor2`.

Além dos casos de teste que têm por objetivo testar o comportamento esperado dos aspectos, com as condições de guarda atendidas, é necessário testar os aspectos com casos de teste que não atendam as condições associadas às transições. Para isso, é necessário analisar e identificar casos de teste adicionais para cada condição de transição. A estratégia empregada nesta tarefa é o critério de cobertura multicondicional, que tem o objetivo de exercitar todas as possibilidades de disparo de uma transição. Por exemplo, todas as condições possíveis da transição `debitar()` do estado `Aberta` estão presentes na árvore de transições. Esse não é o caso da transição `desativar`, como se pode observar na Tabela 7.2. A cobertura multicondicional para `(anoAtual-ultimoAno) <= 5 && saldo < SALDO_MINIMO` requer que outras transições sejam incluídas na árvore de transições da Figura 7.6(a), resultando na árvore de transições da Figura 7.6(b).

Tabela 7.2 – Condições da transição desativar

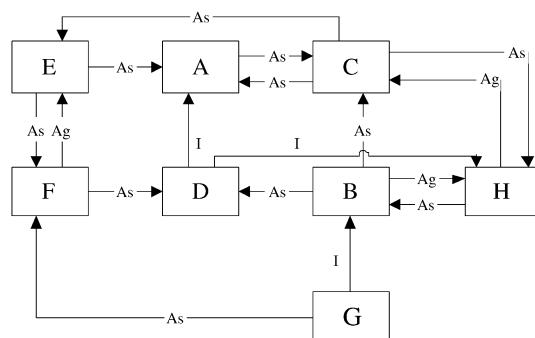
| Transição | Condição  | Próximo estado |
|-----------|---|----------------|
| desativar | <code>(anoAtual-ultimoAno) &lt;= 5 &amp;&amp; saldo &lt; SALDO_MINIMO</code>  | Aberta         |
| desativar | <code>(anoAtual-ultimoAno) &lt;= 5 &amp;&amp; saldo &gt;= SALDO_MINIMO</code> | Aberta         |
| desativar | <code>(anoAtual-ultimoAno) &gt; 5 &amp;&amp; saldo &lt; SALDO_MINIMO</code>   | Inativa        |
| desativar | <code>(anoAtual-ultimoAno) &gt; 5 &amp;&amp; saldo &gt;= SALDO_MINIMO</code>  | Aberta         |

### 7.3.4 Integração: ordenação de classes e aspectos

Um problema comum encontrado durante o teste de integração de programas orientados a objetos é a ordem pela qual as classes são testadas [384], já que isso influencia a ordem pela qual as classes são desenvolvidas, a ordem pela qual os erros são encontrados e o número de módulos pseudocontrolados e pseudocontroladores (drivers) implementados. O problema de ordenação das classes surge quando o sistema a ser testado é constituído de classes que possuem ciclos de dependência. Várias propostas foram feitas para ordenar a implementação e o teste de classes com a intenção de minimizar o número de módulos pseudocontrolados e, consequentemente, minimizar o esforço [46, 232, 229, 384]. A maioria das propostas usa o ORD (*Object Relation Diagram*) para representar as dependências entre as classes, como pode ser observado na Figura 7.7(a), em que ‘I’ representa o relacionamento de herança, ‘Ag’ representa o relacionamento de agregação e ‘As’ representa o relacionamento de associação.

A estratégia de Briand et al. [46] utiliza algumas características de outras propostas para minimizar o número de módulos pseudocontrolados: a) usa um algoritmo recursivo, conhecido como algoritmo de Tarjan, para identificar um componente fortemente conexo (CFC) no grafo [232, 386], já que o ORD pode ser visto com um dígrafo; e b) associa pesos às arestas que representam dependência de associação para indicar quais devem ser removidas e quebrar os ciclos de dependência [384].

A estratégia inicia com a aplicação do algoritmo de Tarjan em um grafo construído a partir de um ORD. Devem-se, recursivamente, remover arestas dos CFCs com mais de um nó para quebrar os ciclos e aplicar o algoritmo de Tarjan. A aresta que deve ser removida é a que apresenta maior peso, obtido por meio da multiplicação das arestas de entrada do vértice



(a) Exemplo de ORD.

$\text{ORD} = \{A, B, C, D, E, F, G\}$   
 1.  $\{A, C, E, F, D, H, B\}$  : aresta de maior peso  $A \rightarrow C = 9$   
 1.1  $\{A\}$   
 1.2  $\{B, C, E, F, D, H\}$  : aresta de maior peso  $H \rightarrow B = 9$   
 1.2.1  $\{C, E, F, D, H\}$  : aresta de maior peso  $E \rightarrow F = 4$   
 1.2.1.1  $\{E\}$   
 1.2.1.2  $\{C, H\}$  : aresta de maior peso  $C \rightarrow H = 1$   
 1.2.1.2.1  $\{C\}$   
 1.2.1.2.2  $\{H\}$   
 1.2.1.3  $\{D\}$   
 1.2.1.4  $\{F\}$   
 1.2.2  $\{B\}$   
 2.  $\{G\}$

(b) Aplicação da estratégia de Briand et al.

Figura 7.7 – Exemplo de ORD e da aplicação do algoritmo de Briand et al.

origem pelas arestas de saída do vértice destino de cada aresta de associação. Esses passos devem ser seguidos até não existirem mais CFCs não-triviais.

A Figura 7.7(b) mostra um exemplo da aplicação da estratégia em que é possível observar em um ORD as arestas que foram removidas para a quebra dos ciclos. Nesse exemplo, após a aplicação inicial do algoritmo de Tarjan, são encontrados o CFC não trivial  $\{A, C, E, F, D, H, B\}$  e o CFC trivial  $\{G\}$ . O peso das arestas é, então, calculado:

$$\begin{array}{lllll}
 H \rightarrow B = 3 * 3 = 9 & B \rightarrow D = 1 * 2 = 2 & B \rightarrow C = 1 * 3 = 3 & A \rightarrow C = 3 * 3 = 9 & C \rightarrow A = 3 * 1 = 3 \\
 C \rightarrow E = 3 * 2 = 6 & E \rightarrow A = 2 * 1 = 2 & E \rightarrow F = 2 * 2 = 4 & F \rightarrow D = 1 * 2 = 2 & C \rightarrow H = 3 * 2 = 6
 \end{array}$$

No exemplo, a remoção da aresta  $A \rightarrow C$  ou a remoção da aresta  $H \rightarrow B$  é indiferente, pois ambas geram o mesmo número de módulos pseudocontrolados, já que possuem o mesmo peso. Ao se eliminar a aresta  $A \rightarrow C$ , o algoritmo de Tarjan deve ser aplicado ao CFC  $\{A, C, E, F, D, H, B\}$ , que gera o CFC não-trivial  $\{B, C, E, F, D, H\}$  e o CFC trivial  $\{A\}$ . Novamente, uma aresta do CFC não-trivial deve ser eliminada para quebrar os ciclos.

No contexto de teste de programas orientados a aspectos, algumas estratégias propõem a implementação e o teste incremental dos programas [60, 458]. Nessa abordagem, primeiramente as classes-base devem ser implementadas e testadas e, posteriormente, os aspectos

devem ser adicionados e testados um a um para que se possam realizar as atividades do teste de integração.

Em orientação a aspectos, os tipos de dependência entre aspectos e entre classes e aspectos são diferentes dos tipos de dependência existentes em orientação a objetos. O tipo de dependência gerada por relacionamentos de herança e por relacionamentos de associação é encontrado em aspectos, ao contrário da agregação, que não existe no contexto da orientação a aspectos. Da mesma forma, a semântica e os diferentes elementos sintáticos da orientação a aspectos não possuem correspondente similar em orientação a objetos.

Os tipos de dependência são caracterizados por conjuntos de junção, adendos, declarações intertipo, herança e métodos pertencentes aos aspectos. De forma geral, os aspectos são dependentes das classes porque elas precisam existir para que o aspecto seja testado. No entanto, alguns autores [60, 210, 211, 338, 391] identificaram casos em que as classes possuem conhecimento dos aspectos que as entrecortam, o que é denominado inconsciência incompleta [138]. A inconsciência incompleta gera dependência circular [211] entre as classes e os aspectos, que é tratada nesse trabalho como dependência bidirecional. Um exemplo de dependência bidirecional pode ser encontrado na Figura 7.8, em que o aspecto Aa depende da classe D e esta depende do aspecto Aa.

Dos vários elementos sintáticos da orientação a aspectos, os tipos de dependência que diferem da orientação a objetos podem ser categorizados em dois grupos: i) dependências geradas por conjuntos de junção; e ii) dependências geradas por declarações intertipo. Os conjuntos de junção e as declarações intertipo geram tipos de dependência que não são consideradas em um ORD tradicional e são representadas em um ORD estendido por arestas do tipo ‘P’ e do tipo ‘IT’, respectivamente (Figura 7.8).

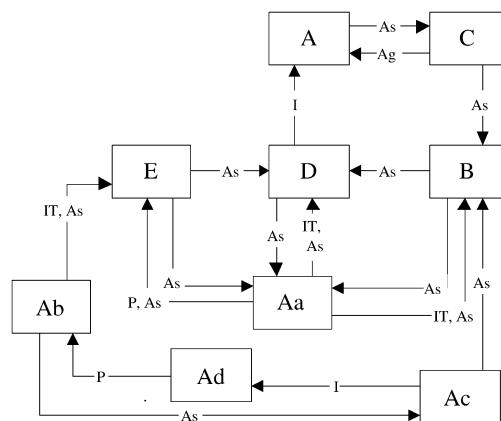


Figura 7.8 – Exemplo de ORD estendido.

A proposta de Briand et al. para ordenação de implementação e teste de classes no contexto de teste de integração pode ser aplicada ao ORD estendido. Considerando o critério de minimização de módulos pseudocontrolados, nota-se uma tendência geral em desenvolver e testar as classes antes dos aspectos, conforme pode ser observado na Figura 7.8, na Figura 7.9 e na Figura 7.10. Nesse exemplo, as classes são: A, B, C, D e E; e os aspectos são: Aa, Ab, Ac, e Ad. Quando os aspectos implementados na aplicação são todos transversais,

```

ORD={A, B, C, D, E, Aa, Ab, Ac, Ad}
1. {A, B, C, D, Aa} : aresta de maior peso D→Aa=9
   1.1 {A, B, C, D, E, Aa} : aresta de maior peso B→Aa=6
      1.1.1 {A, B, C, D} : aresta de maior peso B→D=1
         1.1.1.1 {C, A} : aresta de maior peso A→C=1
            1.1.1.1.1 {A}
            1.1.1.1.2 {C}
         1.1.1.2 {D}
         1.1.1.3 {B}
   1.1.2 {Aa, E} : aresta de maior peso Aa→E=1
      1.1.2.1 {Aa}
      1.1.2.2 {E}
2. {Ab, Ac, Ad} : aresta de maior peso Ad→Ab=1
   2.1 {Ad}
   2.2 {Ac}
   2.3 {Ab}

```

Figura 7.9 – Aplicação do algoritmo de Briand et al. ao ORD da Figura 7.8.

1. A é testado com módulo pseudocontrolado de C;
2. C é testado com A e módulo pseudocontrolado de B;
3. D é testado com A e aspecto pseudocontrolado Aa;
4. B é testado com D e aspecto pseudocontrolado Aa;
5. Aa é testado com B, D e módulo pseudocontrolado B;
6. E é testado com D e Aa;
7. Ad é testado com com aspecto pseudocontrolado Ab;
8. Ac é testado com B e Ad;
8. Ab é testado com E e Ac.

Figura 7.10 – Ordem de teste de aspectos e classes sugerida pela estratégia de Briand et al. ao ORD da Figura 7.8.

a estratégia incremental pode ser aplicada normalmente, minimizando o número de módulos pseudocontrolados. Um ponto interessante no teste incremental é que a estratégia de Briand et al. pode ser útil para ordenar e minimizar o número de módulos pseudocontrolados na integração dos aspectos, já que podem existir ciclos de dependência entre os aspectos, tornando necessária a implementação de aspectos pseudocontrolados.

Quando há aspectos bidirecionais, o emprego da estratégia incremental exige que uma ou mais classes tenham implementações alternativas que poderão ser modificadas quando a classe for integrada ao aspecto. Por exemplo, para o aspecto representado por Aa, que poderia ser um aspecto de persistência, podem-se criar objetos na memória principal, sem persistência, desenvolver e testar as classes. Isso pode ser mais trabalhoso que criar aspectos pseudocontrolados porque para o teste de integração será necessário remover tudo o que foi feito da memória interna, implementar a persistência novamente e executar mais uma vez os testes de regressão.

Com aspectos bidirecionais no modelo, como os de persistência e controle de transações, a estratégia de Briand et al. mostra que os aspectos podem ser desenvolvidos e testados logo no início, antes de qualquer classe que dele tenha consciência. Essa estratégia, em geral, gera mais módulos pseudocontrolados, mas evita retrabalho e teste de regressão. É impor-

tante notar que se houver um *framework* que implemente o interesse bidirecional, como o de persistência [338, 401], esta estratégia pode ser eficiente, pois implementam-se as classes na ordem sugerida pela estratégia de Briand et al. e instancia-se o *framework* no momento necessário, pois, em princípio, ele já está testado. Outra estratégia é quebrar a dependência bidirecional, transformando o aspecto em dois, cada um dependente da classe que, por sua vez, não tem consciência dos aspectos, e pode ser desenvolvida e testada primeiramente [211].

Caso se considere uma estratégia em que aspectos e classes são desenvolvidos conjuntamente, em determinadas situações algumas classes podem ser testadas depois de um conjunto de aspectos, o que só acontece quando elas têm consciência dos aspectos. Nesse caso, o algoritmo de Briand et al. estabelece a ordem de implementação e teste de classes e aspectos, minimizando tanto o número de módulos pseudocontrolados de classes assim como o número de aspectos pseudocontrolados.

## 7.4 Considerações finais

Este capítulo abordou duas perspectivas relacionadas à programação orientada a aspectos e o teste de software: como a POA pode auxiliar o teste de programas OO e como se podem testar programas orientados a aspectos estendendo as técnicas de teste existentes, como as técnicas estrutural e baseada em estados.

Na primeira perspectiva foram apresentadas abordagens práticas para o teste de programas OO, nas quais a POA auxilia a separar o código-base do código de infra-estrutura relacionado ao teste de software. O uso de POA para apoiar o teste em seus vários estágios, incluindo também o teste de regressão, é um tópico recente de pesquisa e deve continuar a evoluir. Novas técnicas e ferramentas devem ser desenvolvidas.

Na segunda perspectiva foram apresentadas algumas técnicas de teste primeiramente propostas para programas procedimentais e OO, adaptadas para o contexto da POA. Essa tendência deve continuar, aperfeiçoando-se as técnicas adaptadas e criando-se novas técnicas, quando necessárias. Nesse sentido, podem-se citar os novos trabalhos que vêm sendo propostos para o teste de conjuntos de junção [236] e o aperfeiçoamento do teste baseado em modelos de programas OA proposto por Xu e Xu, apresentados na Seção 7.3.3 [448, 450].

# Capítulo 8

## Teste de Aplicações Web

*Silvia Regina Vergilio (DInf/UFPR)*

*Maria Cláudia Figueiredo Pereira Emer (DCA/FEEC/UNICAMP)*

*Mario Jino (DCA/FEEC/UNICAMP)*

### 8.1 Introdução

Originalmente, aplicações *Web* possuíam o objetivo de apenas apresentar informações que consistiam basicamente em documentos no formato de texto [91]. Os *Web sites* eram compostos de arquivos HTML (*Hypertext Markup Language*) estáticos. A arquitetura utilizada era a cliente-servidor duas camadas: um cliente *Web browser* usado para visitar *Web sites* que residem em diferentes computadores servidores que enviam arquivos HTML para o cliente.

Entretanto, as aplicações *Web* tornaram-se aplicações de software para comércio eletrônico, distribuição de informações, entretenimento, trabalho cooperativo e numerosas outras atividades [304]. Elas estão cada vez mais complexas e são utilizadas em áreas críticas na grande maioria das empresas. Se essas aplicações falham, os prejuízos são enormes. Por isso, a garantia da qualidade e da confiabilidade é crucial, e a demanda por metodologias e ferramentas para realizar o teste de aplicações *Web* é crescente.

A arquitetura e as tecnologias envolvidas em uma aplicação *Web* mudaram drasticamente nos últimos anos. A configuração foi estendida do modelo cliente-servidor duas camadas para várias camadas e, embora muito semelhante às aplicações cliente-servidor tradicionais, o teste de aplicações *Web* é um pouco mais complicado, pois existem outros aspectos a considerar. O principal deles é que aplicações *Web* são dinâmicas e heterôgeneas.

A palavra heterógena é comumente utilizada para designar as diversas maneiras utilizadas pelos componentes de software para se comunicarem e as diferentes e recentes tecnologias envolvidas [230, 247, 248]. Além disso, deve-se considerar que os componentes encontram-se, na maioria das vezes, geograficamente distribuídos. Eles incluem software tradicional, programas em linguagens de *scripts*, HTML, base de dados, imagens gráficas e interfaces com o usuário complexas. Entre as principais tecnologias envolvidas, Conallen [91] destaca:

- HTTP (*HyperText Transfer Protocol*) – protocolo utilizado para especificar como um navegador (*browser*) deve formatar e enviar uma solicitação para um servidor *Web*.

- URL (*Uniform Resource Locator*) – identificador completo para referência e obtenção de um documento.
- HTML (*Hypertext Markup Language*) – linguagem para expressar a formatação visual de uma página ou um documento. A HTML possui algumas marcas ou *tags* que informam ao navegador como mostrar a página Web. A *tag anchor*, por exemplo, é utilizada para criar ligações para uma página Web. A *tag form* permite o uso de formulários que aceitam entradas dos usuários. Um formulário geralmente possui um botão especial para a sua ativação. A *tag frame* é utilizada para dividir a área de exibição do navegador em regiões ou quadros que apresentam o seu próprio arquivo HTML.
- Scripts – permitem a execução de comandos dentro de um documento HTML. Pode-se utilizar, por exemplo, *JavaScript* para escrever esses comandos.
- XML (*eXtensible Markup Language*) – linguagem de marcação para conteúdo, composta por um conjunto de elementos e atributos. Os elementos podem ser aninhados hierarquicamente. XML usa um esquema ou gramática para definir elementos, atributos e todas as regras referentes aos dados que devem ser seguidas pelo documento XML. Os esquemas mais usados são: DTD (*Document Type Definition*) [45] e XML Schema (*XML Language Schema*) [392]. Um documento XML é dito bem-formado se ele obedece a determinadas regras sintáticas, como, por exemplo: a utilização de marcas de início (`<marca>`) e de fim (`</marca>`) para identificar o conteúdo dos elementos. Um documento XML bem-formado é válido se ele segue as regras definidas por um esquema. Documentos XML podem ser passados como mensagens entre aplicações ou componentes Web para troca de informações, podem ser usados para armazenar dados e para disponibilizar dados recuperados de uma base de dados relacional.
- Web Services – ou serviços Web são uma coleção de funções ou serviços disponíveis em uma rede, a serem utilizados pelos clientes. Para que isso seja possível, é necessário o envio de um documento que especifique o tipo de serviço desejado. O SOAP (*Simple Object Access Protocol*) fornece um formato para esse tipo de documento. A linguagem WSDL (*Web Service Description Language*) é utilizada para descrever o serviço, e o UDDI (*Universal Description Discovery*) é um mecanismo utilizado para divulgar os serviços disponíveis para seus potenciais clientes.

Existem inúmeras outras tecnologias utilizadas em aplicações Web. O uso dessas tecnologias contribui para aumentar a flexibilidade, assim como a complexidade. O aspecto dinâmico das aplicações Web também gera novos desafios para o desenvolvimento de software e para a atividade de teste [445], tais como:

- o controle da execução é diretamente afetado pelo usuário. Por exemplo, ele pode, ao pressionar um botão, modificar totalmente o contexto de execução do programa;
- mudanças no cliente e no servidor ocorrem freqüentemente. Programas e dados no lado do cliente podem ser gerados e alterados dinamicamente;
- os componentes de hardware e software são heterôgeneos e devem ser integrados dinamicamente. Isso faz com que atributos de qualidade pouco críticos em aplicações tradicionais, tais como compatibilidade e interoperabilidade, sejam fundamentais em aplicações Web;

- as tecnologias envolvidas estão sempre se modificando e evoluindo. Isso requer modificações e evoluções constantes nos métodos e ferramentas de teste utilizadas;
- os requisitos da aplicação são alterados rapidamente. Por isso, o tempo de desenvolvimento e manutenções deve ser o menor possível;
- a equipe de desenvolvimento deve ser formada por pessoal com habilidades e conhecimento diversos.

Embora esses aspectos tornem a aplicação *Web* muito diferente das aplicações tradicionais, algumas abordagens de teste aplicadas ao software tradicional podem ser adaptadas ou modificadas. Offutt [304] diz que o fato de as aplicações *Web* coletarem, processarem e transmitirem dados pode ser utilizado para que o teste estrutural baseado em fluxo de dados seja explorado. Vários trabalhos exploram este contexto [230, 247, 248, 343]. A maioria das aplicações *Web* é orientada a objetos; portanto, os testes inter e intraclasse podem ser aplicados [230, 247, 248]. As próximas seções deste capítulo têm como objetivo descrever trabalhos que buscam estender as técnicas de teste existentes para permitir o teste de aplicações *Web*.

## 8.2 Teste estrutural de aplicações *Web*

O teste estrutural em aplicações *Web* tem sido explorado em alguns trabalhos pela extensão de técnicas de teste baseadas em fluxo de dados, aplicadas em software tradicional, para aplicações baseadas na *Web* [230, 247, 248].

Para a aplicação do teste estrutural, modelos são propostos para revelar defeitos associados à informação manipulada pela aplicação, ao comportamento de navegação e aos estados dependentes da interação de objetos na aplicação *Web*. Os elementos da aplicação *Web* são capturados considerando modelos de: objeto, de fluxo de controle e de dados, e de comportamento. Esses elementos são descritos em Liu et al. [248], como:

- página cliente: documento HTML ou XML com scripts embutidos, visualizado por meio de um navegador *Web* no lado cliente;
- página servidora: script CGI (*Common Gateway Interface*), ASP (*Active Server Page*), JSP (*Java Server Page*) ou um *servlet* executado pelo servidor *Web* no lado servidor;
- componente: *template HTML*, *Java applet*, *ActiveX Control*, *Java Bean* ou qualquer módulo de programa que interaja com uma página cliente, com uma página servidora ou com outro componente.

### 8.2.1 Modelo de objetos, fluxo de controle e fluxo de dados

No modelo de objeto, os elementos da aplicação *Web* são considerados objetos compostos de atributos: variáveis de programas ou elementos de documentos HTML ou XML, e operações: funções implementadas por meio de um script ou de uma linguagem de programação.

O diagrama de relação de objeto (*Object Relation Diagram* – ORD) é usado para representar os objetos e suas relações (ver Capítulo 7). As relações descritas para esses objetos são:

- $I$  – relação de herança;
- $Ag$  – relação de agregação;
- $As$  – relação de associação;
- $N$  – relação de navegação entre duas páginas clientes;
- $Req$  – relação de requisição entre uma página cliente e uma página servidora;
- $Rs$  – relação de resposta entre uma página cliente e uma página servidora;
- $Rd$  – relação de redirecionamento de solicitação entre páginas servidoras.

O diagrama ORD é um grafo cujos nós representam os objetos e cujos ramos representam os relacionamentos entre os objetos. A Figura 8.1 apresenta um exemplo de ORD para uma aplicação Web, que busca informações sobre o acervo de uma biblioteca, na qual o usuário fornece o autor, o título ou o assunto para verificar se o livro está cadastrado no acervo, via página-cliente *Bib*. A página-servidora *Bib* solicita ao componente *BuscaBib* a verificação do acervo, gerando a página-cliente *BibCompleta* com o resultado da busca, que pode levar a informações sobre o livro por meio de requisição à página-servidora *InfBib*. A página-servidora *InfBib* pede ao componente *BuscaInfBib* o carregamento das informações, gerando a página-cliente *InfBibCompleta*. As páginas geradas possuem ligação para a página-cliente *Bib*.

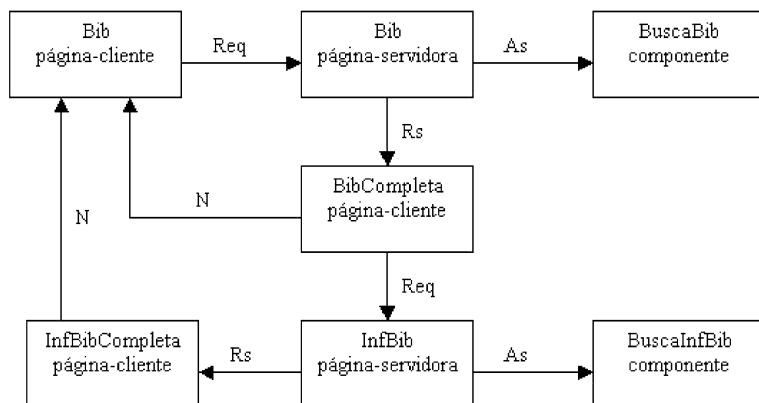


Figura 8.1 – Exemplo de diagrama ORD.

Para capturar informações sobre os fluxos de controle e de dados da aplicação Web são utilizados quatro tipos de grafos, muito similares aos utilizados em aplicações tradicionais, nos testes de unidade e de integração e no teste de software orientado a objetos.

**Grafo de Fluxo de Controle** (GFC – *Control Flow Graph*) – contém informações sobre o fluxo de controle e sobre definição e uso de variáveis. Um grafo de fluxo de controle é construído para cada função.

**Grafo de Fluxo de Controle Interprocedimental** (GFCI – *Interprocedural Control Flow Graph*) – representa informações de fluxo de dados que envolvem mais de uma função. Neste grafo são obtidas associações definição-uso para variáveis definidas em uma função e usadas em outra.

**Grafo de Fluxo de Controle de Objeto** (GFCO – *Object Control Flow Graph*) – representa as associações definição-uso resultantes de diferentes seqüências de invocação de funções no objeto. As seqüências de invocação de funções podem ser diferentes porque dependem da interação do usuário da aplicação Web. Neste grafo são visualizados todos os GFCIs ou GFCs juntos.

**Grafo de Fluxo de Controle Composto** (GFCC – *Composite Control Flow Graph*) – descreve o fluxo de informações entre páginas Web, ou seja, associações definição-uso entre essas páginas. Informações podem ser passadas de uma página Web para outra página Web quando os usuários selecionam uma ligação (*hyperlink*) ou submetem um formulário. Este grafo é composto de GFCIs ou GFCs conectados, relacionados às páginas que têm interação.

A partir desses modelos e grafos introduzidos, e considerando níveis de teste utilizados no teste de integração e de software orientados a objetos (ver Capítulos 4 e 6), Liu et al. [248] propõem uma abordagem de teste em cinco níveis.

**Nível de função** – testa as funções individualmente utilizando o GFC.

**Nível de agrupamento de funções** – testa um agrupamento de funções de um objeto. Os GFCIs são utilizados.

**Nível de objeto** – testa um objeto utilizando associações definição-uso em diferentes seqüências de invocação de funções. Neste nível de teste são usados os GFCOs. É necessário o uso de um critério para selecionar um subconjunto de seqüências de invocação de funções para obtenção das associações, pois o número de tais seqüências pode ser excessivamente grande.

**Nível de agrupamento de objetos** – testa um agrupamento de objetos utilizando associações definição-uso entre funções de objetos em um agrupamento, ou seja, um conjunto de objetos que interagem por meio de passagem de mensagens. A passagem de mensagem pode ser direta, passagem de dados por chamada de função, ou indireta, transmissão de dados entre páginas Web via protocolo HTTP. Na obtenção dos caminhos de teste, o GFCI pode ser usado para passagem de mensagem direta e o GFCC pode ser usado para passagem de mensagem indireta.

**Nível de aplicação** – testa a aplicação utilizando associações definição-uso com relação às variáveis da aplicação possíveis de serem compartilhadas e manipuladas por todos os clientes que estão acessando a aplicação. Essas variáveis são denominadas variáveis do escopo da aplicação. Os caminhos de teste são derivados a partir dos GFCs envolvendo variáveis do escopo da aplicação. Este teste é similar ao teste de programas concorrentes (ver Capítulo 9).

Os cinco níveis de teste descritos exploram associações definição-uso na perspectiva dos objetos que compõem uma aplicação Web. Os níveis de função, agrupamento de funções

e objeto correspondem à perspectiva intra-objeto; o nível de agrupamento de objetos está relacionado à perspectiva interobjeto; e o nível de aplicação corresponde à perspectiva intercliente.

### 8.2.2 Modelo de comportamento

O modelo de comportamento da aplicação *Web* deseja capturar dois aspectos: navegação entre páginas e dependência de estado na interação entre objetos.

O diagrama de navegação de página (*Page Navigation Diagram – PND*) representa o comportamento de navegação de páginas *Web*. Esse diagrama é similar a uma máquina de estados finitos, no qual cada estado representa uma página cliente e cada transição entre os estados representa uma ligação. A Figura 8.2 apresenta um exemplo de PND. Nessa figura, a Página-B*BibCompleta* e a Página-C*BibCompleta* são geradas pela mesma página servidora, de uma mesma ligação. A página que será visualizada depende da satisfação da condição especificada entre colchetes na transição.

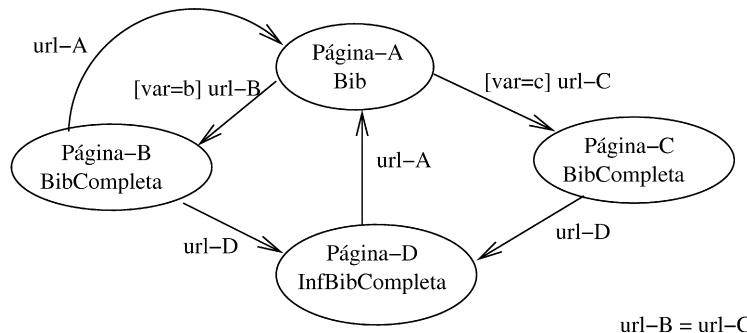


Figura 8.2 – Exemplo de PND.

O diagrama de estado do objeto (*Object State Diagram – OSD*) representa o comportamento dependente do estado de um objeto. Este diagrama é obtido da especificação dos requisitos ou da implementação da aplicação *Web*. O comportamento dependente da interação entre os objetos é descrito pelo diagrama composto de estado dos objetos (*Composite OSDs – COSD*). Um exemplo de COSD é dado na Figura 8.3, na qual são apresentados OSDs de objetos que interagem entre si na aplicação que busca informações do acervo de uma biblioteca. Esses objetos são: a página cliente *Bib*, a página servidora *Bib* e o componente *BuscaBib*. De acordo com a Figura 8.3, a página cliente pode aceitar, reinicializar ou rever entradas do usuário do acervo de uma biblioteca e submeter uma requisição para a página servidora, que extrai os dados submetidos, repassa esses dados para o componente realizar a operação requerida e retornar os resultados e, por fim, gera uma página de resposta.

Os casos de teste derivados do modelo de comportamento da aplicação *Web* são obtidos de árvores de teste de navegação construídas a partir dos diagramas PND e COSD. Na árvore de teste, os nós representam estados e os ramos representam transições entre os estados. Um caso de teste é uma seqüência de ramos que inicia no nó raiz e termina em um nó qualquer.

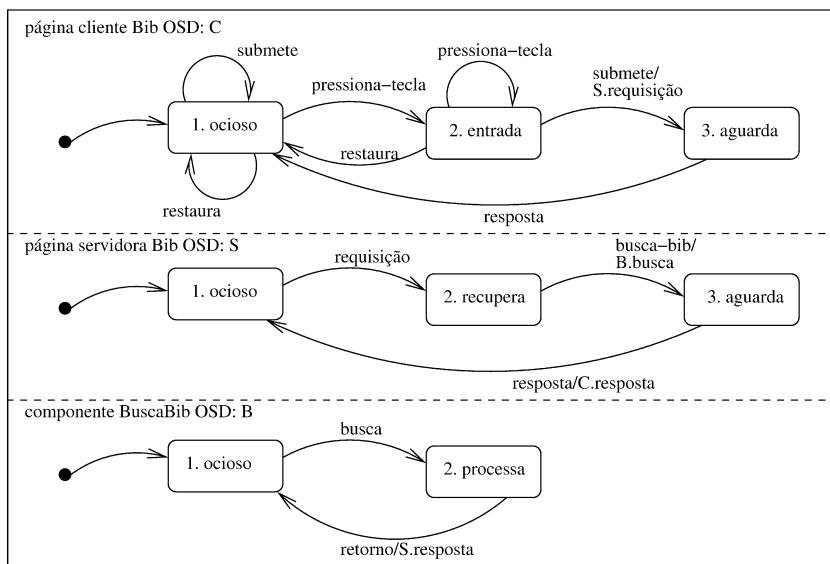


Figura 8.3 – Exemplo de COSD.

O teste de comportamento de navegação é realizado com o uso do PND. O PND gera a árvore de teste para derivar casos de teste capazes de revelar defeitos relacionados aos caminhos de navegação na aplicação *Web*. A Figura 8.4 apresenta a árvore de teste gerada a partir do PND da Figura 8.2. Com base nessa árvore de teste são identificadas possíveis seqüências de teste de navegação da Página-*ABib* para a Página-*DInfBibCompleta*.

**Exemplo 1:** Seqüências de teste de navegação obtidas a partir da árvore de teste da Figura 8.4.

1. [var=b] url-B, url-D
2. [var=c] url-C, url-D

O COSD é usado para gerar casos de teste capazes de detectar defeitos associados ao comportamento dependente do estado da interação entre objetos de uma aplicação *Web*. A Figura 8.5 mostra a árvore de teste obtida a partir do COSD da Figura 8.3. Na figura, *C* é a página cliente *Bib*, *S* é a página servidora *Bib* e *B* é o componente *BuscaBib*. Ao observar essa árvore de teste pode ser identificado um possível caso de teste.

**Exemplo 2:** Caso de teste obtido a partir da árvore de teste da Figura 8.5.

pressiona-tecla → (submete, requisição) →  
(busca-bib, busca) → (retorno, resposta, resposta)

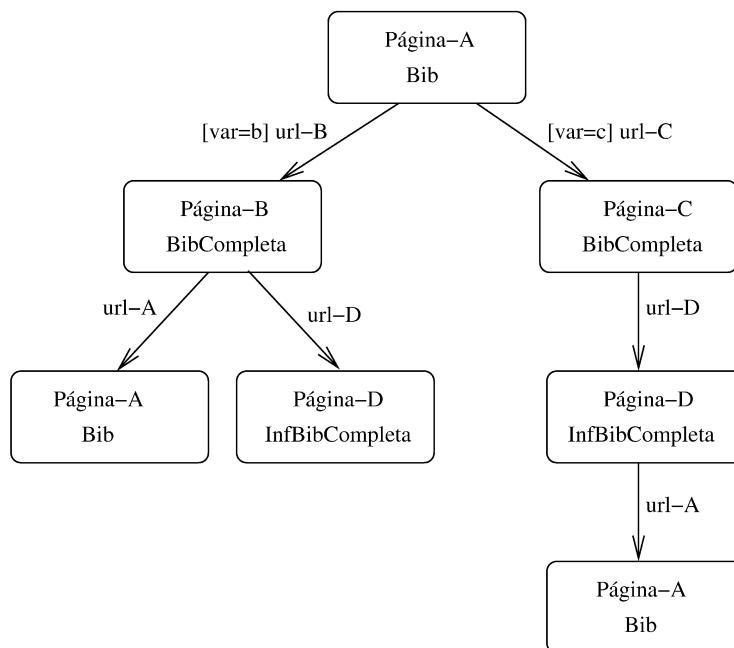


Figura 8.4 – Árvore de teste baseada no PND.

### 8.3 Teste baseado em modelos de especificação

O entendimento de qualquer tipo de sistema de software pode ser auxiliado pelo uso de modelos de representação. Em se tratando de aplicações Web, Conallen [90, 91] propõe uma extensão da UML (*Unified Modeling Language*), na qual páginas Web são mapeadas uma a uma para componentes UML.

Os trabalhos [253, 343] abordados nesta seção tratam da execução de teste em aplicações Web com base em informações advindas de modelos de representação. Esses trabalhos propõem abordagens de teste baseadas em diagramas de classes, explorando a geração de casos de teste com base nos critérios de teste estrutural e funcional para aplicações Web.

Ricca e Tonella [343] descrevem uma aplicação Web genérica por meio de um metamodelo UML que é instanciado para representar uma determinada aplicação Web. O metamodelo apresentado na Figura 8.6 é composto das seguintes classes:

- Página Web – contém informações visualizadas pelo usuário, ligações para outras páginas, formulários e frames.
- Página Estática – página Web com conteúdo fixo.
- Página Dinâmica – página Web com conteúdo dependente do processamento realizado no servidor e de informações providenciadas pelo usuário.
- Frame – região na página Web, cuja navegação pode ser independente.

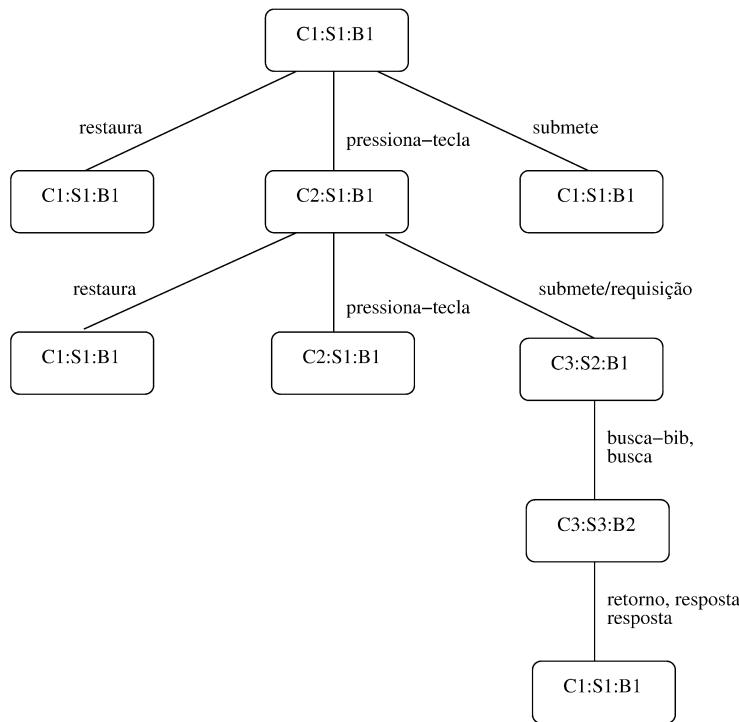


Figura 8.5 – Árvore de teste baseada no COSD.

- Página no *Frame* – carregamento de uma página em um *frame* diferente, ativado por uma ligação da página anterior.
- Formulário – conjunto de variáveis de entrada, fornecidas pelo usuário para gerar uma página dinâmica.
- Ramo Condicional – condição que provoca alterações na página *Web* quando satisfeita, por meio de valores de entrada.

Ricca e Tonella propuseram duas abordagens de teste baseadas no modelo: a estática, que analisa caminhos de navegação ou fluxo de informações fornecidas pelo usuário, e a dinâmica, que avalia a aplicação por meio da execução de casos de teste.

A realização do teste estático permite a identificação de defeitos relacionados a:

- Páginas inalcançáveis – páginas disponíveis no servidor que não são alcançadas a partir da página inicial.
- Páginas fantasmas – páginas associadas a ligações pendentes, ou seja, que indicam páginas inexistentes.
- *Frames* alcançados – *frames* nos quais páginas podem ser carregadas.

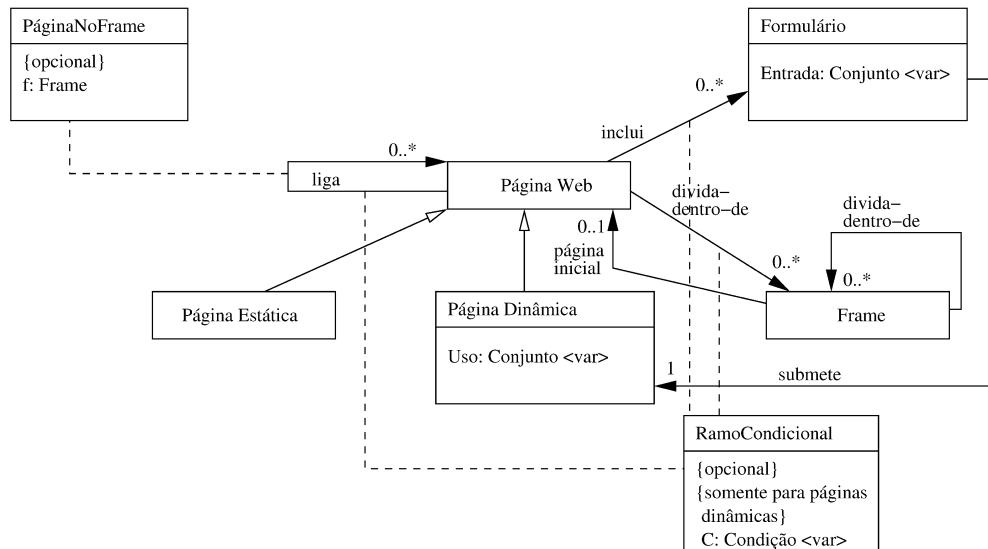


Figura 8.6 – Metamodelo para uma aplicação Web genérica [343].

- Dependência de dado – um nó do tipo formulário gera uma definição de cada variável de entrada, que é propagada pelos ramos da aplicação Web. Se a definição de uma variável alcança um nó em que a variável é usada, ocorre uma dependência de dado. A dependência de dado pode revelar o uso de uma variável ainda não definida ou o uso de uma definição incorreta de uma variável.
- Páginas cruzadas – páginas alcançadas somente percorrendo um conjunto de outras páginas, chamadas de dominadoras. Estas páginas são percorridas porque possuem informações importantes de navegação.
- Caminho mais curto – número mínimo de páginas que precisam ser visitadas para ser alcançada a página desejada.

O teste dinâmico permite a revelação de defeitos relacionados aos fluxos de controle e de dados entre as páginas da aplicação Web. O caso de teste é composto de uma seqüência de páginas da aplicação associadas aos dados de entrada. A seqüência de páginas é gerada por meio da computação de expressão de caminho, representação algébrica dos caminhos de um grafo. Os dados de entrada são determinados pelo testador por meio de técnicas de teste funcional; por exemplo, análise de valores limites. Nesse caso, o teste envolve somente páginas que coletam informações do usuário, processam dados ou mostram resultados.

Critérios de teste estrutural aplicados a programas tradicionais são estendidos para o teste de aplicações Web e empregados no teste dinâmico. Ricca e Tonella [343] propõem os seguintes critérios:

- Teste de Página (*page testing*) – cada página na aplicação deve ser visitada pelo menos uma vez em algum caso de teste.

- Teste de ligação (*hyperlink testing*) – cada ligação de cada página na aplicação deve ser percorrida pelo menos uma vez.
- Teste de Definição-Uso (*definition-use testing*) – todos os caminhos de navegação que exercitam cada associação definição-uso devem ser exercitados.
- Teste de Todos-Usos (*all-uses testing*) – pelo menos um caminho de navegação que exercita cada associação definição-uso deve ser exercitado.
- Teste de Todos-Caminhos (*all-paths testing*) – cada caminho na aplicação deve ser percorrido pelo menos uma vez em algum caso de teste.

Os critérios Teste de Definição-Uso e Teste de Todos-Caminhos nem sempre são aplicáveis, pois podem requerer um número excessivo de elementos.

Se considerada a aplicação *Web* de biblioteca, descrita anteriormente, a Figura 8.7 apresenta um exemplo, com o fragmento do metamodelo instanciado para essa aplicação, indicando os caminhos de navegação que podem ser percorridos na aplicação. A expressão de caminho de navegação  $e1(e2 + (e3\ e4\ e5(e6 + e7)))$ , obtida na Figura 8.7, juntamente com os dados de entrada, disponibilizados pelo testador, é utilizada para gerar casos de teste para satisfazer o critério Teste de Página. Veja o Exemplo 3.

**Exemplo 3:** Casos de teste para cobrir o critério Teste de Página, obtidos da instância do metamodelo para a aplicação *Web* apresentada na Figura 8.7.

1. pagWeb0, pagWeb1{usuario, senha}
2. pagWeb0, pagWeb2{usuario, senha}, pagWeb3, pagWeb4{palavra}
3. pagWeb0, pagWeb2{usuario, senha}, pagWeb3, pagWeb5{palavra}

Outro trabalho que envolve diagramas de classe no teste de aplicações *Web* é o de Di Lucca et al. [253]. Nesse trabalho é proposto um modelo de teste que emprega UML para representar uma aplicação *Web* e explorar uma abordagem para o teste de unidade e o teste de integração.

Nessa abordagem de teste o modelo de teste é descrito como um diagrama de classes, no qual as classes representam páginas *Web* da aplicação e seus subcomponentes, tais como: página *Web*, página cliente, página servidora, página estática, página dinâmica, quadro e formulário; as associações representam o relacionamento entre os componentes, tais como: usa, modifica, redireciona, constrói, submete, ativa e restaura. Também é empregado um modelo de requisitos funcionais, como um diagrama de casos de uso, para determinar o comportamento esperado para a página.

Os casos de teste para o teste de unidade e o teste de integração são gerados com base no modelo de requisitos funcionais. A seleção de casos de teste pode ser realizada, por exemplo, com o emprego de tabela de decisão ou com particionamento de conjuntos de entrada em classes de equivalência.

O teste de unidade emprega o modelo de teste para identificar os componentes da página e qualquer outro componente responsável pelo comportamento da página. Os casos de teste são executados com o auxílio de módulos pseudocontrolados construídos para tratar a interação de objetos da página com objetos externos a ela.

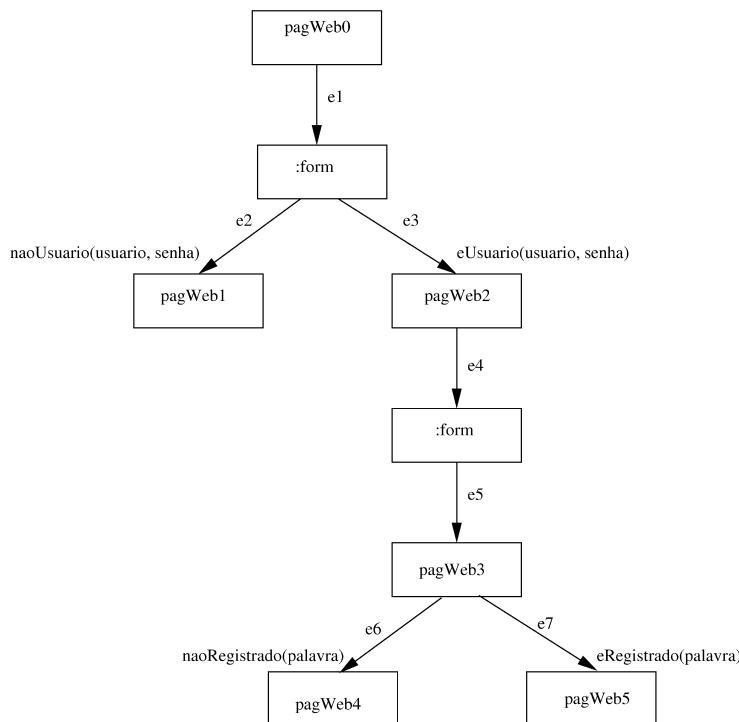


Figura 8.7 – Instância do metamodelo para a aplicação *Web* de biblioteca.

No teste de integração, o modelo de teste e o modelo de requisitos funcionais são usados para identificar a interação entre as páginas da aplicação *Web*. Nessa fase, módulos pseudocontrolados também estão envolvidos para a execução dos casos de teste. No contexto do teste de integração, o principal problema é definir como integrar páginas conectadas que formam um ciclo. Uma forma de resolver este problema é:

- gerar um grafo, baseado no modelo de teste, no qual os nós são as páginas *Web* e os seus componentes internos, e as ligações são os relacionamentos entre os nós;
- associar a cada ligação um peso (valor) proporcional ao número de parâmetros passados pela ligação;
- transformar o grafo em um grafo acíclico, eliminando ligações com pesos menores;
- computar uma ordem topológica para o grafo acíclico;
- executar o teste de integração, conforme a ordem topológica.

## 8.4 Teste baseado em defeitos

Critérios de teste baseados em defeitos também são utilizados para testar alguns aspectos de aplicações *Web*; por exemplo, a interação entre componentes *Web* por meio de mensagens XML e o esquema associado a um documento XML.

### 8.4.1 Mutação de interação em documentos XML

A abordagem de teste proposta por Lee e Offutt [233] aplica análise de mutantes [115] para a validação da interação de dados por meio de mensagens XML entre componentes de aplicações *Web*. Essa abordagem de teste é denominada mutação de interação.

Qualquer processo de software ou combinação de processos executados na *Web* é definido como componente *Web*. Esta definição inclui: *Java Server Pages*, *Java Servlets*, *JavaScripts*, *Active Server Pages*, bases de dados e outros.

Uma mensagem XML pode ser vista como uma árvore, na qual os vértices  $V$  representam nomes de elementos e nomes de atributos, e as folhas representam conteúdo. Os nomes dos elementos e os nomes de atributos são definidos em um esquema específico, nesse caso, em uma DTD.

A abordagem de teste de mutação de interação usa um modelo de especificação de interação (*Interaction Specification Model* – ISM) para auxiliar na geração dos mutantes de mensagens XML. Esse modelo é composto da DTD, que especifica a estrutura da mensagem XML, de uma interação entre componentes *Web* (mensagem XML de requisição e mensagem XML de resposta) e do conjunto de restrições XML.

Além de estabelecer um modelo de interação baseado nas mensagens XML trocadas pelos componentes da aplicação *Web*, é necessário criar operadores de mutação. Os operadores de mutação de interação (*Interaction Mutation Operators* – IMO) são regras aplicadas às mensagens XML para obtenção de mutantes, ou seja, de mensagens XML modificadas segundo as regras. Mensagens XML contêm elementos e atributos, especificados em uma DTD conforme o domínio da aplicação. Portanto, não é possível criar operadores de mutação que trabalhem em qualquer aplicação. Dessa forma, as restrições especificadas no modelo de interação são usadas para definir classes genéricas de operadores de mutação que podem ser empregadas em aplicações baseadas em DTDs diferentes. As classes são instanciadas para gerar operadores específicos para cada DTD. Inicialmente, duas classes de operadores de mutação foram desenvolvidas baseadas em restrições XML:

- NOT `memberOf` – criada a partir da restrição *memberOf*, relacionada aos filhos de um elemento da mensagem XML;
- NOT `lenOf` – criada a partir da restrição *lenOf*, associada ao tamanho (número de caracteres na *string*) do conteúdo de um elemento da mensagem XML.

O processo de teste é realizado em uma interação  $I$  definida no modelo de interação por meio de um conjunto de casos de teste. Um caso de teste é uma instância mutante de uma interação na forma de mensagem XML, na qual um operador de mutação foi aplicado. Os operadores de mutação são gerados especificamente para a DTD definida no modelo de

interação por meio das classes genéricas de operadores de mutação, usadas para que sejam detectados na DTD elementos nos quais as restrições possam ser aplicadas. No teste, os casos de teste são executados para a interação entre os componentes *Web* que está sendo testada. O resultado obtido com a execução das mensagens XML mutantes é analisado, de modo que os mutantes sejam classificados como vivos, mortos ou equivalentes.

Considere a aplicação *Web* que busca informações sobre o acervo de uma biblioteca; suponha que para acessar essa aplicação um usuário precise fornecer um nome de usuário e uma senha, que devem ser validados. A mutação de interação poderia ser aplicada na mensagem de requisição de validação do usuário. A classe genérica do operador de mutação NOT memberOf poderia ser empregada no elemento usuário associado ao elemento senha, supondo que essa associação tenha sido especificada no modelo de interação. A classe genérica do operador de mutação NOT lenOf poderia ser empregada no elemento senha, supondo que esse elemento tenha sido especificado no modelo de interação com lenOf igual a seis.

**Exemplo 4:** Mensagens XML de requisição original e mutantes, conforme os operadores de mutação NOT memberOf e NOT lenOf, e mensagens XML de resposta correspondentes. Veja Figura 8.8.

|  |  |
|--|--|
| <b>Mensagem de requisição original</b>   | <b>Mensagem de resposta</b>  |
| <pre>&lt;?xml version=1.0?&gt; &lt;requisicao&gt;     &lt;usuario&gt;Pedro&lt;/usuario&gt;     &lt;senha&gt;Rs12aM&lt;/senha&gt; &lt;/requisicao&gt;</pre> | <pre>&lt;?xml version=1.0?&gt; &lt;resposta&gt;     &lt;validacao&gt;aceita&lt;/validacao&gt;     &lt;usuario&gt;Pedro&lt;/usuario&gt; &lt;/resposta&gt;</pre> |
| <b>Mensagem de requisição mutante aplicando o operador NOT memberOf</b>  | <b>Mensagem de resposta</b>  |
| <pre>&lt;?xml version=1.0?&gt; &lt;requisicao&gt;     &lt;usuario&gt;Paulo&lt;/usuario&gt;     &lt;senha&gt;Rs12aM&lt;/senha&gt; &lt;/requisicao&gt;</pre> | <pre>&lt;?xml version=1.0?&gt; &lt;resposta&gt;     &lt;validacao&gt;negada&lt;/validacao&gt;     &lt;usuario&gt;Paulo&lt;/usuario&gt; &lt;/resposta&gt;</pre> |
| <b>Mensagem de requisição mutante aplicando o operador NOT lenOf</b>   | <b>Mensagem de resposta</b>  |
| <pre>&lt;?xml version=1.0?&gt; &lt;requisicao&gt;     &lt;usuario&gt;Pedro&lt;/usuario&gt;     &lt;senha&gt;Rs12&lt;/senha&gt; &lt;/requisicao&gt;</pre>   | <pre>&lt;?xml version=1.0?&gt; &lt;resposta&gt;     &lt;validacao&gt;negada&lt;/validacao&gt;     &lt;usuario&gt;Pedro&lt;/usuario&gt; &lt;/resposta&gt;</pre> |

Figura 8.8 – Mensagens XML de requisição e resposta do Exemplo 4.

#### 8.4.2 Teste de mutação e serviços *Web*

O trabalho de Offutt e Xu [315] explora a perturbação de dados em XML para gerar casos de teste para aplicações que utilizam serviços *Web*. Nessa abordagem a interação entre pares de serviços *Web* é testada por meio da mutação de mensagens de requisição em formato XML.

O modelo de interação de serviços *Web* considerado no teste é dois a dois, ou seja, dois serviços *Web* se comunicando por meio de chamada de procedimento remota (*Remote Procedure Call – RPC*) ou documento XML usando SOAP.

Nessa abordagem, a mutação de mensagem XML por perturbação de dados é realizada de duas formas: perturbação de valor de dado e perturbação de interação.

A perturbação de valor de dados (*Data Value Perturbation* – DVP) altera o conteúdo de elementos em mensagens SOAP, considerando as definições feitas para os dados dos elementos no esquema XML e aplicando valores limites para tipos primitivos de dados, tais como: *decimal*, *float*, *double*, *string* e *boolean*. Os casos de teste são criados por alterações dependentes do tipo de dado. Por exemplo, elementos do tipo *string* podem ser modificados em relação aos tamanhos mínimo ou máximo permitidos para o seu conteúdo, conforme a definição especificada no esquema associado à mensagem XML.

A perturbação de interação altera mensagens em RPC (mensagens com valores de argumentos para funções remotas) e em comunicação de dados (mensagens cuja finalidade é transferir dados).

A perturbação de comunicação RPC (*RPC Communication Perturbation* – RCP) está baseada no teste do uso de dados (valores usados dentro de programas) e no teste do uso de SQL (valores usados como entrada para uma consulta em uma base de dados). Esses testes aplicam operadores de mutação para fazer pequenas alterações nos valores dos dados, ou seja, no conteúdo dos elementos das mensagens XML de comunicação.

Operadores de mutação para uso de dados de tipo numérico são:

- **Divisão( $n$ )** (*Divide( $n$ )*): altera o valor  $n$  por  $1 \div n$ , sendo  $n$  do tipo *double*;
- **Multipliação( $n$ )** (*Multiply( $n$ )*): altera  $n$  por  $n \times n$ ;
- **Negativo( $n$ )** (*Negative( $n$ )*): altera  $n$  por  $-n$ ;
- **Absoluto( $n$ )** (*Absolute( $n$ )*): altera  $n$  pelo valor absoluto de  $n$ .

Operadores de mutação para uso de SQL são:

- **Troca( $n1, n2$ )** (*Exchange( $n1, n2$ )*): troca o valor de  $n1$  com  $n2$  e vice-versa,  $n1$  e  $n2$  do mesmo tipo;
- **NãoAutorizado( $str$ )** (*Unauthorized( $str$ )*): troca o valor de  $str$  por  $str$  “OR” 1“=”1.

Considerando ainda a aplicação *Web* que busca informações sobre o acervo de uma biblioteca, suponha que a validação do usuário e da senha seja realizada por meio de mensagem SOAP, na qual os conteúdos dos elementos relacionados a usuário e senha são usados em uma consulta SQL. A seguir é dado um exemplo de uso do operador *Unauthorized()*, nesse contexto.

**Exemplo 5:** Corpo das mensagens SOAP de requisição original e mutante, conforme o operador de mutação *Unauthorized()*, e consulta SQL usando o conteúdo modificado por esse operador de mutação. Veja Figura 8.9.

A perturbação de comunicação de dados (*Data Communication Perturbation* – DCP) está baseada no teste de relacionamento e restrições definidas para os dados em formato XML por meio do esquema. A manipulação de relacionamento e restrições aos dados é

**Corpo da mensagem SOAP de requisição original**

```
.....  
<soapenv:Body>  
<adminLogin soapenv:encodingStyle=  
    "http://schemas.xmlsoap.org/soap/encoding/">  
    <arg0 xsi:type="xsd:string">Pedro</arg0>  
    <arg1 xsi:type="xsd:string">Rs12aM</arg1>  
</adminLogin>  
</soapenv:Body>  
.....
```

**Corpo da mensagem SOAP de requisição mutante, após aplicação do operador de mutação Unauthorized()**

```
.....  
<soapenv:Body>  
<adminLogin soapenv:encodingStyle=  
    "http://schemas.xmlsoap.org/soap/encoding/">  
    <arg0 xsi:type="xsd:string">Pedro</arg0>  
    <arg1 xsi:type="xsd:string">Rs12aM "OR" 1 "=" 1</arg1>  
</adminLogin>  
</soapenv:Body>  
.....
```

**Consulta SQL com conteúdo modificado pelo operador de mutação Unauthorized()**

```
SELECT usuario FROM usuarios  
WHERE usuario="Pedro" AND senha="Rs12aM" OR "1"="1"
```

Figura 8.9 – Corpo de mensagem SOAP e Consulta SQL do Exemplo 5.

feita usando um modelo formal RTG (*Regular Tree Grammar*), que representa um esquema e deriva documentos XML. O modelo formal possui um conjunto finito de: elementos, tipos de dados, atributos, não-terminais e regras de produção que determinam os relacionamentos e as restrições.

O teste quanto ao relacionamento está focado na integridade referencial entre elementos-pai e elementos-filho especificada no esquema XML por meio de operadores que indicam a quantidade máxima de ocorrência de um elemento; esses operadores são: “?” (zero ou um), “+” (pelo menos um) e “\*” (zero ou mais). Com base nesses operadores, são gerados os casos de teste:

- operador “?” – gera dois casos de teste, um contendo uma instância do elemento associado ao operador e outro contendo uma instância vazia do elemento associado ao operador;
- operador “+” – gera dois casos de teste, um contendo uma instância do elemento associado ao operador e outro contendo um número permitido de instâncias do elemento associado ao operador;

- operador “\*” – gera dois casos de teste, um contendo a duplicação de uma instância do elemento associado ao operador; e outro contendo a destruição de uma instância do elemento associado ao operador.

O teste quanto às restrições em um esquema está relacionado à associação pai-filho entre elemento não-terminal e elemento terminal. Nesse caso, as alterações feitas nas mensagens XML obedecem às restrições definidas na recomendação de tipo de dados XML do W3C (*W3C's XML datatype recommendation*) [38]. Portanto, a estratégia de teste é usar as restrições a determinado tipo de dado para trocar o valor do dado. Por exemplo: se um elemento contendo o dado referente ao mês, declarado como *monthType* no esquema, tem por restrição o total de dois dígitos, para gerar os casos de teste, essa restrição seria considerada.

### 8.4.3 Teste de esquemas XML

O teste de esquemas deve revelar defeitos que poderiam ocasionar uma falha na aplicação *Web* que manipula dados de um documento XML associado ao esquema em teste. Os defeitos detectados em um esquema estão relacionados à definição incorreta ou à definição ausente de restrições aos dados armazenados em documento XML, permitindo que dados incorretos possam ser considerados válidos ou que dados corretos possam ser considerados inválidos.

A abordagem de teste de esquemas proposta por Emer et al. [129] é baseada em defeitos. Essa abordagem permite a detecção de defeitos em esquemas de documentos XML por meio de consultas a documentos XML válidos em relação ao esquema em teste.

Os defeitos que podem ser encontrados em um esquema estão associados aos erros mais comuns cometidos na fase de desenvolvimento ou durante a evolução de um esquema, por causa de atualizações ocorridas no documento XML associado ao esquema. Esses defeitos são identificados em três classes, subdivididas em tipos de defeitos baseados em restrições que podem ser estabelecidas para os dados.

**Classe 1 – Elementos:** defeitos relacionados à definição do elemento em um esquema, referentes a: ordem em que os elementos devem aparecer no documento XML; associação de um valor padrão ou fixo para o elemento; e quantidade de vezes que um elemento pode ocorrer no documento XML.

**Classe 2 – Atributos:** defeitos relacionados à definição de atributos de um elemento em um esquema, referentes ao: uso do atributo, que pode ser opcional ou obrigatório; e valor do atributo, que pode ser padrão ou fixo.

**Classe 3 – Restrições aos dados:** defeitos relacionados à definição de restrições de dados para elementos ou atributos de um esquema, referentes a: tipo de dado declarado; valores enumerados; valores máximos ou mínimos; seqüência de caracteres que um elemento pode assumir; manipulação de espaço em branco; quantidade de caracteres que o elemento pode conter; e quantidade de dígitos ou dígitos decimais que um número pode conter.

Para que os componentes elementos, atributos e restrições aos dados de um esquema possam ser manipulados para a detecção de defeitos por meio de consultas, é necessário que eles sejam formalmente definidos. Portanto, o modelo formal de gramática de árvore

regular (*Regular Tree Grammar* – RTG), apresentado na abordagem anterior, foi estendido para representar um esquema, identificando os componentes necessários para a aplicação da abordagem de teste. A RTG estendida é uma 7-tupla  $\langle E, A, D, R, N, P, n_s \rangle$ , na qual:

$E$  é um conjunto finito de elementos.

$A$  é um conjunto finito de atributos.

$D$  é um conjunto finito de tipos de dados.

$R$  é um conjunto finito de restrições.

$N$  é um conjunto finito de não-terminais.

$P$  é um conjunto finito de regras de produção:

- $n \rightarrow a < d >$ , em que  $n$  é não-terminal em  $N$ ,  $a$  é um atributo em  $A$  ou um elemento em  $E$  e  $d$  é um tipo de dado em  $D$ .
- $n \rightarrow a < d r >$ , em que  $n$  é não-terminal em  $N$ ,  $a$  é um atributo em  $A$  ou um elemento em  $E$ ,  $d$  é um tipo de dado em  $D$  e  $r$  é uma restrição em  $R$  para  $a$ .
- $n \rightarrow e < m r >$ , em que  $n$  é não-terminal em  $N$ ,  $e$  é um elemento em  $E$ ,  $m$  é uma expressão regular composta de não-terminais e  $r$  é uma restrição em  $R$  para  $e$ .
- $n_s$  é o não-terminal inicial,  $n_s \in N$ .

A RTG estendida  $G_1 = \langle E, A, D, R, N, P, n_s \rangle$ , gerada do *XML Schema*, que define a estrutura do documento XML contendo dados de alunos, como: número de registro do aluno, disciplina, nota, freqüência, é apresentada no Exemplo 6.

**Exemplo 6:** RTG estendida que representa o esquema de um documento XML com dados de alunos.

$E = \{ \text{alunos}, \text{aluno}, \text{disciplina}, \text{nome-disciplina}, \text{nota}, \text{freqüência} \}$

$A = \{ \text{num-aluno} \}$

$D = \{ \text{xs:string}, \text{xs:decimal}, \text{xs:integer} \}$

$R = \{ \text{complexType}, \text{type}, \text{maxOccurs}, \text{use} \}$

$N = \{ n_s, n_1, n_2, n_3, n_4, n_5, n_6 \}$

$P = \{ n_s \rightarrow \text{alunos} < n_1, \text{complexType} >,$

$n_1 \rightarrow \text{aluno} < n_2 n_6, \text{complexType}, \text{maxOccurs} >,$

$n_2 \rightarrow \text{disciplina} < n_3 n_4 n_5, \text{complexType}, \text{maxOccurs} >,$

$n_3 \rightarrow \text{nome-disciplina} < \text{xs:string} >,$

$n_4 \rightarrow \text{nota} < \text{xs:decimal} >,$

$n_5 \rightarrow \text{freqüência} < \text{xs:integer} >,$

$n_6 \rightarrow \text{num-aluno} < \text{xs:integer} > \}$

As consultas a documentos XML válidos são escritas em linguagem de consulta para documento XML, no caso a linguagem de consulta *XQuery* [39, 208] é empregada. Essas

consultas são definidas de acordo com as classes de defeito previamente introduzidas, ou seja, cada defeito está relacionado a uma consulta, cujo resultado permite que o defeito seja revelado. Portanto, existem consultas previamente definidas que são instanciadas para documentos XML distintos e, consequentemente, para um elemento ou atributo específico.

O processo de teste de esquemas pode ser visualizado na Figura 8.10. Inicialmente, o esquema em teste e um ou mais documentos XML associados a este esquema são providenciados pelo testador ou extraídos da aplicação Web. O próximo passo é a construção do modelo RTG estendido para que as classes de defeito presentes no esquema sejam identificadas e elementos ou atributos sejam associados a essas classes de defeito. As classes de defeito associadas a elementos ou atributos são selecionadas para que sejam geradas as instâncias das consultas aos documentos XML mutantes válidos.

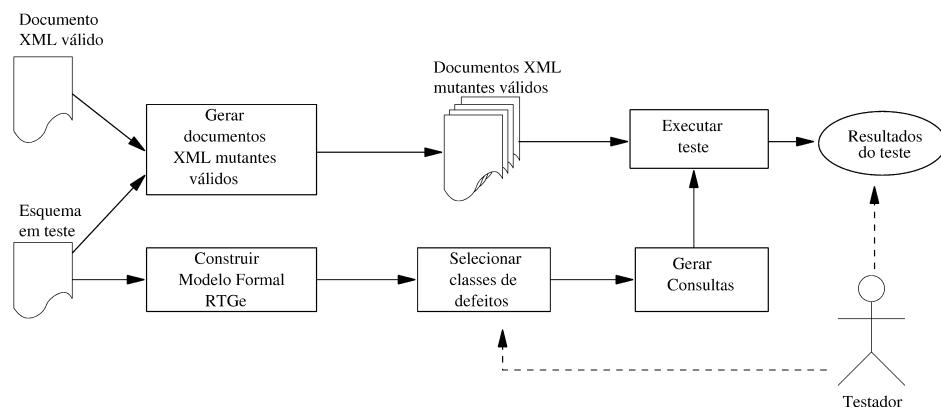


Figura 8.10 – Processo de teste para esquema XML [129].

Os documentos XML mutantes são gerados por meio de pequenas alterações feitas nos documentos XML válidos, obtidos juntamente com o esquema em teste. Essas alterações são feitas em conformidade com as classes de defeito. É importante ressaltar que os mutantes a ser consultados devem ser bem-formados e válidos com respeito ao esquema em teste.

A próxima etapa é realizar o teste executando as consultas aos documentos XML mutantes e analisando os resultados obtidos. Um dado de teste é composto de um documento XML mutante válido e da instância da consulta gerada. O resultado obtido deve ser comparado com a especificação do resultado esperado, que pode ser conseguido da especificação do esquema em teste ou da aplicação Web que usa o documento XML relacionado ao esquema em teste.

## 8.5 Ferramentas

Muitas das extensões para os critérios de teste no contexto de aplicações Web descritas anteriormente ainda são objeto de pesquisa e não possuem o suporte de uma ferramenta. Entre as ferramentas encontradas destacam-se as ferramentas propostas para as abordagens baseadas em teste de modelos de especificação, mais particularmente no diagrama de classes.

Ricca e Tonella [343] propuseram ferramentas para auxiliar na construção do modelo UML, na geração e na execução dos casos de teste. Estas ferramentas são denominadas, respectivamente, *ReWeb* e *TestWeb*.

A ferramenta *ReWeb* é usada para que as páginas da aplicação *Web* sejam obtidas e analisadas, gerando a instância do metamodelo UML. Essa ferramenta possui três módulos:

- Vasculhador (*Spider*): obtém todas as páginas de uma aplicação *Web* a partir de um URL.
- Analisador (*Analyzer*): usa a instância do metamodelo UML para analisar pontos que podem ser explorados durante o teste estático.
- Visualizador (*Viewer*): providencia uma interface gráfica para a visualização da instância do metamodelo UML da aplicação *Web* em teste.

A ferramenta *TestWeb* gera e executa o conjunto de casos de teste, conforme a instância do metamodelo UML gerado pela *ReWeb* e o critério de teste que será empregado. Após a execução dos casos de teste, o testador avalia os resultados obtidos, isto é, verifica se a saída obtida está correta para determinada entrada. A ferramenta também fornece uma saída numérica especificando o nível de cobertura alcançado pelo conjunto de teste.

O processo de teste é semi-automático e envolve intervenções do testador. Na Figura 8.11 é apresentado o processo, com as ferramentas *ReWeb* e *TestWeb*. Os losangos indicam as intervenções do testador.

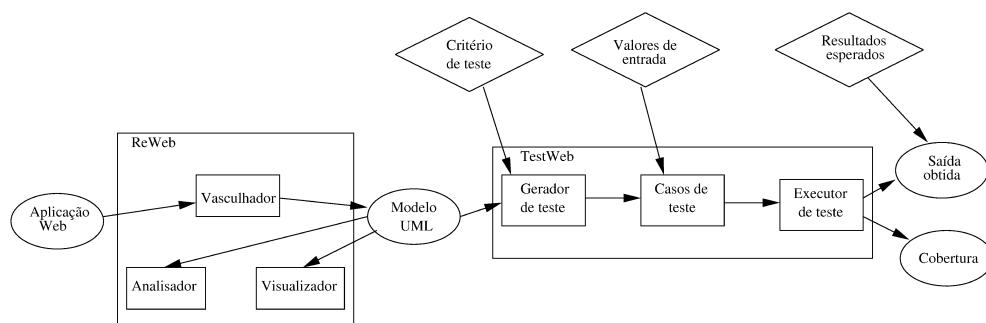


Figura 8.11 – Processo de teste com a *ReWeb* e a *TestWeb* [343].

A ferramenta denominada WAT (*Web Application Testing*) foi desenvolvida por Di Lucca et al. [253] para apoiar sua abordagem de teste em aplicações *Web*, descrita anteriormente. A ferramenta implementa as funções necessárias à geração de casos de teste, execução do conjunto de casos de teste gerado e avaliação dos resultados obtidos com a execução dos casos de teste. A WAT utiliza a análise estática de uma aplicação *Web* resultante da execução da ferramenta WARE [252], que realiza engenharia reversa em uma aplicação *Web*.

A WAT é composta por (Figura 8.12):

- Interface – permite ao testador acessar as funções oferecidas pela ferramenta.

- Serviço (*Service*) – compõe as ferramentas de serviço:
  - Extrator do modelo de teste (*Test Model Abstractor*): produz uma instância do modelo de teste para a aplicação Web com base nas informações obtidas da WARE;
  - Gerenciador de casos de teste (*Test Case Manager*): fornece funções de suporte ao projeto de casos de teste e gerenciamento da documentação do teste;
  - Gerador de módulo pseudocontrolado (*Driver and Stub Generator*): produz o código das páginas Web implementando os módulos pseudocontrolados;
  - Gerenciador de instrumentação de código (*Code Instrumentation Manager*): executa os casos de teste e armazena os resultados obtidos;
  - Analisador do resultado do teste (*Test Result Analyzer*): analisa e avalia os resultados obtidos com a execução dos casos de teste;
  - Gerador de relatório (*Report Generator*): fornece o relatório com a cobertura dos componentes Web exercitados no teste.
- Repositório (*Repository*) – base de dados relacional, na qual são armazenados: o modelo de teste da aplicação Web, os casos de teste, os registros de teste, os arquivos das páginas Web instrumentadas, os módulos pseudocontrolados e os relatórios de teste.

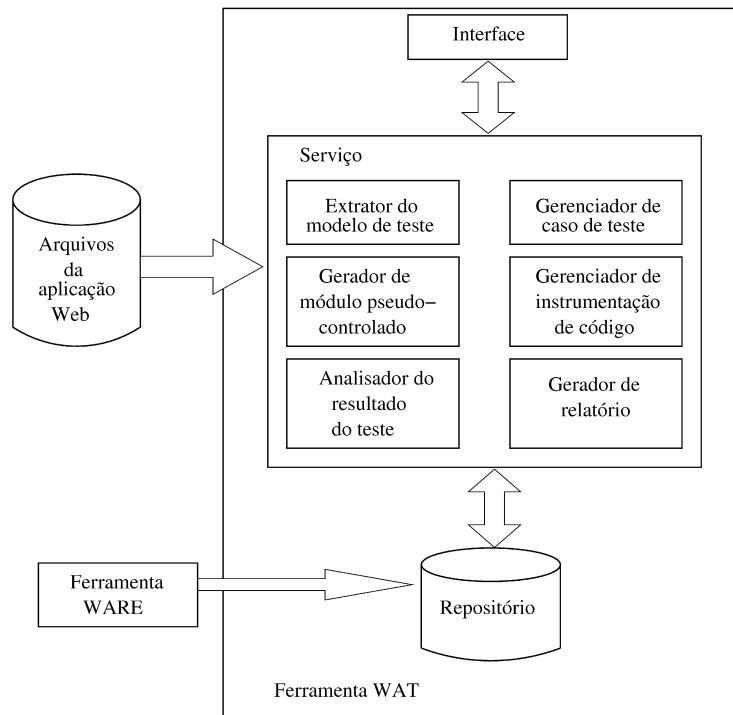


Figura 8.12 – Arquitetura da ferramenta WAT [253].

## 8.6 Considerações finais

As aplicações *Web* tornaram-se indispensáveis e também complexas. Por causa disso, introduzir abordagens e ferramentas de teste para essas aplicações é essencial.

Dada a importância do teste de aplicações *Web*, os trabalhos descritos neste capítulo são fundamentais, pois significam um ponto de partida para aumentar a qualidade dessas aplicações e torná-las mais confiáveis, além de facilitar a atividade de teste, reduzindo seu custo. Entretanto, ainda existem muitas questões não abordadas que podem ser exploradas em trabalhos futuros.

Os testes funcional, estrutural e baseado em defeitos podem ser empregados com sucesso em aplicações *Web* para revelar diferentes tipos de defeitos, tanto na especificação como no código. Basicamente, os trabalhos descritos introduzem modelos para capturar diferentes aspectos de uma aplicação *Web* visando à execução do teste baseado em fluxos de controle e de dados e do teste de mutação.

É importante ressaltar que nenhuma das abordagens de teste de aplicação *Web* descritas neste capítulo é completa, no sentido de que a maioria delas depende da tecnologia envolvida na aplicação em teste, uma tecnologia particular requer um tipo de teste específico, algumas tecnologias não foram exploradas, aspectos relacionados à concorrência e sincronização não foram considerados na maioria das abordagens e ainda existem poucas ferramentas de suporte para o teste de aplicações *Web*. As ferramentas descritas na Seção 8.5 são de fato protótipos, e muitas das abordagens, por não possuírem suporte automatizado, ainda não foram efetivamente avaliadas. Outras questões relacionadas à atividade de manutenção e teste de regressão ainda precisam ser consideradas.

# Capítulo 9

## Teste de Programas Concorrentes

*Simone do Rocio Senger de Souza (ICMC/USP)*

*Silvia Regina Vergilio (DInf/UFPR)*

*Paulo Sérgio Lopes de Souza (ICMC/USP)*

### 9.1 Introdução

Nos capítulos anteriores foram discutidos critérios de teste propostos para programas seqüenciais. Programas seqüenciais possuem um comportamento chamado determinístico, ou seja, toda execução de um programa com a mesma entrada irá produzir sempre a mesma saída. Isso acontece porque os comandos do programa são executados seqüencialmente, e os desvios, selecionados deterministicamente de acordo com o valor de entrada informado.

Diferentemente dos programas seqüenciais, programas concorrentes possuem comportamento não-determinístico, uma característica que torna a atividade de teste ainda mais complexa. O não-determinismo permite que diferentes execuções de um programa concorrente com o mesmo valor de entrada possam produzir diferentes saídas corretas. Esse comportamento é produzido devido à comunicação, à sincronização e ao paralelismo existentes entre os processos que fazem parte do programa concorrente. Essas características distinguem os programas concorrentes dos programas seqüenciais e precisam ser consideradas durante a atividade de teste de software.

Apesar de existirem novos desafios a serem considerados para testar programas concorrentes, observa-se que o conhecimento adquirido no contexto de programas seqüenciais pode ser aproveitado, com as devidas adaptações, para a definição de técnicas, critérios e ferramentas para o teste de programas concorrentes. Neste capítulo procura-se ilustrar essas contribuições e as limitações impostas para o uso dos critérios de teste propostos para programas seqüenciais.

Em geral, o teste de software aplicado a programas concorrentes visa a identificar erros relacionados à comunicação, ao paralelismo e à sincronização. Assim, critérios de teste são definidos para identificar erros dessas categorias. Entretanto, os demais tipos de erros, classificados para programas seqüenciais [31], também devem ser considerados pelos critérios de teste para programas concorrentes. Krawczyk e Wiszniewski [227] apresentam uma possível classificação de erros para programas paralelos, registrando a importância de obser-

var também a classificação de erros de Beizer [31]. A taxonomia de erros para programas concorrentes é discutida e ilustrada neste capítulo.

Na próxima seção (Seção 9.2) são apresentadas as principais características dos programas concorrentes. Um exemplo é utilizado para ilustrar as diferenças entre programas seqüenciais e concorrentes. Na Seção 9.3 são ilustrados os tipos de erros comuns em programas concorrentes. Nas Seções 9.4 e 9.5 são apresentados critérios de teste definidos para esse tipo de programa e na Seção 9.6 são discutidas questões relacionadas ao não-determinismo, juntamente com algumas soluções para gerenciá-lo. A Seção 9.7 mostra algumas ferramentas de suporte aos critérios de teste definidos. A Seção 9.8 apresenta as considerações finais do capítulo.

## 9.2 Programas concorrentes

A programação seqüencial possui instruções executadas seqüencialmente, desvios condicionais e incondicionais, estruturas de repetição e sub-rotinas. Para determinadas aplicações esses recursos implicam processamento lento e subutilização dos recursos de hardware. A programação concorrente tem por objetivo otimizar o desempenho dessas aplicações, permitindo assim uma melhor utilização dos recursos de hardware disponíveis.

A idéia básica da programação concorrente, portanto, é que determinadas aplicações sejam divididas em partes menores e que cada parcela resolva uma porção do problema. Para essa divisão, há a necessidade de certos recursos adicionais que são responsáveis, por exemplo, pela ativação e pela finalização dos processos concorrentes, e que não estão disponíveis na programação seqüencial.

Desse modo, programas concorrentes são tipicamente formados por vários processos que interagem entre si para gerar um resultado. Entende-se por processo um programa em execução, que consiste em um programa executável, seus dados, o contador de instruções, os registradores e todas as informações necessárias para sua execução [385].

A concorrência existe quando, em determinado instante, dois ou mais processos tiverem começado sua execução, mas não tiverem terminado [13]. Por essa definição, concorrência pode ocorrer tanto em sistemas com um único processador quanto em sistemas com múltiplos processadores. Afirmar que processos estão sendo executados em paralelo implica a existência de mais de um processador, ou seja, paralelismo ocorre quando há mais de um processo sendo executado no mesmo intervalo de tempo e em sistemas com multiprocessadores. Quando vários processos são executados em um único processador, sendo que somente um deles é executado a cada vez, tem-se um pseudoparalelismo. O usuário tem a falsa impressão de que suas tarefas são executadas em paralelo, mas, na realidade, o processador é compartilhado entre os processos. Isso significa que, em determinado instante, somente um processo é executado, enquanto os outros que já foram iniciados aguardam a liberação do processador para continuar.

Segundo Almasi [13], três fatores são fundamentais para a execução paralela:

- definir um conjunto de subtarefas para serem executadas em paralelo;
- capacidade de iniciar e finalizar a execução das subtarefas;

- capacidade de coordenar e especificar a interação entre as subtarefas enquanto estiverem executando.

Para os processos concorrentes serem ativados, faz-se necessário um conjunto de ferramentas que determine exatamente qual porção do código será paralela e qual será seqüencial. A coordenação e a especificação da interação entre os processos concorrentes são feitas pela comunicação e pela sincronização entre eles. A comunicação permite que a execução de um processo interfira na execução de outro e pode ser feita ou pelo uso de variáveis compartilhadas (quando se tem memória compartilhada), ou pelo uso de passagem de mensagens (quando se tem memória distribuída). A sincronização é necessária para que o acesso simultâneo não torne os dados compartilhados inconsistentes e para tornar possível o controle na seqüência da execução paralela.

Várias notações têm sido propostas para a ativação de processos concorrentes. A notação *fork/join* foi uma das primeiras a ser criada, apresentando a característica de não separar a definição de ativação dos processos da definição de sincronização [14]. Outras propostas (como *parbegin/parend*, *doall*, *co-rotinas*) distinguem bem o conceito de sincronização e ativação de processos e possuem restrições sintáticas que impedem algumas das execuções possíveis com *fork/join*. Esses aspectos estão fora do escopo deste livro. Para mais detalhes, ver as referências [13, 14].

Os mecanismos de coordenação (ou seja, comunicação e sincronização) foram desenvolvidos para atuar em um dos dois grandes grupos: nas arquiteturas com memória compartilhada ou nas arquiteturas com memória distribuída [13, 194, 298]. A sincronização de processos concorrentes que utiliza memória compartilhada tem dois objetivos. O primeiro é o controle de seqüência no qual, para determinados trechos do programa, é estabelecida a ordem de execução dos processos. O segundo objetivo é a exclusão mútua, ou seja, estabelecer um controle de acesso a determinadas áreas do programa (chamadas regiões críticas), para que dois ou mais processos não tenham acesso simultaneamente, por exemplo, a variáveis de memória, tornando-as potencialmente inconsistentes. *Busy-waiting*, semáforos e monitores são exemplos de mecanismos utilizados para a coordenação de processos concorrentes que utilizam memória compartilhada.

A sincronização de processos concorrentes que utiliza memória distribuída é realizada por meio de mecanismos como comunicação ponto a ponto, *rendezvous* e chamada de procedimento remoto (RPC). Quando a passagem de mensagens é utilizada, os processos enviam e recebem informações em vez de compartilhar variáveis. A troca de mensagens é feita por meio de primitivas *send/receive* ou por meio de chamadas de procedimentos remotos (RPC) [13, 194, 298]. A sintaxe dessas primitivas é: *send(msg, destino)* e *receive(msg, origem)*.

Os conceitos de síncrono/assíncrono e bloqueante/não-bloqueante variam dependendo do autor ou do contexto. De uma maneira geral, uma comunicação é considerada síncrona quando o *send* (ou *receive*) aguarda que o *receive* (ou *send*) seja executado. Isso é realizado por meio de confirmações (*ack*) a cada troca de mensagens. Nesse contexto, toda comunicação síncrona é bloqueante. Na comunicação assíncrona, o comando *send* (ou *receive*) não necessita esperar o comando *receive* (ou *send*) executar. Uma possível implementação para comunicação assíncrona é por meio de *buffers*. A mensagem é copiada/lida do *buffer*, que esconde dos processos a falta de sincronização. O sistema operacional tem papel fundamental, pois, normalmente, esses *buffers* são gerenciados por ele.

Os conceitos de bloqueante/não-bloqueante podem ser similares aos de síncrono/assíncrono, como descrito anteriormente. No entanto, um *send* pode ser considerado bloqueante se garantir que o *buffer* com a mensagem possa ser reutilizado logo após a liberação do *send*, sem que a mensagem seja perdida (ou sobreposta). Isso não garante a sincronização, apenas que o conteúdo da mensagem não será corrompido pelo próprio processo transmissor. Novamente aqui o uso de *buffers* é fundamental. Considerando o *receive*, ele será bloqueante até que a mensagem esteja disponível para o processo receptor no *buffer*. Mais uma vez, não há garantia de sincronização, apenas do recebimento da mensagem. Neste livro, utilizaremos esses conceitos para bloqueante/não-bloqueante. Para obter mais detalhes, ver as referências [366, 385].

No mecanismo de comunicação ponto a ponto, dois processos concorrentes executam as primitivas *send/receive*. Esse mecanismo é caracterizado por apresentar comunicação síncrona e unidirecional. No mecanismo *rendezvous* dois processos concorrentes executam as primitivas *send/receive* duas vezes, apresentando comunicação síncrona e bidirecional.

O desenvolvimento de algoritmos paralelos requer ferramentas eficazes para ativar os mecanismos que iniciam/paralisam e coordenam processos concorrentes. Almasi [13] cita três tipos de ferramentas para a construção de algoritmos paralelos: ambientes de paralelização automática, linguagens de programação concorrente e extensões para linguagens seqüenciais. Os ambientes de paralelização automática geralmente utilizam compiladores paralelizadores, que geram algoritmos paralelos a partir de algoritmos seqüenciais. As linguagens de programação concorrente possuem comandos específicos necessários a algoritmos paralelos. CSP e ADA são exemplos dessas linguagens. As extensões para linguagens seqüenciais normalmente são implementadas por meio de ambientes de passagem de mensagens (ou interfaces de passagem de mensagens), que são bibliotecas de comunicação que permitem a elaboração de aplicações paralelas. A aplicação paralela consiste em um conjunto de processos que executam programas escritos em uma linguagem de programação seqüencial, como C ou Fortran, estendida por meio de uma biblioteca de funções, que inclui comandos para enviar e receber mensagens.

Os ambientes de passagem de mensagens foram desenvolvidos inicialmente sem a preocupação com portabilidade; cada fabricante desenvolveu o próprio ambiente. Atualmente, existem muitos ambientes, independentemente da máquina a ser utilizada. Exemplos desses ambientes são PVM, p4, Express e PARMAKS [55, 57, 139, 149]. Entre esses, destaca-se o PVM (*Parallel Virtual Machine*), o qual muitos autores chamam de padrão de fato para o desenvolvimento de aplicações paralelas [149] e o MPI (*Message Passing Interface*) [366]. O MPI surgiu numa tentativa de padronizar os ambientes de passagem de mensagens e, desse modo, procura reunir as características encontradas nos outros ambientes.

Os códigos dos Programas 9.1 e 9.2 ilustram uma aplicação concorrente em PVM para o cálculo do máximo divisor comum (mdc) entre três números. A aplicação é formada por quatro processos paralelos: o mestre (Programa 9.1), responsável por ativar os demais, e três processos que executam o Programa 9.2. No Programa 9.1, após a leitura de três entradas, ocorre a criação dos processos escravos (linha 6). Cada escravo calcula o mdc para dois valores. Por meio do comando `pvm_recv()` (linha 4 do Programa 9.2) os processos escravos ficam bloqueados até que o processo mestre envie os valores para o cálculo. Após isso, os escravos enviam o valor calculado para o mestre pelo comando `pvm_send()` (linha 15 do Programa 9.2) e finalizam sua execução. O cálculo pode envolver os três processos escravos ou somente dois, dependendo dos valores de entrada. Independentemente disso, o recebi-

mento dos valores enviados pelos processos escravos é não determinístico, ou seja, depende da ordem de chegada no processo mestre. Para isso, dois comandos `pvm_recv()` não determinísticos (linhas 15 e 17) são definidos no processo mestre. Após o cálculo do mdc, o resultado é apresentado pelo processo mestre, que finaliza todos os processos criados que ainda estiverem executando (linha 32 do Programa 9.1).

---



---

```

1 int main() { // Programa mestre
2     int x,y,z;
3     int S[3];
4 /*1*/     printf("Entre com x, y e z: ");
5 /*1*/     scanf("%d%d%d",&x,&y,&z);
6 /*1*/     pvm_spawn("gcd",0,0,"",3,S);
7 /*2*/     pvm_initsend(PvmDataDefault);
8 /*2*/     pvm_pkint(&x, 1, 1);
9 /*2*/     pvm_pkint(&y, 1, 1);
10 /*2*/     pvm_send(S[0],1);
11 /*3*/     pvm_initsend(PvmDataDefault);
12 /*3*/     pvm_pkint(&y, 1, 1);
13 /*3*/     pvm_pkint(&z, 1, 1);
14 /*3*/     pvm_send(S[1],1);
15 /*4*/     pvm_recv(-1,2);
16 /*4*/     pvm_upkint(&x, 1, 1);
17 /*5*/     pvm_recv(-1,2);
18 /*5*/     pvm_upkint(&y, 1, 1);
19 /*6*/     if ((x>1)&&(y>1)){
20 /*7*/         pvm_initsend(PvmDataDefault);
21 /*7*/         pvm_pkint(&x, 1, 1);
22 /*7*/         pvm_pkint(&y, 1, 1);
23 /*7*/         pvm_send(S[2],1);
24 /*8*/         pvm_recv(-1,2);
25 /*8*/         pvm_upkint(&z, 1, 1);
26 /*8*/     }
27 /*9*/     else {
28 /*9*/         pvm_kill(S[2]);
29 /*9*/         z = 1;
30 /*9*/     }
31 /*10*/     printf("%d", z);
32 /*10*/     pvm_exit();
33 /*10*/}

```

---



---



---



---

```

1 int main() { // Programa escravo - gcd.c
2     int tid,x,y;
3 /*1*/     tid = pvm_parent();
4 /*2*/     pvm_recv(tid,-1);
5 /*2*/     pvm_upkint(&x,1,1);
6 /*2*/     pvm_upkint(&y,1,1);
7 /*3*/     while (x != y) {
8 /*4*/         if (x < y)
9 /*5*/             y = y-x;
10 /*6*/         else
11 /*6*/             x = x-y;
12 /*7*/
13 /*8*/     pvm_initsend(PvmDataDefault);
14 /*8*/     pvm_pkint(&x,1,1);
15 /*8*/     pvm_send(tid,2);
16 /*9*/     pvm_exit();
17 /*9*/}

```

---



---

### 9.3 Tipos de erros em programas concorrentes

Para a definição de critérios de teste no contexto de programas concorrentes, dois pontos importantes devem ser considerados: 1) os tipos de erros que devem ser evidenciados pelos critérios de teste; e 2) como representar o programa concorrente de modo a obter as informações necessárias para os critérios de teste. Esse último item é descrito na Seção 9.4.

Krawczyk e Wisniewski [227] apresentam dois tipos de erros relacionados a programas concorrentes. Dado um caminho  $p_x$ ,  $D(p_x)$  é o seu domínio de entrada, isto é,  $D(p_x)$  é formado pelas entradas que exercitam  $p_x$  e  $C(p_x)$  é a computação de  $p_x$ , isto é,  $C(p_x)$  é formada pelas saídas produzidas pela execução de  $p_x$ . Os dois tipos de erros definidos são:

1. erro de observabilidade (*observability error*): ocorre em um programa  $P$  se para algum caminho  $p_i \in P$  existe outro caminho  $p_j \in P$  tal que  $D(p_i) = D(p_j)$  e  $C(p_i) \neq C(p_j)$ .
2. erro de travamento (*locking error*): ocorre em um programa  $P$  se para algum caminho  $p \in P$ , formado pelo conjunto de nós  $p = q_0 q_1 \dots q_j$ , existe um nó  $q_i$  em  $p$  tal que o subcaminho  $p_s = q_0 q_1 \dots q_i$  contendo um domínio  $D(p_s)$ , existe pelo menos um elemento de  $D(p_s)$  em que todos os predicados avaliados a partir de  $q_i$  são falsos.

O erro de observabilidade é relacionado ao ambiente de teste. Ocorre quando o testador não pode controlar o comportamento do programa executando em paralelo. Para ilustrar esta situação, considere o Programa 9.1. Suponha que o comando `pvm-receive()` da linha 17 (receive bloqueante) tenha sido, erroneamente, trocado por um `pvm-nrecv()` (receive não bloqueante). Nesse caso, o comportamento do programa pode variar, dependendo do tempo de execução de cada um dos processos. Se o processo escravo  $S_1$  executar rapidamente a tempo de sincronizar com o comando `pvm-nrecv()` incorreto da linha 17, o erro não será observado e a saída obtida será avaliada como correta. Agora, considere que isso não ocorra e que o testador selecione  $T_1 = \langle x = 3, y = 6, z = 12 \rangle$ . Nesse caso, o erro ainda não é observado, pois, mesmo que o valor de  $y$  não seja alterado, esse valor será enviado para  $S_2$ , que processará os resultados e irá gerar uma saída correta. O testador deveria ser capaz de distinguir essas duas situações, utilizando mecanismos para verificar as seqüências de comandos executados. Por outro lado, se o testador selecionar o dado de teste  $T_2 = \langle x = 12, y = 6, z = 3 \rangle$ , o erro será observado, pois uma saída incorreta será produzida. Apesar disso, sem um mecanismo de apoio para verificar os comandos executados, o testador pode chegar à conclusão de que esse erro é de computação e não de observabilidade. Na Seção 9.6 será discutido como introduzir mecanismos para rastrear as seqüências de comunicação e sincronização durante a execução de programas concorrentes.

O erro de travamento irá ocorrer nesse exemplo quando um processo não se comunicar com o processo correto. Para ilustrar, considere que o mestre envia uma mensagem para  $S_1$  em vez de para  $S_2$  na linha 23. Como o processo  $S_1$  já se comunicou com o mestre e já finalizou sua execução, o processo mestre ficará travado na linha 24, esperando uma resposta de  $S_1$ .

Esses tipos de erros e outras variações são explorados pelos critérios de teste apresentados nas próximas seções.

## 9.4 Critérios estruturais em programas concorrentes

Foi observado no Capítulo 4 que critérios de teste estruturais utilizam um grafo para representar o código do programa e extrair as informações necessárias para aplicação desses critérios. Da mesma forma, para programas concorrentes um modelo é necessário para abstrair todas as características desses programas. Assim, existem algumas propostas de representações que buscam atingir essa finalidade. Basicamente, as propostas concentram-se em dois tipos de representação: modelos baseados em estados e modelos baseados em fluxo de controle e de dados. Esses modelos são apresentados a seguir.

### 9.4.1 Modelos para representação de programas concorrentes

Taylor et al. [388] definem um Grafo de Concorrência que representa a transição de estados do programa concorrente, apresentando as possíveis sincronizações. Entretanto, como esse método é baseado em estados possíveis no programa, está suscetível ao problema de explosão de estados.

Yang e Chung [81] definem dois grafos: 1) Grafo de Sincronização, que apresenta o comportamento em tempo de execução do programa concorrente, modelando as possíveis sincronizações; e 2) Grafo de Processo, que apresenta a visão estática do programa, modelando o fluxo de controle entre seus comandos. Uma execução do programa irá sensibilizar uma rota de sincronização (*c*-rota) no Grafo de Sincronização e um caminho (*c*-caminho) no Grafo de Processo, sendo que um *c*-caminho pode possuir mais de uma *c*-rota. Os autores apontam três resultados práticos de sua pesquisa: seleção de caminhos, geração de casos de teste e execução dos casos de teste.

Ao explorar os conceitos de Grafos de Programas, Yang et al. [451] apresentam uma extensão a esses grafos capaz de representar os aspectos de comunicação entre processos em um programa concorrente. Basicamente, o trabalho propõe a construção de um Grafo de Fluxo de Controle (GFC) para cada processo paralelo, da mesma forma que é feito para programas seqüenciais. A partir desses grafos, um Grafo de Fluxo de Controle Paralelo é construído, composto dos GFCs dos processos, adicionando arestas de sincronização. Essas arestas representam a criação dos processos paralelos e a comunicação entre esses processos. A contribuição principal desse trabalho está em apresentar que é possível, com as devidas adaptações, estender critérios de teste seqüenciais para programas paralelos. Os autores consideram somente programas paralelos que utilizam variáveis compartilhadas, implementados na linguagem Ada.

As idéias apresentadas por Yang et al. [451] foram estendidas por Vergilio et al. [409] para permitir a aplicação de critérios de teste para programas paralelos em ambientes de passagem de mensagem. Um modelo de fluxo de dados também foi proposto, adicionando aos grafos informações sobre definição e uso de variáveis, principalmente aquelas envolvidas na comunicação entre os processos. A diferença principal desse modelo de teste em relação ao modelo de Yang et al. [451] é que são considerados processos paralelos executando em diferentes espaços de memória. Com isso, adaptações foram necessárias para que se tornasse possível representar, dentre outras coisas, a comunicação entre os processos paralelos. Além disso, o modelo desenvolvido por Vergilio et al. [409] focaliza a aplicação dos testes para validar um único processo e também a aplicação paralela como um todo.

O modelo de teste proposto por Vergilio et al. [409] considera um número  $n$  fixo e conhecido de processos paralelos criado durante a inicialização da aplicação paralela. Esses processos podem executar o mesmo ou diferentes programas, mas cada processo executa seu código no próprio espaço de memória. A comunicação entre os processos acontece de duas maneiras. A primeira, chamada de comunicação ponto a ponto, utiliza comandos de envio e recebimento de mensagens em que é especificado o processo origem e o processo destino da mensagem. A segunda, chamada de comunicação coletiva entre processos, utiliza comandos que permitem que uma mensagem seja enviada para todos os processos paralelos da aplicação (ou para um grupo de processos). Nesse modelo, a comunicação coletiva acontece somente num único e predefinido domínio (contexto), que envolve todos os processos da aplicação.

O programa paralelo  $Prog$  é dado por um conjunto de processos  $Prog = p^0, p^1, \dots, p^{n-1}$ . Cada processo  $p$  possui seu GFC, construído utilizando-se os mesmos conceitos utilizados para programas seqüenciais. Um nó  $n$  pode ou não estar associado a uma função de comunicação (`send()` ou `receive()`). Considera-se que as funções de comunicação possuem os parâmetros da seguinte forma: `send(i, j, t)` (ou `receive(i, j, t)`), significando que o processo  $i$  envia (ou recebe) a mensagem  $t$  para o (ou do) processo  $j$ . Cada GFC possui um único nó de entrada e um único nó de saída pelos quais, respectivamente, se têm o início e o término da execução do processo  $p$ .

Um Grafo de Fluxo de Controle Paralelo (GFCP) é construído para  $Prog$ , sendo composto pelos  $GFC^p$  (para  $p = 0..n-1$ ) e pelas arestas de comunicação dos processos paralelos.  $N$  e  $E$  representam, respectivamente, o conjunto de nós e arestas de GFCP. Um nó  $i$  em um processo  $p$  é representado pela notação:  $n_i^p$ . Dois subconjuntos de  $N$  são definidos:  $N_s$  e  $N_r$ , compostos por nós que possuem, respectivamente, funções de envio e funções de recebimento de mensagens. Para cada  $n_i^p \in N_s$ , um conjunto  $R_i^p$  é associado, o qual contém os possíveis nós que recebem a mensagem enviada pelo nó  $n_i^p$ .

As arestas de GFCP podem ser de dois tipos:

- arestas intraprocessos ( $E_i$ ): são arestas internas a um processo  $p$ ;
- arestas interprocessos ( $E_s$ ): são arestas que representam a comunicação entre os processos, ou seja, são arestas que ocorrem entre os processos paralelos.

Um caminho  $\pi$  no GFC de um processo  $p$  é chamado intraprocesso se ele não contém nenhuma aresta interprocessos. É dado por uma seqüência de nós de  $\pi = (n_1, n_2, \dots, n_m)$ , tal que  $(n_i, n_{i+1}) \in E_i^p$ .

Um caminho  $\pi$  no GFCP que contém pelo menos uma aresta interprocessos é dito um caminho interprocessos. Um caminho interprocessos é completo se todos os caminhos intraprocessos que o compõem são também completos.

O símbolo  $\prec$  é utilizado para indicar a ordem de execução entre dois nós de diferentes processos. Portanto,  $n_j^{p_1} \prec n_k^{p_2}$  indica que  $n_j^{p_1}$  é executado antes que  $n_k^{p_2}$ . Um caminho completo  $\pi$  inclui (ou cobre) um nó ou uma aresta  $(n_j^p, n_k^p) \in E_i^p$  se ele inclui um subcaminho intraprocessos que inclui o nó ou aresta dada. Um caminho  $\pi$  inclui (ou cobre) uma aresta  $(n_j^{p_1}, n_k^{p_2}) \in E_s$  se caminhos  $\pi_1$  e  $\pi_2$  de  $\pi$ , executados respectivamente nos processos  $p_1$  e  $p_2$ , incluírem os nós  $n_j^{p_1}$  e  $n_k^{p_2}$  e se  $n_j^{p_1} \prec n_k^{p_2}$ , ou seja, deve-se garantir que o comando `send` do nó  $n_j^{p_1}$  sincronizou com o `receive` do nó  $n_k^{p_2}$ .

Uma variável  $x$  é definida quando um valor é armazenado na sua posição de memória correspondente. Geralmente, uma variável é definida quando ela se encontra no lado esquerdo da atribuição ou é passada por referência em alguma função, armazenando o valor de saída dessa função ou o valor de seu retorno. No contexto de programas escritos em ambiente de passagem de mensagem, um novo tipo de definição de variável é possível: a definição decorrente da execução de uma função de comunicação tal como um comando *receive*. O conjunto de variáveis definidas em um nó  $n_i^p$  é dado por  $\text{def}(n_i^p)$ .

O uso de uma variável  $x$  ocorre quando  $x$  é referenciado em uma expressão. No contexto de programas concorrentes, são caracterizados três tipos de usos de variáveis:

1. *uso computacional (c-uso)*: ocorre em uma computação, relacionado a um nó  $n_i^p$  do GFCP;
2. *uso predicativo (p-uso)*: ocorre em uma condição (predicado), associado a comandos de fluxo de controle, em arestas intraprocessos  $(n_i^p, n_j^p)$  do GFCP;
3. *uso comunicacional (s-uso)*: ocorre em comandos de sincronização (*send* e *receive*), em arestas interprocessos  $(n_i^{p1}, n_j^{p2}) \in E_s$ .

Da mesma forma que ocorre com os demais critérios de teste estruturais, um caminho  $\pi = (n_1, n_2, \dots, n_j, n_k)$  é livre de definição em relação a uma variável  $x$  definida no nó  $n_1$  e usada no nó  $n_k$  ou aresta  $(n_j, n_k)$ , se  $x \in \text{def}(n_1)$  e  $x \notin \text{def}(n_i)$ , para  $i = 2 \dots j$ .

A partir dessas informações, é possível caracterizar as associações que devem ser requeridas pelos critérios estruturais aplicados a programas concorrentes. Três tipos de associações podem ser derivados:

- associação c-uso – dada pela tripla  $(n_i^p, n_j^p, x)$  em que  $x \in \text{def}(n_i^p)$ , e  $n_j^p$  possui um c-uso de  $x$  e existe um caminho livre de definição com relação a  $x$  de  $n_i^p$  para  $n_j^p$ ;
- associação p-uso – dada pela tripla  $(n_i^p, (n_j^p, n_k^p), x)$  em que  $x \in \text{def}(n_i^p)$ , e  $(n_j^p, n_k^p)$  possui um p-uso de  $x$  e existe um caminho livre de definição com relação a  $x$  de  $n_i^p$  para  $(n_j^p, n_k^p)$ ;
- associação s-uso – dada pela tripla  $(n_i^{p1}, (n_j^{p1}, n_k^{p2}), x)$  em que  $x \in \text{def}(n_i^{p1})$ , e  $(n_j^{p1}, n_k^{p2})$  possui um s-uso de  $x$  e existe um caminho livre de definição com relação a  $x$  de  $n_i^{p1}$  para  $(n_j^{p1}, n_k^{p2})$ .

Note que as associações p-uso e c-uso são associações intraprocessos, ou seja, a definição e o uso de  $x$  ocorrem no mesmo processo  $p_1$  e são geralmente requeridas no teste de programas seqüenciais, realizando o teste de cada processo isoladamente. Uma associação s-uso pressupõe a existência de um segundo processo  $p_2$  e pode ser considerada uma associação interprocessos. Essas associações permitem verificar erros na comunicação entre processos, ou seja, no envio e no recebimento de mensagens. Considerando associações s-usos e a comunicação entre processos, outros tipos de associações interprocessos são caracterizadas:

- associação s-c-uso – dada pela quíntupla  $(n_i^{p1}, (n_j^{p1}, n_k^{p2}), n_l^{p2}, x^{p1}, x^{p2})$ , tal que existe uma associação s-uso  $(n_i^{p1}, (n_j^{p1}, n_k^{p2}), x^{p1})$  e uma associação c-uso  $(n_k^{p2}, n_l^{p2}, x^{p2})$ ;

- associação s-p-uso – dada pela quíntupla  $(n_i^{p_1}, (n_j^{p_1}, n_k^{p_2}), (n_l^{p_2}, n_m^{p_2}), x^{p_1}, x^{p_2})$ , tal que existe uma associação s-uso  $(n_i^{p_1}, (n_j^{p_1}, n_k^{p_2}), x^{p_1})$  e uma associação p-uso  $(n_k^{p_2}, (n_l^{p_2}, n_m^{p_2}), x^{p_2})$ .

Para ilustrar essas definições, considere o programa PVM (Programas 9.1 e 9.2). O processo  $p^m$  representa o processo mestre que gera três processos escravos ( $p^0, p^1$  e  $p^2$ ). Cada processo possui um GFC associado. O GFCP é apresentado na Figura 9.1. Observe que o grafo está simplificado; para facilitar sua visualização, algumas arestas interprocessos foram omitidas. Os comandos de sincronização são separados dos demais nós de comandos para possibilitar a geração das arestas interprocessos ( $E_s$ ). Assim, o nó  $2^m$  (nó 2 do processo  $p^m$ ) está associado aos comandos compreendidos entre as linhas 7 e 10, os quais são responsáveis por preparar o buffer e enviar as variáveis  $x$  e  $y$  para o processo  $p^0$ . Assim, é caracterizada a aresta interprocesso de  $2^m$  para  $2^0$ . A Tabela 9.1 contém os conjuntos obtidos a partir dessas definições, os quais são utilizados para derivação dos requisitos de teste. A Tabela 9.2 apresenta o conjunto de variáveis definidas em cada nó do grafo da Figura 9.1.

Tabela 9.1 – Informações obtidas pelo modelo de teste para o programa *gcd*

|  |
|--|
| $n = 4$  |
| $Prog = p^m, p^0, p^1, p^2$  |
| $N = \{1^m, 2^m, 3^m, 4^m, 5^m, 6^m, 7^m, 8^m, 9^m, 10^m, 1^0, 2^0, 3^0, 4^0, 5^0, 6^0, 7^0, 8^0, 9^0, 1^1, 2^1, 3^1, 4^1, 5^1, 6^1, 7^1, 8^1, 9^1, 1^2, 2^2, 3^2, 4^2, 5^2, 6^2, 7^2, 8^2, 9^2\}$                                 |
| $N_s = \{2^m, 3^m, 7^m, 8^0, 8^1, 8^2\}$ (nós com <i>pvm_send()</i> )  |
| $N_r = \{4^m, 5^m, 8^m, 2^0, 2^1, 2^2\}$ (nós com <i>pvm_recv()</i> )  |
| $R_2^m = \{2^0, 2^1, 2^2\}$  |
| $R_3^m = \{2^0, 2^1, 2^2\}$  |
| $R_7^m = \{2^0, 2^1, 2^2\}$  |
| $R_{80} = \{4^m, 5^m, 8^m\}$   |
| $R_{81} = \{4^m, 5^m, 8^m\}$   |
| $R_{82} = \{4^m, 5^m, 8^m\}$   |
| $E = E_i^p \cup E_s$   |
| $E_i^m = \{(1^m, 2^m), (2^m, 3^m), (3^m, 4^m), (4^m, 5^m), (5^m, 6^m), (6^m, 7^m), (7^m, 8^m), (8^m, 10^m), (6^m, 9^m), (9^m, 10^m)\}$   |
| $E_i^0 = \{(1^0, 2^0), (2^0, 3^0), (3^0, 4^0), (4^0, 5^0), (4^0, 6^0), (5^0, 7^0), (6^0, 7^0), (7^0, 3^0), (3^0, 8^0), (8^0, 9^0)\}$   |
| $E_i^1 = \{(1^1, 2^1), (2^1, 3^1), (3^1, 4^1), (4^1, 5^1), (4^1, 6^1), (5^1, 7^1), (6^1, 7^1), (7^1, 3^1), (3^1, 8^1), (8^1, 9^1)\}$   |
| $E_i^2 = \{(1^2, 2^2), (2^2, 3^2), (3^2, 4^2), (4^2, 5^2), (4^2, 6^2), (5^2, 7^2), (6^2, 7^2), (7^2, 3^2), (3^2, 8^2), (8^2, 9^2)\}$   |
| $E_s = \{(2^m, 2^0), (2^m, 2^1), (2^m, 2^2), (3^m, 2^0), (3^m, 2^1), (3^m, 2^2), (7^m, 2^0), (7^m, 2^1), (7^m, 2^2), (8^0, 4^m), (8^0, 5^m), (8^0, 8^m), (8^1, 4^m), (8^1, 5^m), (8^1, 8^m), (8^2, 4^m), (8^2, 5^m), (8^2, 8^m)\}$ |

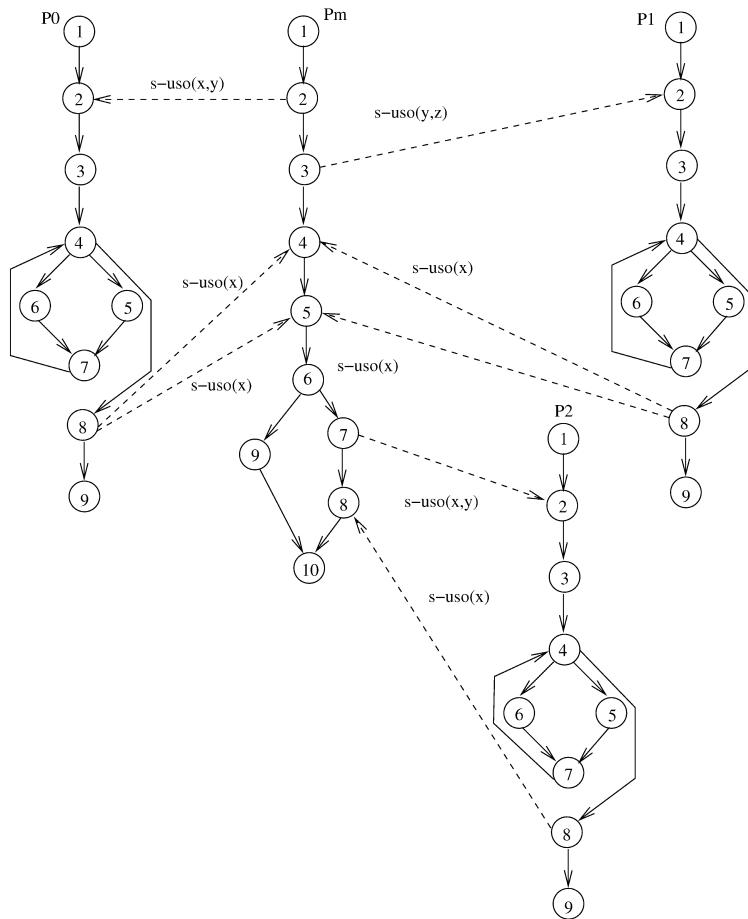


Figura 9.1 – GFCP do Programa gcd.

Tabela 9.2 – Variáveis definidas no GFCP do Programa gcd

|  |  |
|--|--|
| $\text{def}(1^m) = \{x, y, z, S\}$<br>$\text{def}(4^m) = \{x\}$<br>$\text{def}(5^m) = \{y\}$<br>$\text{def}(8^m) = \{z\}$<br>$\text{def}(9^m) = \{z\}$ | $\text{def}(1^0) = \{\text{tid}\}$<br>$\text{def}(2^0) = \{x, y\}$<br>$\text{def}(5^0) = \{y\}$<br>$\text{def}(6^0) = \{x\}$ |
| $\text{def}(1^1) = \{\text{tid}\}$<br>$\text{def}(2^1) = \{x, y\}$<br>$\text{def}(5^1) = \{y\}$<br>$\text{def}(6^1) = \{x\}$                           | $\text{def}(1^2) = \{\text{tid}\}$<br>$\text{def}(2^2) = \{x, y\}$<br>$\text{def}(5^2) = \{y\}$<br>$\text{def}(6^2) = \{x\}$ |

### 9.4.2 Critérios de teste

Taylor et al. [388] introduzem alguns critérios e ilustram suas idéias em programas ADA. Entre esses critérios destacam-se: Todos-Caminhos do Grafo de Concorrência, Todas-Arestas entre os estados de Sincronização, Todos-Estados, Todos-Possíveis-Rendezvous. Como dito anteriormente, alguns desses critérios podem ser inaplicáveis, pois o número de estados pode crescer rapidamente.

Yang et al. [451] estendem o critério de fluxo de dados Todos-Du-Caminhos para programas paralelos. O Grafo de Fluxo de Programa Paralelo (PPFG) é percorrido para obtenção dos du-caminhos. Todos os du-caminhos que possuem definição e uso de variáveis relacionadas ao paralelismo das tarefas são requisitos de teste a serem executados, sendo as tarefas atividades de um programa paralelo que se comunicam entre si mas são executadas independentemente (concorrentemente).

Com base nas definições dadas na seção anterior, Vergilio et al. [409] definem duas famílias de critérios de teste estrutural para programas concorrentes em ambientes de passagem de mensagens: família de critérios baseados em fluxo de controle e no fluxo de comunicação e família de critérios baseados em fluxo de dados e em passagem de mensagens.

#### Critérios baseados em fluxo de controle e no fluxo de comunicação

Cada GFC pode ser testado individualmente utilizando-se os critérios Todos-Nós e Todas-Arestas. Entretanto, o objetivo é testar as comunicações presentes no GFCP para um dado *Prog*. Desse modo, os seguintes critérios são definidos:

- Todos-Nós-s: requer que todos os nós do conjunto  $N_s$  sejam exercitados pelo menos uma vez pelo conjunto de casos de teste.
- Todos-Nós-r: requer que todos os nós do conjunto  $N_r$  sejam exercitados pelo menos uma vez pelo conjunto de casos de teste.
- Todos-Nós: requer que todos os nós do conjunto  $N$  sejam exercitados pelo menos uma vez pelo conjunto de casos de teste.
- Todos-Arestas-s: requer que todas as arestas do conjunto  $E_s$  sejam exercitadas pelo menos uma vez pelo conjunto de casos de teste.
- Todas-Arestas: requer que todas as arestas do conjunto  $E$  sejam exercitadas pelo menos uma vez pelo conjunto de casos de teste.

Ainda podem-se definir os critérios Todos os Caminhos do  $GFC^p$  (para  $p = 0, \dots, n - 1$ ) e Todos os Caminhos do GFCP, equivalente a executar todas as combinações possíveis dos caminhos nos  $GFC^p$ , mas esses critérios são em geral impraticáveis.

#### Critérios baseados em fluxo de dados e em passagem de mensagens

- Todas-Defs: requer que para cada nó  $n_i^p$  e cada  $x$  em  $def(n_i^p)$  uma associação definição-uso (c-uso ou p-uso) com relação a  $x$  seja exercitada pelo conjunto de casos de teste.

- Todas-Defs/s: requer que para cada nó  $n_i^p$  e cada  $x$  em  $def(n_i^p)$  uma associação s-c-uso ou s-p-uso com relação a  $x$  seja exercitada pelo conjunto de casos de teste. Caso não exista associação s-c-uso ou s-p-uso com relação a  $x$ , é requerido qualquer outro tipo de associação interprocessos em relação a  $x$ .
- Todos-c-Usos: requer que todas as associações c-usos sejam exercitadas pelo conjunto de casos de teste.
- Todos-p-Usos: requer que todas as associações p-usos sejam exercitadas pelo conjunto de casos de teste.
- Todos-s-Usos: requer que todas as associações s-usos sejam exercitadas pelo conjunto de casos de teste.
- Todos-s-c-Usos: requer que todas as associações s-c-usos sejam exercitadas pelo conjunto de casos de teste.
- Todos-s-p-Usos: requer que todas as associações s-p-usos sejam exercitadas pelo conjunto de casos de teste.

A Tabela 9.3 apresenta exemplos de elementos requeridos por esses critérios, para o programa da Figura 9.1. Da mesma forma que ocorre para programas seqüenciais, a não-executabilidade é inherente a esses critérios de teste. Para ilustrar, considere o critério Todos-s-Usos. A associação  $(1^m, (2^m, 2^1), x, y)$  é não-executável, pois o comando send em  $2^m$  envia a mensagem explicitamente para o processo  $p^0$  e, portanto, não sincroniza com o processo  $p^1$ .

## 9.5 Teste de mutação em programas concorrentes

No Capítulo 5 foram abordados aspectos da aplicação do teste de mutação para programas seqüenciais. Esses aspectos são válidos no contexto de programas concorrentes, mas novas questões precisam ser avaliadas.

O problema de aplicar teste de mutação para programas concorrentes é a dificuldade em analisar o comportamento dos mutantes. Uma vez que o programa pode apresentar diferentes resultados corretos, o fato de o mutante possuir um comportamento diferente do programa original deve ser analisado de maneira diferente. A discrepância pode ocorrer devido ao não-determinismo e não por causa da mutação realizada. Quando um mutante  $M$  é executado com um caso de teste  $t$ , ele deve ser morto se seu comportamento for incorreto, de acordo com sua especificação. No caso de programas seqüenciais, isso é semelhante a dizer que o resultado da execução de  $M$  com  $t$  difere do resultado da execução do programa original  $P$  com  $t$ , uma vez que  $P$  já deve ter sido executado com  $t$  e apresentado ou não o resultado esperado. Para programas concorrentes, é necessário comparar o conjunto de todos os resultados possíveis de  $M$  quando executado com  $t$  em relação ao conjunto de todos os resultados possíveis de  $P$  com  $t$ . Se  $M$  apresentar um comportamento diferente, ele foi distinguido de  $P$ . Entretanto, em geral, não é possível conhecer todo o comportamento possível de um programa concorrente para determinada entrada.

Offutt et al. [314] propõe uma técnica que calcula um conjunto aproximado  $\Omega$ , que representa um subconjunto de todos os resultados possíveis de  $P(t)$ . Esse conjunto é obtido

Tabela 9.3 – Elementos requeridos pelos critérios de teste em relação ao programa *gcd*

| Critério        | Elementos requeridos   |
|-----------------|--|
| Todos-Nós-s     | $2^m, 3^m, 7^m, 8^0, 8^1, 8^2$   |
| Todos-Nós-r     | $4^m, 5^m, 8^m, 2^0, 2^1, 2^2$   |
| Todos-Nós       | $1^m, 2^m, 3^m, 4^m, 5^m, 6^m, 7^m, 8^m, 9^m, \dots, 1^0, 2^0, 3^0, \dots, 1^1, 2^1, 3^1, \dots$   |
| Todas-Arestas-s | $(2^m, 2^0), (2^m, 2^1), (2^m, 2^2), (3^m, 2^0), (3^m, 2^1), (3^m, 2^2), (7^m, 2^2), (8^0, 4^m), (8^0, 5^m), (8^0, 8^m), (8^1, 4^m), (8^1, 5^m), (8^1, 8^m), (8^2, 4^m), (8^2, 5^m), (8^2, 8^m)$   |
| Todas-Arestas   | $(1^m, 2^m), (2^m, 3^m), (3^m, 4^m), \dots, (1^0, 2^0), (2^0, 3^0), \dots, (1^1, 2^1), (2^1, 3^1), \dots, (2^m, 2^0), (2^m, 2^1) \dots$  |
| Todas-Defs      | $(8^m, 10^m, z), (2^0, 5^0, x), (2^0, 6^0, x), (2^0, (3^0, 4^0), x), (2^0, 6^0, y) \dots$  |
| Todas-Defs/s    | $(1^m, (2^m, 2^0), 5^0, x, x), (1^m, (2^m, 2^0), 6^0, y, y), (1^m, (2^m, 2^0), 4^0, 5^0, y, y), (1^m, (3^m, 2^0), 5^0, z, y), \dots$   |
| Todos-c-Usos    | $(1^m, 10^m, z), (8^m, 10^m, z), (2^0, 8^0, x) \dots$  |
| Todos-p-Usos    | $(4^m, (6^m, 7^m), x), (4^m, (6^m, 9^m), x), (5^m, (6^m, 7^m), y), (5^m, (6^m, 9^m), y), (2^0, (3^0, 4^0), x), (2^0, (3^0, 8^0), y) \dots$   |
| Todos-s-Usos    | $(1^m, (2^m, 2^0), x, y), (1^m, (2^m, 2^1), x, y), (1^m, (3^m, 2^0), y, z), (4^m, (7^m, 2^2), x), (5^m, (7^m, 2^0), y), (5^m, (7^m, 2^1), y), \dots$   |
| Todos-s-c-Usos  | $(1^m, (2^m, 2^0), 5^0, x, x), (1^m, (2^m, 2^0), 6^0, x, x), (1^m, (2^m, 2^0), 5^0, y, y), (1^m, (2^m, 2^0), 6^0, y, y), (1^m, (2^m, 2^1), 6^1, x, x), (1^m, (3^m, 2^1), 6^1, x, x), (2^0, (8^0, 8^m), 10^m, x, z), \dots$   |
| Todos-s-p-Usos  | $(1^m, (2^m, 2^0), (3^0, 4^0), x, x), (1^m, (2^m, 2^0), (3^0, 8^0), x, x), (1^m, (2^m, 2^0), 4^0, 5^0, x, x), (1^m, (2^m, 2^0), 4^0, 5^0, y, y), (1^m, (3^m, 2^0), (3^0, 4^0), z, y), (5^m, (7^m, 2^0), (3^0, 4^0), y, x), (2^0, (8^0, 4^m), (6^m, 7^m), x, y), \dots$ |

executando  $P(t)$  várias vezes. O mutante  $M$  é distinguido se produzir no mínimo um resultado  $M(t) \notin \Omega$ .

Silva-Barradas propõe a aplicação do critério Análise de Mutantes para programas concorrentes escritos em Ada [359]. O autor define um conjunto de operadores de mutação para tratar os aspectos de concorrência de programas em Ada. Para isso, foram identificados todos os comandos (ou construtores) da linguagem que se relacionam à concorrência (por exemplo, os comandos *accept* e *entry*) e definidos operadores de mutação para tais comandos. Para tratar o não-determinismo, o autor propõe um mecanismo chamado de análise de mutantes comportamental. Nesse mecanismo, para cada execução do programa são consideradas a saída obtida  $t$  e a seqüência de comandos executados  $s$ , de forma que o comportamento do programa possa ser repetido a partir do valor de entrada e da seqüência percorrida. Assim, é esperado que duas execuções do programa  $P$  com a mesma entrada de teste e seqüência de execução  $s$  irá gerar a mesma saída  $t$  (notação  $P(t, s)$ ). A técnica considera que um mutante é distinguido se uma das condições ocorre: 1)  $M(t, s) \neq P(t, s)$ ; ou 2)  $M$  não pode reproduzir a seqüência  $s$ .

Delamaro et al. [111] definem um conjunto de operadores de mutação para testar aspectos de concorrência e sincronização da linguagem Java. Foram identificadas as principais estruturas relacionadas à concorrência e foram definidos operadores de mutação para execução dessas estruturas. Os autores descrevem os operadores de mutação definidos e argumentam

que eles cobrem a maioria dos aspectos relacionados à concorrência e sincronização e são também de baixo custo (geram um número pequeno de mutantes). Para tratar o problema de não-determinismo os autores consideraram a proposta de Silva-Barradas [359].

Giacometti et al. [151] apresentam uma proposta inicial para a aplicação do teste de mutação para programas em ambientes de passagem de mensagens, considerando o ambiente PVM. O conjunto de operadores de mutação definido procura modelar erros no fluxo de dados entre os processos paralelos; erros no empacotamento de mensagens; erros na sincronização e no paralelismo dos processos. Os autores indicam a necessidade de tratar o não-determinismo no contexto de PVM, a exemplo do trabalho de Silva-Barradas [359].

## 9.6 Execução não determinística

Na maioria dos modelos de programação paralela, a execução determinística não é garantida. Assim, se um programa paralelo é executado diversas vezes com a mesma entrada, a seqüência de comandos executados pode variar. Considerando o programa exemplo (Programas 9.1 e 9.2), o comando `recv()` – linha 15 do processo mestre – pode sincronizar com a mensagem recebida tanto do processo escravo  $p1 (S[0])$  como do processo escravo  $p2 (S[1])$ . O mesmo fato vale para o comando `recv()` (linha 17 do processo mestre). Nesse caso, isso não caracteriza uma situação de erro, mas é necessário garantir que o programa funciona corretamente para todas as sincronizações possíveis. Além disso, o não-determinismo pode ser uma situação de erro do programa e precisa ser identificado e corrigido.

Carver e Tai [59] desenvolveram uma abordagem conhecida como execução controlada para tratar a execução não determinística durante o teste e depuração de programas concorrentes. A partir dos construtores de sincronização da linguagem utilizada (por exemplo, monitores e semáforos), é definida uma seqüência de eventos de sincronização. Durante a execução são coletadas as seqüências de sincronização possíveis no programa e, em seguida, um mecanismo de controle é utilizado de modo a forçar a execução das sincronizações. Assim, o teste de execução controlada é realizado da seguinte forma:

1. selecionar um conjunto de teste da forma  $(X, S)$ , em que  $X$  representa uma entrada para o programa  $P$  e  $S$ , uma seqüência de sincronização de  $P$ ;
2. para cada teste  $(X, S)$  selecionado:
  - (a) determinar se  $S$  é ou não executável para  $P$  com  $X$ , tentando sensibilizar a execução de  $P$  com  $X$  conforme  $S$ ;
  - (b) se a seqüência  $S$  é possível, examinar o resultado dessa execução.

Essa abordagem mostra-se bastante eficiente para a execução determinística de programas concorrentes em Ada. Adaptações são necessárias para tratar programas concorrentes executando em diferentes processadores e compostos por diferentes *threads*.

Damodaran-Kamal e Francioni [103] propõem um mecanismo não intrusivo de execução controlada para programas paralelos em ambientes de passagem de mensagens. Os mecanismos de controle coletam informações sobre envio e recebimento de mensagens e são implementados em um programa que é executado juntamente com cada processo paralelo. Os autores citam que, por meio desses mecanismos, é possível explorar as possíveis disputas de

mensagens recebidas (quando um *receive* de um processo pode receber mais de uma mensagem e depende da ordem de chegada para que tais mensagens sejam recebidas) e detectar erros de mensagens enviadas/recebidas.

Yang [452] descreve técnicas que podem ser utilizadas para testar programas concorrentes por meio de execução não determinística. O objetivo dessas técnicas é mudar o tempo de execução de eventos de sincronização (por exemplo, eventos de envio e de recebimento de mensagens), sem introduzir mudanças no ambiente em que o programa está sendo executado. Esse tipo de teste é conhecido como teste temporal. Se um programa não tem erros de sincronização, mudar o tempo desses eventos não deveria causar *deadlocks* ou outro comportamento incorreto no programa. Duas técnicas podem ser empregadas para esse fim: múltiplas execuções, em que o programa concorrente é executado várias vezes com a mesma entrada de teste e inserção de atrasos, em que são inseridos comandos de atraso (*delays*) no programa concorrente de modo a atrasar alguns segmentos do programa. O autor destaca estudos que indicam que a inserção de atrasos pode ser um mecanismo eficiente para identificar erros em programas concorrentes.

Essas técnicas apresentam vantagens e desvantagens. Por exemplo, a execução não determinística, inserindo atrasos ou executando várias vezes o programa, tem a vantagem de ser simples de implementar. Entretanto, um dos problemas é que a inserção de atrasos pode aumentar significativamente o tempo de execução do programa concorrente, que ficará atrasado até a sincronização desejada ser atingida. Sem dúvida, o maior problema é a quantidade de informações que precisam ser testadas, pois é preciso inserir atrasos de modo que todas as combinações possíveis de sincronizações sejam testadas. Por outro lado, a execução controlada pode ser mais barata se comparada com teste temporal. Porém, os mecanismos de controle inseridos para forçar a execução de certas sincronizações podem modificar sensivelmente o comportamento do programa. O efeito disso ainda precisa ser estudado.

De uma maneira geral, o testador precisa ter em mente quais são seus objetivos quando estiver tratando a execução não determinística de programas concorrentes. De modo simplista, dois objetivos principais podem ser almejados: 1) reproduzir um teste já realizado, possivelmente buscando verificar se os erros corrigidos não produziram outros erros no programa; e 2) executar todas as seqüências de sincronização, buscando um mecanismo que permita combinar todas as possibilidades de sincronização e garantir que todas funcionem adequadamente, sem se esquecer de gerenciar a explosão de combinações possíveis.

## 9.7 Ferramentas

A maioria das ferramentas existentes auxilia a análise, a visualização, o monitoramento e a depuração de um programa concorrente. Exemplos dessas ferramentas são: TDCAda [383] que apóia a depuração de programas na linguagem ADA, utilizando execução determinística; ConAn (*ConcurrencyAnalyser*) [249], que gera *drivers* para o teste de unidade de classes de programas concorrentes Java; e Xab [30], MDB [103] e Paragraph [174] para programas escritos em ambiente de passagem de mensagens. Xab e Paragraph objetivam o monitoramento de programas PVM e MPI, respectivamente. MDB permite uma execução controlada e a depuração de programas PVM. Outras ferramentas que analisam o desempenho de programas MPI são avaliadas em por Moore et al. [287].

A ferramenta Della Pasta (*Delaware Parallel Software Testing Aid*) [451] foi uma das primeiras ferramentas desenvolvidas para a análise de caminhos em programas concorrentes. Ela permite o teste de programas concorrentes com memória compartilhada. Possui um analisador estático, que recebe como entrada o nome do arquivo a ser testado e gera possíveis caminhos para cobrir associações com relação a variáveis envolvidas na sincronização entre tarefas.

Em se tratando de suporte a critérios de teste em ambientes de passagem de mensagens, destacam-se três ferramentas: STEPS [251], Astral [351] e ValiPar [376].

STEPS e Astral são ferramentas que apoiam a visualização e a depuração de programas PVM. Ambas as ferramentas geram caminhos para cobrir elementos requeridos por critérios estruturais baseados em fluxo de controle.

ValiPar [376] apóia a aplicação dos critérios apresentados na Seção 9.4.2. Essa ferramenta permite que programas em diferentes ambientes de passagem de mensagens sejam testados. Atualmente, a ferramenta encontra-se configurada para PVM e MPI. A ValiPar é composta por quatro módulos principais (Figura 9.2), discutidos a seguir.

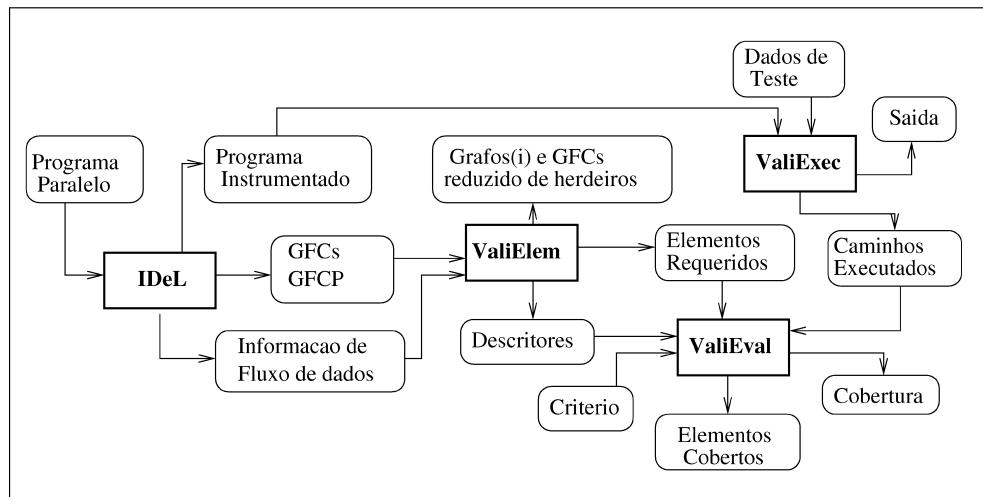


Figura 9.2 – Arquitetura da ferramenta de teste ValiPar.

- O módulo **IDeL** é responsável por extrair informações sobre o fluxo de controle, de dados e de comunicação dos programas paralelos. Ele utiliza a linguagem *ITDeL* (*Instrumentation Description Language*) desenvolvida por Simão et al. [362]. *ITDeL* é uma metalinguagem que apóia a instrumentação de programas. *ITDeL* utiliza a gramática da linguagem para extrair as informações necessárias para geração do grafos e do programa instrumentado. A instrumentação é obtida pela adição de comandos especiais que não mudam a funcionalidade original do programa, mas registram algumas informações em um arquivo de rastro (ou trace). Esse arquivo de rastro é gerado durante a execução do programa instrumentado. Devido à *ITDeL* ser uma metalinguagem, ela pode ser instanciada para diferentes programas. No contexto do módulo IDeL, a versão

da  $\mathcal{IDeL}$  para programas em C foi usada, a qual foi estendida para tratar comandos específicos do PVM e MPI, relacionados à comunicação e sincronização de processos. Se forem utilizados os GFCs e o GFCP gerados pelo módulo IDel, é possível obter informações sobre o fluxo de controle, fluxo de dados e fluxo de comunicação dos processos paralelos. Esse módulo é o único dependente da linguagem utilizada.

- O **módulo ValiElem** gera os elementos requeridos, tendo como entrada as informações produzidas pelo módulo IDel. Similarmente à ferramenta POKE-TOOL [61], apresentada no Capítulo 4, ValiElem utiliza dois tipos de grafos: um Grafo Reduzido de Herdeiros e vários grafos( $i$ ). O primeiro grafo contém somente arestas primitivas, ou seja, arestas que garantem a execução das demais e, por questões de minimização, são efetivamente requeridas. O segundo tipo, o grafo( $i$ ), é um grafo construído para todo nó  $i$  que contém uma definição de variável. Um determinado nó  $k$  pertence ao grafo( $i$ ) se existe pelo menos um caminho do nó  $i$  para  $k$  que não redefine a variável  $x$ , definida em  $i$ . Esse grafo é utilizado para estabelecer as associações requeridas. ValiElem gera, para cada elemento requerido, um descritor representado por meio de uma expressão regular, que descreve os possíveis caminhos que cobrem o elemento requerido.
- O **módulo ValiExec** executa o programa instrumentado mediante o comando e entradas fornecidas pelo usuário. Para cada entrada o programa instrumentado produzirá os caminhos percorridos em cada processo, bem como a seqüência de sincronização realizada. O módulo ainda armazena as entradas fornecidas, sejam elas pela linha de comando ou via teclado, e as saídas produzidas a serem verificadas com o resultado esperado pelo testador.
- O **módulo ValiEval** calcula a cobertura do conjunto de dados de teste fornecido, utilizando os caminhos e seqüências produzidos pelo módulo ValiExec. ValiEval utiliza os descritores produzidos pelos critérios de teste. Se um dado caminho é reconhecido pelo autômato correspondente a um elemento requerido, significa que esse elemento foi coberto pelo conjunto de teste.

A Tabela 9.4 sintetiza as principais características das ferramentas existentes para apoiar a atividade de teste de programas concorrentes.

## 9.8 Considerações finais

Neste capítulo foram apresentadas as principais contribuições para a definição de teste de programas concorrentes. Em seu trabalho, Yang [452] apresenta os principais desafios impostos para testar programas concorrentes:

- desenvolver técnicas de análise estática para analisar programas concorrentes;
- detectar situações não desejadas, tais como: erros de sincronização, comunicação, de fluxo de dados e de *deadlock*;
- reproduzir uma execução com a mesma entrada de teste e forçar a execução de um caminho na presença de não-determinismo;
- gerar uma representação do programa concorrente que capture as informações pertinentes à atividade de teste;

Tabela 9.4 – Ferramentas de teste/depuração para programas concorrentes

| Ferramenta                    | Fluxo de Dados | Fluxo de Controle | Execução Controlada | Depuração |
|-------------------------------|----------------|-------------------|---------------------|-----------|
| TDC Ada <sup>a</sup>          |                |                   | ✓                   |           |
| ConAn <sup>b</sup>            |                |                   | ✓                   |           |
| Della Pasta <sup>c</sup>      | ✓              |                   | ✓                   | ✓         |
| Xab <sup>d</sup>              |                |                   |                     | ✓         |
| Visit <sup>d</sup>            |                |                   |                     | ✓         |
| MDB <sup>d</sup>              |                |                   | ✓                   | ✓         |
| STEPS <sup>d</sup>            |                | ✓                 | ✓                   | ✓         |
| Astral <sup>d</sup>           |                | ✓                 |                     | ✓         |
| XMPI <sup>e</sup>             |                |                   |                     | ✓         |
| Umpire <sup>e</sup>           |                |                   |                     | ✓         |
| <b>ValiPar<sup>d, e</sup></b> | ✓              | ✓                 | ✓                   | ✓         |

<sup>a</sup>Ada, <sup>b</sup>Java, <sup>c</sup>shmem, <sup>d</sup>PVM, <sup>e</sup>MPI.

- investigar a aplicação de critérios de teste seqüenciais para programas concorrentes; e
- reprojetar critérios de fluxo de dados para programas concorrentes, considerando troca de mensagens e variáveis compartilhadas.

Em linhas gerais, as pesquisas nessa área já apresentam algumas soluções para esses desafios. É claro que algumas soluções são mais dispendiosas, como a execução não determinística, mas os resultados obtidos são muito promissores.

Os critérios de teste apresentados indicam que é factível considerar os critérios definidos para programas seqüenciais a fim de testar programas concorrentes. Ainda são necessários estudos para avaliar a complexidade e os aspectos complementares dos critérios propostos.

Com relação às ferramentas de apoio, observa-se que o esforço inicial concentrou-se em dispor ferramentas de apoio ao monitoramento e à depuração de programas concorrentes. Isso é natural, pois há um esforço muito maior para localizar erros em programas concorrentes, e as ferramentas de suporte à depuração de programas seqüenciais não são adequadas. Algumas ferramentas de apoio a critérios de teste foram apresentadas, indicando um esforço na direção de fornecer suporte ao teste de programas concorrentes.

# Capítulo 10

## Estudos Teóricos e Experimentais

*Simone do Rocio Senger de Souza (ICMC/USP)*

*Sandra C. Pinto Ferraz Fabbri (UFSCar)*

*Ellen Francine Barbosa (ICMC/USP)*

*Marcos Lordello Chaim (EACH/USP)*

*Auri Marcelo Rizzo Vincenzi (UNISANTOS)*

*Márcio Eduardo Delamaro (UNIVEM)*

*Mario Jino (DCA/FEEC/UNICAMP)*

*José Carlos Maldonado (ICMC/USP)*

### 10.1 Introdução

Devido à diversidade de critérios de teste existentes, decidir qual deles deva ser utilizado ou como utilizá-los de maneira complementar para obter melhores resultados com baixo custo não é tarefa fácil. A realização de estudos teóricos e experimentais permite comparar os diversos critérios de teste existentes, procurando fornecer uma estratégia viável para realização dos testes. A estratégia de teste que se deseja alcançar deve ser eficaz para revelar erros, ao mesmo tempo que apresenta baixo custo de aplicação.

Na abordagem teórica procuram-se estabelecer propriedades e características dos critérios de teste, como, por exemplo, a eficácia de uma estratégia de teste ou uma relação de inclusão entre os critérios. Na abordagem experimental são coletados dados e estatísticas que registram, por exemplo, a frequência na qual diferentes estratégias de teste revelam a presença de erros em determinada coleção de programas, fornecendo diretrizes para a escolha entre os diversos critérios disponíveis [189].

Do ponto de vista de estudos teóricos, os critérios de teste são avaliados segundo dois aspectos: a *relação de inclusão* e a *complexidade dos critérios*. A relação de inclusão estabelece uma ordem parcial entre os critérios, caracterizando uma hierarquia entre eles. Nessa hierarquia, um critério de teste  $C_1$  inclui um critério  $C_2$  se, para qualquer programa  $P$  e qualquer conjunto de casos de teste  $T_1$   $C_1$ -adequado (ou seja,  $T_1$  cobre todos os elementos requeridos executáveis de  $C_1$ ),  $T_1$  for também  $C_2$ -adequado, e para algum conjunto de casos de teste  $T_2$   $C_2$ -adequado,  $T_2$  não for  $C_1$ -adequado. A complexidade de um critério  $C_1$  é definida como o número máximo de casos de testes ou de elementos requeridos, no pior caso [337, 302, 429].

Do ponto de vista de estudos experimentais, os seguintes fatores são utilizados para avaliar os critérios de teste: *custo*, *eficácia* e *dificuldade de satisfação (strength)*. Entende-se por *custo* o esforço necessário para que o critério seja usado, o qual pode ser medido pelo número de casos de teste necessários para satisfazer o critério ou por outras métricas dependentes do critério, tais como: o tempo necessário para executar todos os mutantes gerados, ou o tempo gasto para identificar mutantes equivalentes, caminhos e associações não executáveis, construir manualmente os casos de teste e aprender a utilizar as ferramentas de teste. *Eficácia* refere-se à capacidade que um critério possui em detectar um maior número de erros em relação a outro. *Dificuldade de satisfação* refere-se à probabilidade de satisfazer-se um critério tendo sido satisfeito outro critério [435].

Este capítulo sintetiza os principais estudos teóricos e experimentais, relacionados aos critérios de teste vistos nos Capítulos 2, 4 e 5. Na Seção 10.2 são descritos os resultados de estudos teóricos relacionados aos critérios de teste estruturais e ao teste de mutação. Na Subseção 10.3.1 são apresentados os resultados de estudos experimentais relacionados aos critérios de teste estruturais e ao teste de mutação. Na Subseção 10.3.2 são apresentados os resultados de estudos experimentais relacionados ao teste funcional. Finalmente, na Seção 10.4 são apresentadas as considerações finais deste capítulo.

## 10.2 Estudos teóricos de critérios de teste

A complexidade e a relação de inclusão, utilizadas nos estudos teóricos, refletem, em geral, as propriedades básicas que devem ser consideradas no processo de definição de um critério de teste. Ou seja, o critério  $C$  deve:

1. incluir o critério Todas-Arestas. Um conjunto de casos de teste que exerçite os elementos requeridos pelo critério  $C$  deve exercitar todos os arcos do programa;
2. requerer, do ponto de vista de fluxo de dados, ao menos um uso de todo resultado computacional. Isso equivale ao critério  $C$  incluir o critério Todas-Defs;
3. requerer um conjunto de casos de teste finito.

Esses fatores influenciam também na escolha entre os diversos critérios de teste existentes.

Segundo Frankl e Weyuker [143], uma das propriedades que deve ser satisfeita por um bom critério de teste é a *aplicabilidade*: diz-se que um critério  $C$  satisfaz essa propriedade se para todo programa  $P$  existe um conjunto de casos de teste  $T$  que seja  $C$ -adequado para  $P$ , ou seja, o conjunto de caminhos executados por  $T$  inclui cada elemento exigido pelo critério  $C$ . Os critérios de fluxo de dados não satisfazem essa propriedade devido à existência de caminhos não-executáveis. Desse modo, é indecidível saber se existe um conjunto de casos de teste  $T$  que exerce todos os elementos requeridos por um dado critério de teste estrutural.

Rapps e Weyuker [337] apresentam a relação de inclusão entre os critérios de fluxo de dados (Figura 10.1). Pode-se observar, por exemplo, que o critério Todas-Arestas inclui o critério Todos-Nós, ou seja, se um conjunto de casos de teste satisfaz o critério Todas-Arestas, então esse conjunto também satisfaz o critério Todos-Nós. Quando não é possível estabelecer

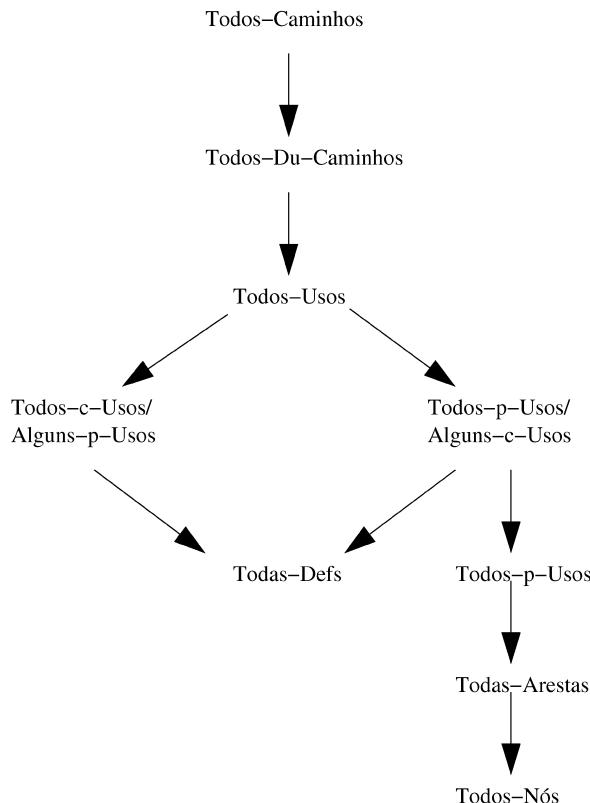


Figura 10.1 – Relação de inclusão dos critérios de fluxo de dados.

essa ordem de inclusão para dois critérios, diz-se que esses critérios são incomparáveis, como é o caso dos critérios Todas-Defs e Todos-p-Usos.

Maldonado [261] estendeu essa hierarquia inserindo os critérios Potenciais-Usos (Figura 10.2). Os estudos teóricos revelam que os critérios Potenciais-Usos satisfazem as propriedades básicas exigidas para um bom critério de teste (listadas anteriormente), mostrando que, mesmo na presença de caminhos não executáveis, eles estabelecem uma hierarquia de critérios entre os critérios Todas-Arestas e Todos-Caminhos. Além disso, esses critérios incluem o critério Todas-Defs e requerem um número finito de casos de teste. Outra característica é que nenhum outro critério baseado em fluxo de dados inclui os critérios Potenciais-Usos.

Segundo Maldonado [261], o primeiro passo na determinação da complexidade de um critério consiste em identificar qual estrutura de fluxo de controle, considerando um fluxo de dados qualquer, maximiza o número de elementos requeridos pelo critério. Desse modo, para determinar a complexidade dos critérios Potenciais-Usos, Maldonado [261] identificou o grafo de controle ilustrado na Figura 10.3 como sendo o que maximiza o número de elementos requeridos pelos critérios Potenciais-Usos, ou seja, potenciais du-caminhos e potenciais associações. Com base nesse grafo, Maldonado [261] mostra que  $((11/2)t + 9)2^t - 10t - 9$

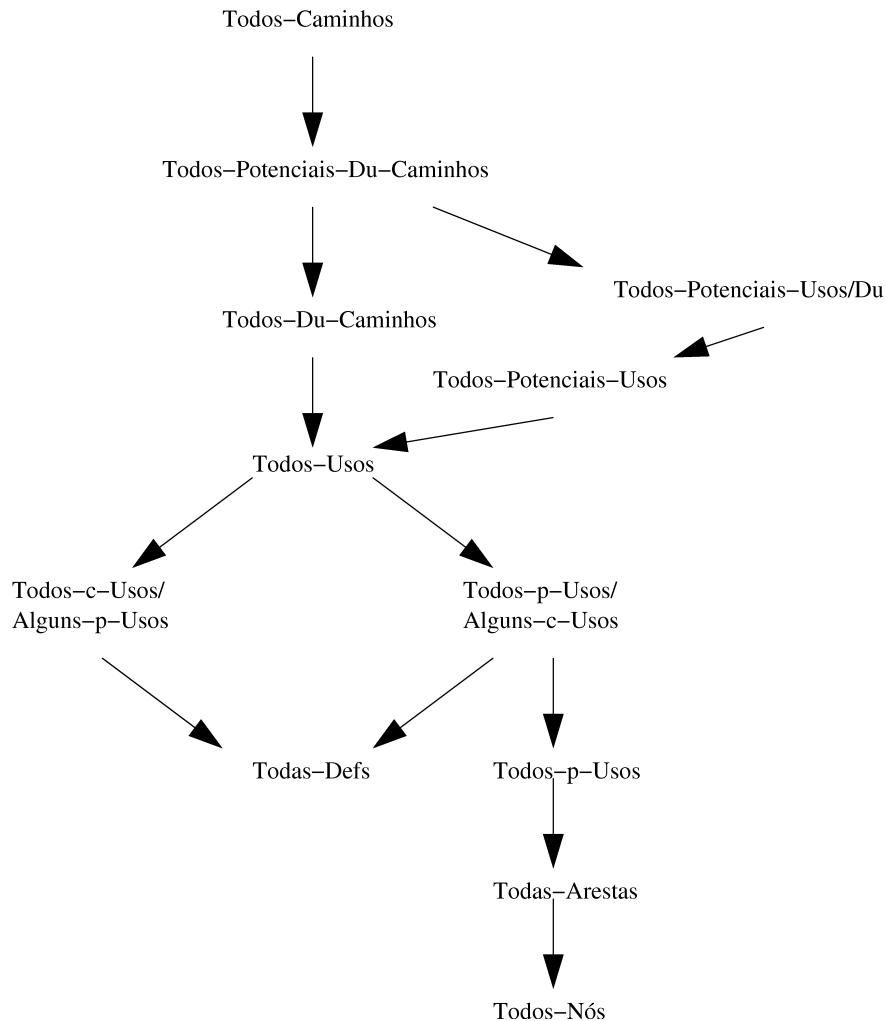


Figura 10.2 – Relação de inclusão dos critérios de fluxo de dados estendida com os critérios Potenciais-Usos.

potenciais du-caminhos seriam requeridos, o que iria exigir, no pior caso,  $2^t$  casos de teste para serem cobertos, sendo  $t$  o número de comandos de decisão, ou seja, a complexidade do critério Todos-Potenciais-Du-Caminhos é da ordem de  $2^t$ . Sendo Todos-Potenciais-Du-Caminhos o critério mais forte da família de critérios Potenciais-Usos, como pode ser visto na relação de inclusão da Figura 10.2, isso significa que, no pior caso,  $2^t$  é o limite superior de todos os critérios incluídos pelo critério Todos-Potenciais-Du-Caminhos, como, por exemplo, os critérios Todos-Potenciais-Usos e Todos-Usos. Resultados similares foram obtidos por Vincenzi [413] na análise de complexidade dos critérios estruturais orientados a objetos.

É importante observar que, embora, teoricamente, os critérios de teste de fluxo de dados apresentem uma complexidade exponencial, na prática tem-se observado que sua complexidade pode ser vista como linear em relação ao número de comandos de decisão  $t$ . Esses resultados foram obtidos por meio de experimentos realizados por Weyuker [430] e Maldonado [261].

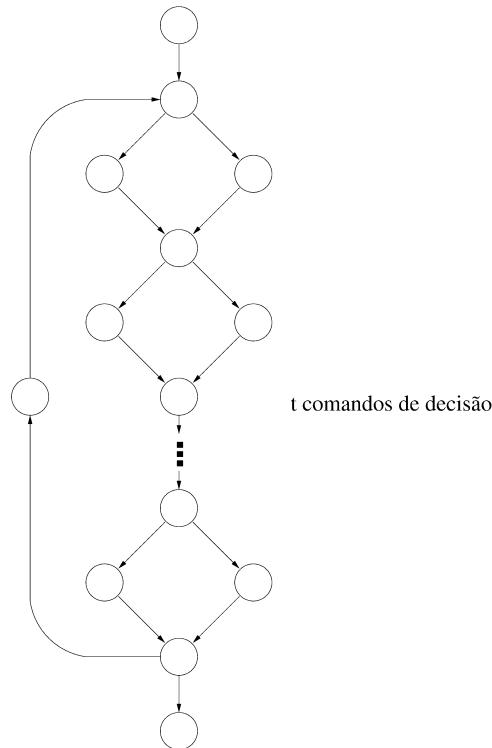


Figura 10.3 – Estrutura de controle que maximiza o número de potenciais du-caminhos.

A complexidade do teste de mutação tem sido determinada por meio da estimativa do número de mutantes gerados. No teste de mutação no nível de unidade, Budd [49] mostra que o número de mutantes gerados é proporcional ao número de variáveis do programa multiplicado pelo número de referências a essas variáveis. Esse resultado foi confirmado experimentalmente por Offutt et al. [312]. Deve-se observar que o número de mutantes depende do conjunto de operadores de mutação utilizado. Delamaro [108] analisou a complexidade do critério Mutação de Interface e concluiu que o número de mutantes gerados pelos operadores é proporcional ao número de variáveis vezes o número de referências às variáveis (similar à mutação no nível de unidade) e também ao número de parâmetros da função chamada.

Não existe uma correspondência direta entre a hierarquia estabelecida pela relação de inclusão e a capacidade de revelar a presença de erros, ou seja, mesmo que um critério  $C_1$  inclua um critério  $C_2$ , isso não implica necessariamente que  $C_1$  tenha uma melhor capacidade de revelar a presença de erros que  $C_2$ . Alguns autores têm abordado do ponto de vista teórico a questão de eficácia dos critérios, definindo outras relações de inclusão que captam a capa-

cidade de revelar erros dos critérios [145] ou indicando que a relação de inclusão proposta por Rapps e Weyuker [337] tem uma estreita relação com a capacidade de revelar erros [459]. Frankl e Weyuker [145] definiram cinco outras relações entre critérios e avaliaram essas relações usando três medidas probabilísticas que procuram agregar à hierarquia estabelecida por essas relações a capacidade de revelar a presença de erros, ou seja, refletir a eficácia do critério.

## 10.3 Estudos experimentais

Nesta seção são apresentados alguns dos trabalhos que procuram avaliar experimentalmente diversos critérios de teste. Para facilitar, dividimos esta seção em duas subseções. A primeira trata dos estudos desenvolvidos sobre critérios estruturais e baseados em erros. A segunda discute dos trabalhos relacionados a critérios funcionais

### 10.3.1 Técnicas estrutural e baseada em erros

O desenvolvimento de estudos experimentais sobre critérios de teste é motivado, principalmente, pelas seguintes questões:

- Tendo-se dois critérios de teste, qual é mais difícil de satisfazer?
- Qual o custo de um critério de teste?
- Quão bom é um critério de teste para revelar erros em um programa?
- Como diminuir o custo de um critério sem diminuir sua capacidade em detectar erros?
- Qual o efeito de diminuir o tamanho de um conjunto de casos de teste em relação à eficácia do critério?

Mathur [275] ilustra a seguinte situação para mostrar a importância de se avaliarem critérios de teste: considere a necessidade de testar um programa  $P$ , o qual é parte de um sistema crítico. O funcionamento correto desse sistema depende, obviamente, de  $P$ . O testador irá testar  $P$  tanto quanto for possível e, para isso, decide usar vários critérios de teste com o objetivo de verificar a adequação dos casos de teste desenvolvidos. Inicialmente, os casos de teste são gerados de modo a satisfazer um critério  $C_1$ . De posse disso, uma questão que surge é: *Tendo obtido um conjunto de casos de teste adequado ao critério  $C_1$ , e utilizando agora um critério  $C_2$ , é possível melhorar esse conjunto de casos de teste?* Essa é uma questão prática que surge diante da dificuldade em decidir se um programa foi suficientemente testado. O desenvolvimento de estudos experimentais procura, dessa forma, auxiliar a responder questões desse gênero.

No contexto de teste de software, a condução adequada de experimentos requer o desenvolvimento das seguintes atividades:

- seleção e preparação dos programas a serem utilizados;
- seleção das ferramentas de teste;

- geração de conjuntos de casos de teste adequados;
- execução dos programas com os casos de teste gerados;
- análise dos resultados do experimento.

Além dessas, atividades adicionais são necessárias, dependendo do tipo de experimento que será realizado. Por exemplo, quando se compara o critério Análise de Mutantes com os critérios de fluxo de dados, é necessário, durante a execução dos programas, identificar os mutantes equivalentes e caminhos não executáveis. Assim, essas atividades seriam adicionadas a esse experimento.

A geração aleatória costuma ser a opção feita para a geração de conjuntos de testes, devido à sua característica de ser facilmente automatizada e de gerar grandes conjuntos de casos de teste a baixo custo. Outra vantagem é o fato de eliminar qualquer influência possível do testador em conduzir a geração do conjunto de teste conforme conhecimento dos programas utilizados. Em geral, define-se o domínio de entrada de cada programa para a geração aleatória e, quando não se consegue satisfazer o critério, casos de teste criados manualmente são adicionados ao conjunto.

Conforme descrito antes, os critérios de teste baseados em fluxo de dados têm complexidade exponencial [261], o que motiva a condução de estudos experimentais para determinar o custo de aplicação desses critérios do ponto de vista prático. Vários estudos experimentais foram conduzidos para a determinação da complexidade desses critérios em termos práticos, ou seja, uma avaliação experimental desses e de outros critérios de teste objetivando a determinação de um modelo para a estimativa do número de casos de teste necessários. Essa determinação é muito importante para as atividades de planejamento do desenvolvimento. Weyuker [430] caracterizou um benchmark para a avaliação experimental da família de critérios de fluxo de dados; esse mesmo benchmark foi aplicado para uma primeira avaliação experimental dos critérios Potenciais-Usos [264, 405]. Com a aplicação do benchmark, obtiveram-se resultados bastante interessantes. Em geral, pode-se dizer que os critérios Potenciais-Usos, do ponto de vista prático, são factíveis e demandam um número de casos de teste relativamente pequeno.

Por meio de estudos experimentais têm-se obtido evidências de que o critério Análise de Mutantes também pode constituir na prática um critério atrativo para o teste de programas [274]. Tais experimentos, além de mostrar como a Análise de Mutantes se relaciona a outros critérios de teste, buscam novas estratégias a fim de reduzir os custos associados ao critério.

Mathur e Wong [274] compararam dois critérios de mutação alternativos: Mutação Aleatória (selecionado 10% de cada operador de mutação) e Mutação Restrita (selecionado um subconjunto de operadores de mutação). Esse experimento foi conduzido para comparar qual dessas estratégias apresentava melhor relação custo × eficácia. Segundo os autores, ambas mostraram-se igualmente eficazes, obtendo-se significativa redução no número de mutantes a serem analisados sem sensível perda na eficácia em revelar erros.

Em outro trabalho realizado por Mathur et al. [275] foi comparada a adequação de conjuntos de casos de teste em relação aos critérios Análise de Mutantes e Todos-Usos. O objetivo do experimento foi verificar a dificuldade de satisfação entre os dois critérios, bem como seus custos, uma vez que esses critérios são incomparáveis do ponto de vista teórico. Nesse estudo,

os conjuntos de casos de teste Análise de Mutantes-adequados também se mostraram Todos-Usos-adequados. No entanto, os conjuntos de casos de teste Todos-Usos-adequados não se mostraram, em muitos dos casos, adequados para o critério Análise de Mutantes. Esses resultados mostram que é mais difícil satisfazer o critério Análise de Mutantes do que o critério Todos-Usos, podendo-se dizer que, na prática, Análise de Mutantes inclui Todos-Usos [275].

Wong et al. [440] utilizaram a Mutação Aleatória (10%) e a Mutação Restrita para comparar o critério Análise de Mutantes com o critério Todos-Usos; o objetivo foi verificar o custo, a eficácia e a dificuldade de satisfação desses critérios. Os autores forneceram evidências de que os critérios Todos-Usos, Mutação Aleatória (10%) e Mutação Restrita representam, nesta ordem, o decréscimo do custo necessário para a aplicação do critério (número de casos de teste requeridos), ou seja, o critério Todos-Usos requer mais casos de teste para ser satisfeito do que a Mutação Restrita. Em relação à eficácia para detectar erros, a ordem (do mais eficaz para o menos) é Mutação Restrita, Todos-Usos e Mutação Aleatória. Observou-se, com isso, que examinar somente uma pequena porcentagem de mutantes pode ser uma abordagem útil na avaliação e na construção de conjuntos de casos de teste na prática. Desse modo, quando o testador possui pouco tempo para efetuar os testes (devido ao prazo de entrega do produto), pode-se usar o critério Análise de Mutantes para testar partes críticas do software, utilizando alternativas mais econômicas, tal como a Mutação Restrita ou o critério Todos-Usos, para o teste das demais partes do software, sem comprometer significativamente a qualidade da atividade de teste.

Offutt et al. [312] também realizaram um experimento, comparando o critério Análise de Mutantes com o critério Todos-Usos. Os resultados foram semelhantes àqueles obtidos por Wong et al. [440], ou seja, o critério Análise de Mutantes revelou um maior número de erros do que o critério Todos-Usos, e mais casos de testes foram necessários para satisfazer o critério Análise de Mutantes. Além disso, os conjuntos de casos de teste Análise de Mutantes-adequados foram adequados ao critério Todos-Usos, não sendo o inverso verdadeiro; resultado semelhante ao de Mathur [275].

Nos trabalhos de Wong et al. [437] e Souza [372] foram comparadas seis diferentes classes de Mutação Restrita quanto à eficácia em revelar erros. Analisou-se a eficácia das classes de mutação obtidas a partir dos operadores de mutação da ferramenta Proteum. Desse experimento pôde-se observar quais classes de mutação eram mais econômicas (baixo custo de aplicação) e eficazes. Com isso, foi possível o estabelecimento de uma ordem incremental para o emprego dessas classes de mutação, com base na eficácia e no custo de cada uma. Desse modo, os conjuntos de casos de testes podem ser construídos inicialmente de forma a serem adequados à classe com menor relação custo × eficácia. Na seqüência, quando as restrições de custo permitirem, esse conjunto pode ser melhorado de maneira a satisfazer as classes de mutação com maior relação custo × eficácia.

Souza et al. [375] realizaram um experimento com a finalidade de avaliar o *strength* e o custo do critério Análise de Mutantes, empregando, para efeito comparativo, os critérios Potenciais-Usos [261], os quais incluem o critério Todos-Usos. Os resultados indicaram que o custo de aplicação do critério Análise de Mutantes, estimado pelo número de casos de teste necessário para satisfazer o critério, apresentou-se maior do que o custo dos critérios Potenciais-Usos. Em relação à dificuldade de satisfação (*strength*), observou-se que, de uma maneira geral, os critérios Análise de Mutantes e Todos-Potenciais-Usos são incomparáveis mesmo do ponto de vista experimental. Já os critérios Todos-Potenciais-Usos/Du e Todos-Potenciais-Du-Caminhos [261] apresentaram maior *strength* que o critério Todos-Potenciais-

Usos em relação à Análise de Mutantes, o que motiva a investigação do aspecto complementar desses critérios quanto à eficácia.

Souza e Maldonado [373] avaliaram o efeito da minimização de conjuntos de casos de teste na eficácia em revelar a presença de erros do critério Análise de Mutantes. A minimização de conjuntos de casos de teste busca diminuir o tamanho do conjunto de modo que casos de teste obsoletos e redundantes sejam eliminados. O objetivo é reduzir o custo associado à utilização do critério, visto que, na maioria das vezes, o custo é estimado pelo tamanho do conjunto de casos de teste necessário para satisfazer o critério. Além disso, os testes de regressão também têm seus custos diminuídos, uma vez que utilizam o conjunto de casos de teste obtido durante a fase de desenvolvimento como base para a revalidação do software. Os resultados obtidos indicam que a minimização proporciona uma redução significativa no tamanho dos conjuntos de casos de teste, principalmente para conjuntos com maior cardinalidade. Em relação à eficácia, em alguns casos, houve uma pequena redução após a minimização dos conjuntos de casos de teste [373].

Um estudo interessante no contexto do critério Análise de Mutantes é o estudo realizado por Offutt et al. [310]. O objetivo era determinar um conjunto de operadores de mutação essenciais para o teste de programas Fortran. Os resultados obtidos mostraram que, dos 22 operadores de mutação utilizados pela ferramenta de teste Mothra, apenas cinco eram suficientes para aplicar com eficiência o teste de mutação. Com base nesse estudo, Barbosa [25] conduziu uma série de experimentos a fim de determinar o conjunto de operadores de mutação essenciais para a linguagem C, utilizando a ferramenta Proteum. No caso da Proteum, dos 71 operadores de mutação implementados, 8 se mostraram suficientes para garantir um escore de mutação bem próximo de 1. Esses estudos contribuem fortemente para a viabilização da aplicação do critério Análise de Mutantes em ambientes comerciais de desenvolvimento de software.

A mesma linha de estudos experimentais que investiga o critério Análise de Mutantes foi aplicada no âmbito do teste de integração, buscando avaliar experimentalmente o critério Mutação de Interface. Como apresentado por Maldonado et al. [265], com a proposição do critério Mutação de Interface torna-se evidente o aspecto positivo de se utilizar o mesmo conceito de mutação nas diversas fases do teste. Ainda, é natural a indagação sobre qual estratégia utilizar para se obter a melhor relação custo × eficácia quando são aplicados os critérios Análise de Mutantes e Mutação de Interface no teste de um produto. Assim sendo, Delamaro [108] e Vincenzi et al. [416] conduziram experimentos envolvendo o critério Mutação de Interface e suas abordagens alternativas. Os resultados obtidos indicaram que, assim como a Análise de Mutantes, o critério Mutação de Interface apresenta alta eficácia em revelar a presença de erros, porém com um alto custo de aplicação em termos do número de mutantes gerados. Vincenzi et al. [416] investigaram o relacionamento entre os critérios Análise de Mutantes e Mutação de Interface e como utilizar tais critérios de forma complementar na atividade de teste, tendo como objetivo contribuir para o estabelecimento de uma estratégia de teste incremental, de baixo custo de aplicação e que garanta um alto grau de adequação em relação a ambos os critérios. Os resultados obtidos indicam que, mesmo com um número reduzido de operadores, é possível determinar conjuntos de casos de teste adequados ou muito próximos da adequação para ambos os critérios, a um menor custo. Nessa mesma linha, Maldonado et al. [262] conduziram um experimento investigando a aplicação da mutação seletiva no teste de programas C, tanto em nível de unidade quanto de integração. De modo geral, para os conjuntos de programas utilizados, observou-se que com uma redução próxima a 80% no número de mutantes gerados, ainda assim seriam obtidos escores de mutação bastante signifi-

ficativos, em torno de 0,98, indicando que a mutação seletiva é uma abordagem promissora para a aplicação do teste de mutação.

### 10.3.2 Técnica funcional

O teste funcional tem sido explorado em alguns experimentos relatados na literatura, com o objetivo de compará-lo com outras técnicas de teste, como também com técnicas de inspeção de código. Em seguida descrevem-se alguns desses experimentos.

#### Experimento de Hetzel [179]

**Técnicas:** Leitura de código disciplinada, teste de especificação (teste funcional) e teste misto (combinação de teste funcional e teste estrutural).

**Artefatos de software:** Três programas escritos em PL/I, de 64, 164 e 170 instruções que continham defeitos que causavam respectivamente 9, 15 e 25 diferentes tipos de falhas.

**Principais objetivos:** Comparar as três técnicas no que diz respeito à efetividade em revelar falhas.

**Caracterização dos participantes:** Foram selecionados 39 participantes de acordo com sua experiência em programação em PL/I que, em média, era de três anos. O tipo dos participantes variava de alunos de graduação a programadores inexperientes.

**Projeto experimental:** O experimento foi dividido em duas fases: a fase de treinamento e a fase de execução do experimento. O treinamento consistiu em uma introdução teórica às técnicas, uma apresentação das especificações e uma visão geral dos três programas. O experimento foi constituído de três sessões, e os participantes foram divididos em três grupos. Como a ordem em que os participantes utilizaram os programas e aplicaram as técnicas foi randomizada, foi possível, por exemplo, que um indivíduo de determinado grupo aplicasse a técnica leitura de código disciplinada no programa 1 durante a primeira sessão, a técnica teste misto no programa 3 durante a segunda sessão e a técnica teste de especificação no programa 2 durante a terceira sessão. Assim, nem a técnica nem o programa eram fixos em determinada sessão.

#### Forma de utilização das técnicas

*Leitura de código disciplinada:* Durante a aplicação dessa técnica os participantes tiveram de, a partir da especificação, caracterizar as estruturas de código de mais baixo nível (trechos) e, pela combinação dessas estruturas, obter a especificação do programa como um todo. A especificação final foi comparada com a especificação original do programa para que as inconsistências fossem identificadas.

*Teste de especificação:* Durante a aplicação dessa técnica, que consiste em um teste funcional, os participantes utilizaram somente a especificação do programa para derivar os casos de teste. A detecção das falhas foi realizada por meio da comparação das saídas produzidas pelos casos de teste com a especificação original. Nenhum critério foi especificado para a criação dos casos de teste.

*Teste misto:* Durante a aplicação dessa técnica os participantes tiveram de combinar as técnicas de teste funcional e teste estrutural. Assim, os casos de teste foram gerados com

base na especificação, mas o objetivo foi verificar se com eles se alcançava 100% de cobertura de todas as instruções do programa. As falhas foram detectadas comparando as saídas produzidas pelos casos de teste com a especificação original.

**Principais resultados obtidos:** As duas técnicas de teste não apresentaram nenhuma diferença significativa com relação à efetividade; porém os participantes que aplicaram a leitura de código disciplinada tiveram uma efetividade menor do que quem aplicou as técnicas de teste. Os participantes observaram somente 50% das falhas existentes.

## Experimento de Myers [295]

**Técnicas:** Teste funcional, teste estrutural e Walkthrough/Inspection method.

**Artefatos de software:** Um programa escrito em PL/I de 63 declarações com um total de 15 defeitos.

**Principais objetivos:** Comparar as três técnicas no que diz respeito à efetividade e à eficiência em revelar falhas.

**Caracterização dos participantes:** 59 programadores profissionais de um curso promovido para os funcionários da IBM.

**Projeto experimental:** Com base na experiência com programação em PL/I e no conhecimento das técnicas de Walkthrough/Inspection foram formados três grupos, sendo que cada grupo aplicou uma das três técnicas.

### Forma de utilização das técnicas

*Teste funcional:* Durante a aplicação dessa técnica os participantes tiveram acesso somente à especificação do programa para derivar os casos de teste e a uma versão executável do programa. Após a execução dos casos de teste, as falhas foram observadas através da comparação dos resultados da atividade de teste com a especificação. Nenhum critério foi especificado para a criação dos casos de teste.

*Teste estrutural:* Durante a aplicação dessa técnica os participantes tiveram acesso à especificação do programa e ao código-fonte. Nenhum critério foi especificado para a criação dos casos de teste ou a avaliação de cobertura de critério.

*Walkthrough/Inspection method:* Essa técnica corresponde a uma análise do código, com o objetivo de encontrar defeitos. Durante a aplicação dessa técnica os participantes testaram o programa, individualmente, de maneira *ad hoc*. Posteriormente, foi realizado um encontro para coletar as falhas.

**Principais resultados obtidos:** Não foi observada diferença significante em relação à efetividade das três técnicas. Em relação à quantidade de tempo gasto para revelar falhas, o método Walkthrough/Inspection gastou o maior tempo, o teste funcional gastou um pouco menos e o teste estrutural gastou o menor tempo deles. Myers constatou que os participantes ignoraram muitas das falhas existentes e isolaram, na média, somente 1/3 dos defeitos conhecidos. Com relação às combinações teóricas das técnicas, foi observado que, na média, 33% das falhas foram detectadas pela aplicação de cada uma das técnicas de maneira individual e, usando-as de forma combinada, foi possível detectar 50% de falhas.

## Experimento de Basili e Selby [26]

**Técnicas:** Leitura de código, teste funcional e teste estrutural.

**Artefatos de software:** Três programas escritos em Fortran de 169, 147 e 365 linhas de código, com 9, 6 e 12 defeitos, respectivamente.

**Principais objetivos:** Comparar as três técnicas no que diz respeito à efetividade e à eficiência em revelar falhas.

**Caracterização dos participantes:** Esse experimento foi realizado três vezes. Nas duas primeiras, participaram 42 estudantes com nível avançado de conhecimento; na terceira, participaram 32 desenvolvedores profissionais. Os dados a seguir estão relacionados à última das três replicações.

**Projeto experimental:** O treinamento consistiu em um tutorial de quatro horas sobre as técnicas. Na fase de execução do experimento os participantes aplicaram as técnicas em três sessões separadas, uma para cada técnica. Dessa forma, em cada sessão a técnica usada era a mesma e cada participante aplicou a técnica em um dos programas, fazendo com que cada participante usasse um dos programas em cada técnica.

### Forma de utilização das técnicas

*Leitura de código:* A técnica de leitura Abstração Passo a Passo (*Stepwise Abstraction*) [242] foi aplicada no código-fonte do programa, criando-se abstrações gradativas do código, partindo-se das estruturas mais internas e caminhando em direção às estruturas mais externas.

*Teste funcional:* Durante a aplicação dessa técnica os participantes analisaram a especificação e geraram casos de teste baseados nos critérios Particionamento em Classes de Equivalência e Análise do Valor Limite. Os casos de teste foram executados, e, para que as falhas pudessem ser identificadas, as saídas obtidas foram comparadas com as saídas esperadas.

*Teste estrutural:* Durante a aplicação dessa técnica os participantes derivaram casos de teste do código-fonte do programa até que uma cobertura de 100% das instruções fosse obtida. Os casos de teste foram executados e as saídas obtidas foram comparadas com a saída esperada para que as falhas pudessem ser identificadas.

**Principais resultados obtidos:** O maior percentual de inconsistência detectado foi atribuído à técnica leitura de código, que, por ser uma técnica de leitura, identifica inconsistências e não falhas. Com relação às técnicas de teste, observou-se que a técnica de teste funcional detectou mais falhas que a técnica de teste estrutural. Os participantes observaram, em média, somente 50% das falhas.

## Experimento de Kamsties e Lott [207]

**Técnicas:** Leitura de código, teste funcional e teste estrutural.

**Artefatos de software:** Três programas escritos em C, com 44, 89 e 127 linhas de código, foram utilizados na fase de treinamento. Três outros programas também escritos em C, de 260, 279, 282 linhas de código, com 11, 14 e 11 defeitos, respectivamente, foram utilizados durante a execução do experimento.

**Principais objetivos:** Comparar as três técnicas com relação à efetividade e à eficiência em observar inconsistências/falhas e isolar defeitos.

**Caracterização dos participantes:** Esse experimento foi executado duas vezes, ambas com universitários, sendo que na primeira vez participaram 27 estudantes; e na segunda vez, 23.

**Projeto experimental:** Baseou-se no projeto construído por Basili e Selby [26]. No entanto, Kamsties e Lott acrescentaram uma etapa ao projeto que consistia em isolar os defeitos após a observação das falhas. O isolamento dos defeitos tinha como objetivo identificar no código-fonte o exato local que havia produzido a falha. O treinamento foi elaborado para que pudesse ser aplicado em duas partes: uma teórica e outra prática. Foi utilizado o princípio de randomização em relação a técnicas, programas, participantes e ordem de aplicação das técnicas. Foram formados seis grupos, e os programas foram utilizados de maneira fixada para que se tornasse possível prevenir a troca de informações sobre tais programas entre os participantes. Houve a manipulação da ordem de aplicação das técnicas, de forma que, para cada programa, dois grupos aplicaram cada uma das três técnicas.

### Forma de utilização das técnicas

*Leitura de código:* Os participantes aplicaram a técnica de leitura abstração passo a passo (*Stepwise Abstraction*) [242]. Em seguida, os replicadores copiaram a especificação gerada pelos participantes. O objetivo dessa cópia foi garantir que os participantes não iriam realizar alterações nas especificações nos passos seguintes. Depois disso, os participantes compararam a especificação produzida por eles com a especificação original do programa para identificar inconsistências. Após isso, passou-se para o isolamento dos defeitos, sendo nenhuma técnica foi especificada para realizar esse passo.

*Teste funcional:* Durante a aplicação dessa técnica os participantes receberam a especificação do programa, mas não tiveram acesso ao código-fonte. Foi aplicado o Particionamento em Classes de Equivalência e a Análise do Valor Limite. Os casos de teste foram executados no computador e os resultados foram impressos. Os participantes foram orientados a não gerar casos de teste adicionais durante essa etapa. As saídas obtidas foram comparadas com a especificação para que as falhas pudessem ser identificadas. Em seguida, os participantes entregaram uma cópia de seus resultados aos replicadores e receberam o código-fonte impresso para isolar os defeitos. Para realizar esse isolamento não foi especificada nenhuma técnica.

*Teste estrutural:* Durante a aplicação dessa técnica os participantes derivaram casos de teste do código-fonte até alcançar 100% de cobertura de todos os arcos, todas as condições múltiplas, todos os loops e todos operadores relacionais. Os casos de teste foram executados e foram verificados os valores de cobertura obtidos. Casos de teste adicionais foram construídos até que 100% de cobertura fosse alcançada para todos esses critérios, ou até que os participantes estivessem convencidos de que não era possível alcançar uma melhor cobertura devido a caminhos não executáveis. Os resultados de execução foram impressos e as saídas foram comparadas com a especificação do programa para que as falhas fossem identificadas. Em seguida, os participantes entregaram uma cópia de seus resultados aos replicadores e receberam o código-fonte impresso para isolar os defeitos. Para realizar esse isolamento não foi especificada nenhuma técnica.

**Principais resultados obtidos:** Os participantes que aplicaram a técnica de leitura de código e a técnica de teste funcional obtiveram o mesmo percentual de falhas detectadas e os que aplicaram a técnica de teste estrutural obtiveram um percentual bem menor. A técnica

de teste funcional obteve um melhor resultado com relação à eficiência em isolar defeitos do que as outras técnicas. Na média, os participantes detectaram 50% das falhas.

## Experimento de Wood, Roper, Brooks e Miller [441]

**Técnicas:** Foram utilizadas as mesmas técnicas aplicadas por Kamsties e Lott [207] e Basili e Selby [26], ou seja, as técnicas: leitura de código, teste funcional e teste estrutural, com uma exceção – para a técnica teste estrutural foi aplicado somente o critério que cobre 100% das instruções do programa.

**Artefatos de software:** Foram utilizados os programas que estavam disponíveis no pacote de replicação do experimento de Kamsties e Lott [207], tanto para a fase de treinamento quanto para a fase de execução do experimento.

**Principais objetivos:** Comparar as três técnicas no que diz respeito à efetividade e à eficiência em revelar falhas. Foram selecionados alguns tipos de defeitos do total existente nos programas do pacote original.

**Caracterização dos participantes:** 47 alunos de graduação, sendo que todos eles já haviam completado dois anos fazendo disciplinas de programação.

**Projeto experimental:** Realizou-se um treinamento teórico em cada uma das técnicas e, em seguida, foram realizadas três sessões de treinamento prático em cada uma delas. A execução do experimento propriamente foi feita em três sessões, em semanas consecutivas. Organizaram-se seis grupos, balanceados de acordo com a habilidade individual de cada participante em relação à programação. A ordem de utilização dos programas foi fixada e alternou-se a técnica utilizada por cada grupo.

**Principais resultados obtidos:** Ao fazer uma análise particular de cada técnica, observou-se que todas as técnicas apresentaram uma taxa de efetividade similar quanto à observação de falhas e ao isolamento de defeitos. Quando combinadas, as técnicas apresentaram uma taxa de efetividade maior. Mesmo nos piores casos de combinações das técnicas, observou-se que a taxa de efetividade foi 13% maior do que a média de efetividade de cada uma das técnicas individualmente. Na média, todas as técnicas combinadas apresentaram uma taxa de efetividade 25% maior do que a média das técnicas individuais.

## Experimento de Dória [123]

**Técnicas:** As técnicas utilizadas no experimento de Dória [123] basearam-se nas técnicas aplicadas por Basili e Selby [26], Kamsties e Lott [207] e Wood et al. [441], com a diferença de que o teste estrutural foi substituído pelo teste incremental, no qual os critérios estruturais selecionados foram: Todos-Nós, Todas-Arestas, Todos-Usos, Todos-Potenciais-Usos e, além desses, foi também selecionado o critério Análise de Mutantes, da técnica de teste baseada em erros.

**Artefatos de software:** Os mesmos programas utilizados por Wood et al. [441], ou seja, artefatos que estavam disponíveis no pacote de replicação do experimento de Kamsties e Lott [207], tanto para a fase de treinamento quanto para a fase de execução do experimento.

**Principais objetivos:** Comparar as três técnicas com relação à efetividade e à eficiência em observar inconsistências/falhas e isolar defeitos.

**Caracterização dos participantes:** Doze estudantes do curso de pós-graduação da UFS-Car (Universidade Federal de São Carlos) e da USP (Universidade de São Paulo).

**Projeto experimental:** Este projeto experimental baseou-se no projeto construído inicialmente por Basili e Selby [26] e modificado por Kamsties e Lott [207], que acrescentaram a etapa para isolar os defeitos após a observação das falhas. O treinamento foi realizado em dois dias consecutivos e a fase de execução do experimento foi realizada ao longo de três semanas consecutivas. Os programas foram utilizados de uma maneira fixada para que fosse possível evitar a troca de informações sobre os programas entre os participantes.

### Forma de utilização das técnicas

*Leitura de código:* Com base no código-fonte impresso e sem acesso à especificação do programa, os participantes aplicaram a técnica de leitura abstração passo a passo (*Stepwise Abstraction*) [242], gerando uma abstração do programa. Com base nessa abstração, os participantes compararam a especificação original com a própria especificação em busca de inconsistências. Em seguida, os defeitos que produziram as inconsistências observadas foram isolados. Nenhuma técnica especial para isolar defeitos foi especificada.

*Teste funcional:* Os participantes receberam a especificação mas não tiveram acesso ao código-fonte do programa. Os casos de teste foram gerados utilizando os critérios Particionamento em Classes de Equivalência e Análise do Valor Limite da técnica de teste funcional. Em seguida, os casos de teste foram executados no computador e os resultados foram impressos. Com a especificação e os resultados em mãos, os participantes verificaram a existência de falhas. Com acesso ao código-fonte, os participantes isolaram os defeitos que causaram as falhas. Nenhuma técnica especial para isolar defeitos foi especificada.

*Teste incremental:* Compreendeu a aplicação dos critérios Todos-Nós, Todas-Arestas, Todos-Usos, Todos-Potenciais-Usos e Análise de Mutantes. No teste incremental o conjunto de casos de teste vai sendo melhorado por meio de um processo evolutivo que vai do critério Todos-Nós até o critério Análise de Mutantes. Assim, com base no critério Todos-Nós construíram-se casos de teste até alcançar 100% de cobertura desse critério. Utilizando-se a ferramenta POKE-TOOL [61], os casos de teste foram executados e foram verificados os valores de cobertura obtidos. Adicionaram-se novos casos de teste até obter-se 100% de cobertura para o critério ou até se acreditar que não seria possível alcançar cobertura melhor devido aos elementos não executáveis, ou seja, caminhos do programa que não podem ser executados devido à própria formação estrutural do código-fonte. Com a especificação em mão e os resultados da sessão de teste, foi verificada a existência de falhas. O código-fonte foi utilizado para isolar defeitos que causaram as falhas observadas. Nenhuma técnica especial para isolar defeitos foi especificada. Em seguida, com base no critério Todas-Arestas casos de teste foram construídos até alcançar-se 100% de cobertura desse critério. O conjunto de casos de teste inicial utilizado foi o conjunto obtido anteriormente para o critério Todos-Nós. Em seguida, o mesmo foi feito para os critérios Todos-Usos e Todos-Potenciais-Usos. Depois de um tempo preestabelecido no planejamento do experimento, pelo fato de os participantes poderem não ter identificado os elementos e associações não executáveis, essa relação foi entregue a eles, pois esses elementos podiam não permitir que os participantes atingissem 100% de cobertura dos critérios, como desejado. Finalmente, com base no critério Análise de Mutantes, utilizando o conjunto final de casos de teste gerado anteriormente, outros casos de teste foram produzidos até alcançar-se 100% de cobertura desse critério, o que significa alcançar um escore de mutação igual a um. Nesse passo foi utilizada a ferramenta Proteum [107]. Com a especificação em mão e os resultados da sessão de teste, foi verificada a existência de

fallas. Para finalizar, foi utilizado o código-fonte para isolar defeitos que causaram as falhas observadas. Nenhuma técnica especial para isolar defeitos foi especificada. Também nesse caso os participantes receberam a lista contendo os mutantes equivalentes, devido ao fato de esses mutantes não permitirem a obtenção de um escore igual a um.

**Principais resultados obtidos:** A técnica teste funcional obteve um melhor resultado com relação à eficiência em revelar falhas e isolar defeitos do que as outras técnicas. Na média, os participantes detectaram 26,96% das falhas. Se os resultados forem analisados em conjunto, explorando o aspecto complementar das técnicas, o percentual de falhas observadas e dos defeitos isolados pelos participantes é maior. Os piores resultados obtidos por qualquer combinação de técnicas foram sempre melhores que os piores resultados obtidos individualmente por determinada técnica.

## Experimento de Linkman et al. [243]

**Técnicas:** Teste aleatório; técnica funcional: critérios Particionamento de Equivalência, Análise do Valor Limite e Funcional Sistemático; e técnica baseada em erros: critério Análise de Mutantes.

**Artefatos de software:** Programa cal do Unix.

**Principais objetivos:** Avaliar a adequação do teste aleatório e funcional em relação ao critério Análise de Mutantes. Desse modo, diferentes conjuntos de testes foram gerados, e a capacidade de cada um em matar os mutantes do critério Análise de Mutantes foi determinada.

**Caracterização dos participantes:** Seis participantes com conhecimento sobre o programa e sobre a aplicação de critérios funcionais e aleatórios foram selecionados.

**Projeto experimental:** Nenhum treinamento aos participantes foi necessário, tendo em vista que eles já possuíam conhecimento na aplicação dos critérios selecionados. O único artefato submetido aos participantes foi a especificação textual do programa cal e solicitado o envio do conjunto de teste gerado, em função do critério de teste solicitado.

### Forma de utilização das técnicas

*Teste aleatório:* A partir de um pool de casos de testes gerados por Wong (1993), sete conjuntos de testes de diferentes cardinalidades foram gerados. Os conjuntos são referenciados por CTA1, CTA2, CTA3, CTA4, CTA5, CTA6, CTA7 e possuem 10, 20, 30, 40, 50, 60 e 70 casos de testes, respectivamente.

*Funcional Sistemático:* Com base na especificação funcional do programa cal, um dos participantes gerou um conjunto de teste, referenciado por CTFS, contendo ao todo 76 casos de testes.

*Particionamento de Equivalência e Análise do Valor Limite:* Novamente, com base na especificação do programa cal, outros quatro conjuntos de testes foram gerados. Tais conjuntos, referenciados por CTPA1, CTPA2, CTPA3, CTPA4, são compostos por casos de testes gerados a partir de ambos os critérios funcionais (Particionamento de Equivalência e Análise do Valor Limite) e possuem cardinalidades de 21, 15, 21 e 14 casos de testes, respectivamente.

*Análise de Mutantes:* Esse critério foi utilizado para avaliar a eficácia dos conjuntos de testes gerados em revelar a presença de defeitos. Para isso, todos os operadores de muta-

ção para programas C foram aplicados no programa cal e os mutantes equivalentes foram determinados manualmente. Ao todo, 4.624 mutantes foram gerados, dos quais 335 foram identificados como equivalentes. Assim, todos os mutantes restantes podem ser mortos por algum caso de teste.

**Principais resultados obtidos:** A Tabela 10.1 ilustra para cada caso de teste a quantidade de mutantes vivos, a porcentagem de mutantes vivos em relação ao total de mutantes gerados, o escore de mutação obtido e o número de mutantes vivos agrupados por classe de operador de mutação.

Tabela 10.1 – Cobertura dos conjuntos de casos de testes funcionais em relação ao critério Análise de Mutantes

| Conjunto de Teste            | Mutantes Vivos | Porcentagem de Vivos | Escore   | Mutantes Vivos por Classe de Operador |          |           |          |
|------------------------------|----------------|----------------------|----------|---------------------------------------|----------|-----------|----------|
|                              |                |                      |          | Constante                             | Operador | Statement | Variável |
| TS <sub>SFT</sub>            | 0              | 0                    | 1,000000 | 0                                     | 0        | 0         | 0        |
| TS <sub>PB<sub>1</sub></sub> | 371            | 8,02                 | 0,913500 | 193                                   | 78       | 27        | 73       |
| TS <sub>PB<sub>2</sub></sub> | 74             | 1,60                 | 0,982747 | 33                                    | 22       | 0         | 19       |
| TS <sub>PB<sub>3</sub></sub> | 124            | 2,68                 | 0,971089 | 58                                    | 31       | 13        | 22       |
| TS <sub>PB<sub>4</sub></sub> | 293            | 6,34                 | 0,931686 | 116                                   | 84       | 16        | 77       |
| TS <sub>RA<sub>1</sub></sub> | 1875           | 40,55                | 0,563242 | 944                                   | 539      | 103       | 289      |
| TS <sub>RA<sub>2</sub></sub> | 558            | 12,07                | 0,870021 | 287                                   | 161      | 21        | 89       |
| TS <sub>RA<sub>3</sub></sub> | 419            | 9,06                 | 0,902399 | 216                                   | 113      | 15        | 75       |
| TS <sub>RA<sub>4</sub></sub> | 348            | 7,53                 | 0,918938 | 181                                   | 87       | 12        | 68       |
| TS <sub>RA<sub>5</sub></sub> | 311            | 6,73                 | 0,927557 | 159                                   | 77       | 11        | 64       |
| TS <sub>RA<sub>6</sub></sub> | 296            | 6,40                 | 0,931051 | 149                                   | 73       | 11        | 63       |
| TS <sub>RA<sub>7</sub></sub> | 69             | 1,49                 | 0,983927 | 21                                    | 30       | 0         | 18       |

Como pode ser evidenciado pelo experimento, o único conjunto de teste que foi capaz de matar todos os mutantes, evidenciando todos os defeitos modelados por esses mutantes, foi o conjunto CTFS, obtido a partir do critério Funcional Sistemático.

Somente dois outros conjuntos de testes atingiram escore de mutação acima de 0,98 mas inferior a 1,00: CTPA2 e CTA17. Na média, considerando os conjuntos de testes aleatórios CTA11, CTA12 e CTA13, observou-se que apresentaram escores de mutação de 0,56, 0,87, e 0,90, respectivamente, todos inferiores aos escores de mutação acima de 0,91 obtidos pelos conjuntos funcionais CTPA1, CTPA2, CTPA3 e CTPA4.

Como o conjunto de teste obtido pelo critério Funcional Sistemático possui 76 casos de testes e somente 21 destes foram efetivos, ou seja, mataram ao menos um mutante, comparando os resultados obtidos por CTA12 e CTA17, os escores de mutação obtidos foram menores, 0,870 e 0,984, respectivamente. Tais valores são 13 e 1,6% abaixo do escore determinado por CTFS.

Embora estudos de casos adicionais sejam necessários para que os resultados possam ter validade estatística, os dados obtidos com o estudo de caso descrito evidenciam que, se aplicados corretamente, os critérios de testes funcionais podem determinar um alto grau de cobertura do código-fonte do produto em teste, revelando um grande número de defeitos com baixo custo de aplicação.

## 10.4 Considerações finais

Neste capítulo foram apresentados os principais estudos teóricos e experimentos relacionados à atividade de teste de software. Esses estudos fornecem indícios do custo de aplicação, eficácia e aspecto complementar dos critérios de teste. Essas informações são fundamentais para o estabelecimento de uma estratégia de aplicação dos critérios de teste.

Em seu trabalho, Harrold [166] apresenta as perspectivas de pesquisas na área de teste de software, objetivando o desenvolvimento de métodos e ferramentas que permitam a transferência de tecnologia para a indústria. Harrold aponta que o desenvolvimento de estudos para analisar as propriedades dos critérios de teste é fundamental para identificar classes de erros e critérios de teste efetivos para cada uma das classes.

De uma maneira geral, a análise das propriedades dos critérios indica que é possível viabilizar na prática a aplicação de critérios mais caros, como os critérios de fluxo de dados e teste de mutação. Outro ponto importante observado é o aspecto complementar dos critérios. Esse aspecto é ressaltado nos estudos descritos neste capítulo. Seu objetivo é formar um corpo de conhecimento que favoreça o estabelecimento de estratégias de teste incrementais que explorem as diversas características dos critérios. Nessas estratégias seriam aplicados inicialmente critérios mais fracos e talvez menos eficazes para a avaliação da adequação do conjunto de casos de teste, e, em função da disponibilidade de orçamento e de tempo, incrementalmente, poderiam ser utilizados critérios mais fortes e eventualmente mais eficazes, porém, em geral, mais caros. Estudos experimentais são conduzidos no sentido de avaliar os aspectos de custo, *strength* e eficácia dos critérios de teste, buscando contribuir para o estabelecimento de estratégias de teste eficazes, de baixo custo e para a transformação do estado da prática, no que tange ao uso de critérios e ferramentas de teste. Esses trabalhos têm obtido avanços significativos, auxiliando no processo de desenvolvimento de software de alta qualidade.

# Capítulo 11

## Geração de Dados de Teste

*Silvia Regina Vergilio (DInf/UFPR)*  
*José Carlos Maldonado (ICMC/USP)*  
*Mario Jino (DCA/FEEC/UNICAMP)*

### 11.1 Introdução

Testar um programa com todos os seus possíveis valores de entrada ou executar todos os seus caminhos é idealmente desejável, mas impraticável. A maioria dos critérios de teste, descritos nos capítulos anteriores, relaciona valores do domínio de entrada de um programa, agrupando-os em partições não necessariamente disjuntas. Em geral, os critérios de teste dividem o domínio de entrada do programa em subdomínios e requerem que pelo menos um ponto de cada subdomínio seja executado. Por exemplo, o critério estrutural Todos-Nós agrupa em um subdomínio todas as entradas que executam um determinado nó.

Uma vez particionado o domínio, a questão é : “Que pontos de cada subdomínio devem ser escolhidos?” Isso diz respeito à tarefa de geração de dados de teste para satisfazer um determinado critério, ou seja, dados para executar um caminho para cobrir o elemento requerido. Muito embora a automatização dessa tarefa seja desejável, não existe um algoritmo de propósito geral para determinar um conjunto de teste que satisfaça um critério. Nem mesmo é possível determinar automaticamente se esse conjunto existe [141]. O problema de geração de dados de teste é indecidível, sendo que existem restrições inerentes às atividades de teste que impossibilitam automatizar completamente a etapa de geração de dados de teste. Entre elas, destacam-se:

- correção coincidente: ocorre quando o programa em teste possui um defeito que é alcançado por um dado de teste, um estado de erro é produzido, mas coincidentemente um resultado correto é obtido. Alguns autores afirmam que correção coincidente ocorre muito raramente e ao definir suas estratégias supõem que ela não está presente. Considere o Programa 11.1. Um caminho incorreto do programa poderá ser tomado, mas uma saída correta poderá ser produzida. Suponha que na linha 4 o predicado correto seja  $y < 0$ . O caso de teste ( $x = -1, y = 0$ ) deveria executar a parte `else` (linhas 6 e 7), mas devido ao erro executa a parte `then` (linha 5). Entretanto, coincidentemente, o resultado produzido é 4, igual ao esperado. Um outro dado de teste tal como ( $x = 2, y = 0$ ) revelaria o erro;

---

```

1 void calculo (int x, int y)           Programa 11.1
2 {
3     if (y <= 0)
4         printf ("%d", x+x+6-y*y);
5     else
6         printf ("%d", x*x+3-y*y);
7 }
```

---

- caminho ausente: corresponde a uma determinada funcionalidade requerida para o programa, mas que por engano não foi implementada, isto é, o caminho correspondente não existe no programa. Critérios estruturais de teste raramente auxiliam a determinação de caminhos ausentes, isto porque eles selecionam dados de teste baseados no código do programa e a especificação não é considerada. Um exemplo para calcular a área e o tipo de um triângulo – escaleno (retângulo, obtusângulo ou acutângulo), equilátero ou isóceles – é dado no Programa 11.2 [85, 335]. Os lados devem ser fornecidos em ordem decrescente, de tal maneira que  $a \geq b \geq c$ . Note que os pontos dados pelo caso de teste ( $a = 2, b = 1, c = 1$ ) não formam um triângulo. A saída produzida deveria ser `class = 0`, mas não existe um caminho no programa que faça o teste ( $a < b + c$ ), e uma saída incorreta é produzida;

---

```

1 void triangulo(int a, int b, int c)           Programa 11.2
2 {
3     int class;
4     double area,as, bs, cs;
5     if ((a<b) || (b<c))
6     {
7         class = -1;
8         area = 0; /* entrada nao esta forma esperada */
9     }
10    else if ((a!=b) && (b!=c))
11    {
12        as = a*a;
13        bs = b*b;
14        cs = c*c; /* escaleno */
15        if (as ==bs +as)
16        {
17            class = 3;
18            area = b*c/2.0; /* retangulo */
19        }
20        else
21        {
22            s = (a+b+c)/2.0;
23            area = sqrt(s*(s-a)*(s-b)*(s-c));
24            if (as<bs+cs)
25                class = 4; /* acutangulo */
26            else
27                class = 5; /* obtusangulo */
28        }
29    }
30    else if ((a=b)&&(b=c))
31    {
32        class = 1; /* equilatero */
33        area = a*a*sqrt(3.0)/4.0;
34    }
35    else
36    {
```

---

```

37         class = 2; /* isoceles */
38         if (a==b)
39             area = c*sqrt(4*a*b-c*c)/4;
40         else
41             area = a*sqrt(4*b*c-a*c)/4;
42     }
43     printf("%d", class); printf("%f", area);
44 }

```

---

- caminhos não-executáveis: como mencionado no Capítulo 4, um caminho é dito não-executável se não existe um conjunto de valores atribuídos às variáveis de entrada do programa, parâmetros e variáveis globais que causam sua execução. Um elemento requerido por um dado critério estrutural é não-executável se não existir caminho executável que o cubra. O problema de caminhos não-executáveis vem sendo tratado por vários autores [141, 177, 266]. Heurísticas têm sido utilizadas para determiná-los e evitar que esforço e tempo sejam gastos tentando gerar dados de teste para esses caminhos. Um caminho que não executa o laço no Programa 11.3 é um exemplo de caminho não-executável, pois isso não acontece independentemente da entrada fornecida;

---

Programa 11.3

---

```

1 void soma()
2 {
3     int i, s=0;
4     for (i = 0; i < 10; i++)
5         s = s + pow (i,2);
6     printf("%d",s);
7 }

```

---

- mutantes equivalentes: como mencionado no Capítulo 5, muitas vezes a modificação feita no programa original para criar um mutante não altera a função implementada, ou seja, o programa mutante implementa a mesma função que o programa original. Nesse caso, o mutante é dito equivalente. O problema de mutantes equivalentes é análogo ao de caminhos não-executáveis. A determinação de mutantes equivalentes também é indecidível e só é possível utilizando-se heurísticas [20, 96]. Um programa mutante criado substituindo-se o primeiro `if` do Programa 11.2 por `if ((a > b) + (b < c))` é equivalente ao programa original.

Apesar das limitações existentes, encontram-se na literatura diferentes categorias de técnicas que podem ser utilizadas para gerar dados de teste para satisfazer os diversos critérios. As mais conhecidas são: geração aleatória de dados [122, 161]; geração com execução simbólica [44, 83, 188, 335]; geração com execução dinâmica [220]; geração utilizando técnicas de computação evolutiva; e geração de dados sensíveis a defeitos, cujos fundamentos permitem escolher pontos do domínio de entrada relacionados a certos tipos de defeitos e, por isso, com alta probabilidade de que esses sejam revelados. Nessa última categoria destacam-se as técnicas: teste de domínios [433], teste baseado em restrições [116], teste do operador booleano (BOR) e teste dos operadores booleano e relacional (BRO) [382] e teste de expressões booleanas e relacionais com parâmetro  $\varepsilon$  (BRE( $\varepsilon$ )) [382].

Como dito anteriormente, a tarefa de geração de dados de teste busca escolher pontos do domínio de entrada para satisfazer um dado critério. Na geração aleatória, pontos do domínio de entrada são selecionados aleatoriamente até que o critério de teste seja satisfeito. Essa

técnica é defendida por vários autores [37, 122, 161] por ser prática e mais fácil de automatizar, além de menos custosa. No entanto, gerar dados de teste aleatoriamente não garante a satisfação do critério e também não auxilia a determinação de elementos não-executáveis ou mutantes equivalentes, pouco se aprendendo sobre o programa. Outra questão é que gerar aleatoriamente não garante a seleção dos melhores pontos. Os melhores pontos são os que têm maior probabilidade de revelar defeitos. A estratégia utilizada para selecionar esses pontos é fundamental, pois dela depende a eficácia dos dados de teste gerados. Este capítulo descreve as principais técnicas de geração de dados de teste que podem ser utilizadas para satisfazer os diversos critérios de teste existentes e para selecionar os “melhores” pontos. As técnicas são descritas, e uma discussão das principais vantagens e desvantagens da aplicação de cada uma delas é apresentada.

## 11.2 Geração com execução simbólica

A idéia de execução simbólica e sua aplicação ao teste de programas não são novas. Na década de 1970, vários pesquisadores dedicaram-se a esse tema [44, 83, 188, 335]. Em geral, a execução simbólica é utilizada conjuntamente com a técnica baseada em caminhos e tem o objetivo de auxiliar a geração automática de dados de teste para um dado caminho ou conjunto de caminhos. Com a execução simbólica é possível: 1) detectar caminhos não-executáveis; 2) criar representações simbólicas da execução de um caminho que poderão ser comparadas com representações simbólicas das saídas esperadas, facilitando a detecção de um defeito.

Técnicas de execução simbólica derivam expressões algébricas que representam a execução de um determinado caminho. O resultado é uma expressão simbólica chamada computação do caminho que expressa as variáveis de saída em termos das variáveis de entrada [141], e uma expressão chamada condição do caminho que expressa condições para o caminho ser executado.

Variáveis de entrada de um programa são as que recebem um valor por meio de uma comunicação externa ao programa, como, por exemplo: variáveis em comandos de leitura, parâmetros de entrada e variáveis globais. As outras variáveis são chamadas internas. Variáveis de saída comunicam valores ou resultados de computações, como, por exemplo, variáveis globais, parâmetros de saída, variáveis em comandos de escrita e de retorno de funções.

Inicialmente, valores simbólicos são atribuídos às variáveis de entrada. Cada comando associado aos nós do caminho é avaliado em termos desses valores. Quando uma atribuição é executada, os valores das variáveis do lado direito são substituídos pelos seus valores simbólicos. O resultado é simplificado e será o novo valor simbólico atribuído à variável que está do lado esquerdo da atribuição. Da mesma maneira, se um comando condicional é encontrado, o predicado associado é descrito por uma restrição em termos dos valores simbólicos das variáveis nele envolvidas. Essa restrição é chamada de interpretação do predicado.

A computação do caminho é obtida pela execução simbólica de todos os comandos que compõem o caminho, representada pelos valores simbólicos atribuídos às variáveis de saída. A condição do caminho é dada pelo conjunto de restrições associadas a todos os predicados encontrados ao longo do caminho. O domínio do caminho é dado pelo conjunto de valores que satisfazem a condição do caminho. O dado de teste é, então, gerado escolhendo-se um

elemento desse conjunto. Nem sempre é possível determinar uma solução que satisfaça a condição do caminho, ou seja, determinar se o caminho é ou não executável.

Para exemplificar os conceitos apresentados, considere o Programa 11.4 e seu grafo apresentados na Figura 11.1. Sejam  $K$  e  $J$  os valores simbólicos para as variáveis de entrada  $k$  e  $j$ . Após a execução do caminho  $(1, 2, 4)$ , o valor da variável  $j$  será o valor simbólico  $J + 1 - K$ . A restrição criada para o caminho será  $J + 1 > K$ . Para o caminho completo  $(1, 2, 4, 6)$  tem-se a computação do caminho dada pelo valor de  $j$  que é  $J + 1 - K$  e a condição do caminho é  $(J + 1 > K) \&\& (J - K > 0)$ , formada por restrições associadas aos predicados dos nós 1 e 4. O dado de teste  $K = 1$  e  $J = 3$  satisfaz essa restrição e executa o caminho  $(1, 2, 4, 6)$ .

---

Programa 11.4

---

```

1 void f(double j,double k)
2 {
3 /* no' 1 */      j = j + 1;
4 /* no' 1 */      if (j > k)
5 /* no' 2 */      j = j - k;
6 /* no' 3 */      else
7 /* no' 3 */      j = k - j;
8 /* no' 4 */      if (j <= 1)
9 /* no' 5 */      j = j - 1;
10 /* no' 6 */     printf("%f",j);
11 }
```

---

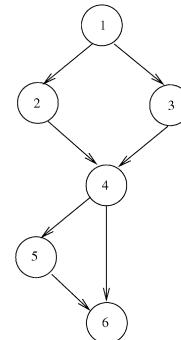


Figura 11.1 – Exemplificando os conceitos de execução simbólica.

Para se derivarem expressões simbólicas, existem alguns problemas discutidos a seguir:

- Laços: a execução simbólica pode tratar laços quando o número de iterações é conhecido. Quando isso não ocorre, Howden [188] propõe executar o laço  $k$  vezes, sendo que  $k$  pode ser escolhido pelo usuário ou pelo sistema. Isso pode gerar algumas inconsistências e gerar restrições sem solução, mas essa técnica é, em geral, adotada por questões de simplicidade.
- Referências a variáveis compostas, tais como vetores e matrizes, e apontadores: o problema é identificar a qual variável se faz referência. O Programa 11.5 ilustra o problema com um vetor. Sendo  $i$  e  $j$  variáveis de entrada com valores simbólicos, a quais elementos do vetor a condição se refere? Boyer et al. [44] propõem que todas as possibilidades sejam analisadas, mas esse número pode crescer rapidamente. Ramamoorthy et al. [335] sugerem que uma instância do vetor seja criada sempre que não for possível identificar qual elemento está sendo definido. A cada instância do vetor os elementos têm o mesmo valor simbólico, exceto aquele que acabou de receber um valor.

Por exemplo, seja  $A_k$  a  $k$ -ésima instância do vetor  $A$ . Se o comando  $A[m] = P$  é executado, uma nova instância  $k + 1$  de  $A$  é criada, tal que:

$$A_{k+1}[i] = A_k, \text{ se } i \neq m.$$

$$A_{k+1}[m] = P, \text{ caso contrário.}$$

Quando uma referência é feita a um vetor, sua última instância é utilizada. As instâncias permitirão que ambigüidades sejam posteriormente resolvidas. O número de

instâncias pode ser reduzido se heurísticas forem implementadas para determinar equivalência entre índices de vetores.

---

```

1 void f()
2 {
3     ....
4     scanf ("%d %d", &i, &j);
5     A[0] = 0;
6     A[1] = 1;
7     ....
8     A[10] = 10;
9     if (A[i] < A[j])
10    ....
}

```

---

Programa 11.5 \*

---

Essa equivalência está relacionada a variáveis que representam a mesma posição de memória. Além do caso  $A[i]$  e  $A[j]$ , quando  $i = j$ , também ocorrem outros, como, por exemplo, relacionados com apontadores  $P$  e  $Q$  para uma mesma variável. É necessário estudar técnicas heurísticas que possam ajudar no tratamento desses casos.

- Chamadas de funções ou procedimentos: representar em uma expressão o resultado da execução de um módulo não é uma tarefa trivial; isso equivaleria a realizar avaliação simbólica global, que possui inúmeras restrições de implementação [141]. O que pode ser feito é escolher um caminho na função chamada. Se inconsistências forem geradas, outro caminho poderá ser escolhido.

Apesar das restrições apresentadas, a execução simbólica é uma técnica bastante difundida. Ela vem sendo utilizada há vários anos. Na década de 1970, surgiram muitos sistemas para geração automática de dados de teste baseados em execução simbólica. Entre esses: o SELECT [44], o CASEGEN [335] e o DISSECT [188]. A maior dificuldade em tais sistemas é resolver a condição do caminho criada.

A execução simbólica é mais custosa do que a geração aleatória, mas, em geral, é mais eficaz. Howden [188] realizou um experimento com um conjunto de programas incorretos. Os dados gerados com execução simbólica revelaram aproximadamente 50% dos erros. A maioria dos erros encontrados foi de computação. Erros de domínio e/ou anomalias de fluxo de dados raramente foram determinados pela execução simbólica.

### 11.3 Geração com execução dinâmica

A técnica dinâmica de geração de dados de teste foi proposta por Korel [220]. Ela surgiu como uma alternativa à execução simbólica para solucionar problemas com arranjos e variáveis dinâmicas. Está baseada na execução real do programa em teste, em métodos de minimização de funções e análise de fluxo de dados. Dados reais são atribuídos às variáveis de entrada, o fluxo de execução do programa é monitorado. Se uma aresta incorreta do grafo de fluxo de controle é executada, métodos de minimização de funções são utilizados para determinar valores para as variáveis de entrada para os quais a aresta correta seria tomada. Além disso, a análise de fluxo de dados dinâmica é utilizada para determinar quais variáveis de entrada são responsáveis pelo comportamento incorreto do programa. Análise dinâmica e *backtracking* permitem o tratamento de arranjos e apontadores.



A técnica considera predicados simples e lineares da forma  $E_1 \ op \ E_2$ . Todo predicado pode ser transformado na forma  $F \ rel \ 0$ , em que  $rel \in \{<, \leq, =\}$ . Por exemplo, o predicado  $a > c$  pode ser transformado em  $c - a \leq 0$ .

A técnica tem como objetivo encontrar um conjunto  $x^0$  de valores de entrada que executem um dado caminho  $P$ . Korel afirma que esse objetivo pode ser reduzido a uma seqüência de subobjetivos, na qual cada subobjetivo será mais bem resolvido utilizando-se técnicas de minimização de funções.

Se  $x^0$  é o conjunto de valores iniciais, que poderia ter sido selecionado aleatoriamente, para o qual o caminho  $P = (n_1, \dots, n_i, \dots, n_m)$  é executado, então  $x^0$  é a solução para o problema de geração. Senão, suponha que se  $P'$  é o caminho executado,  $P_1 = (n_1^P, n_2^P, \dots, n_i^P)$  é o mais longo subcaminho de  $P'$ , em que  $(n_1^P = n_1, \dots, n_i^P = n_i)$  e que o caminho correto não foi executado porque o predicado  $F_i(x) \ rel_i \ 0$ , associado ao arco  $(n_i, n_{i+1})$ , não foi avaliado como se esperava. Então o objetivo é encontrar entradas  $x$ , em que a condição  $F_i(x) \ rel_i \ 0$  seja satisfeita. A função  $F_i(x)$  pode ser minimizada até que ela se torne negativa (ou 0 dependendo de  $rel_i$ ). Uma vez resolvido o primeiro subobjetivo, outros deverão ser resolvidos até que a solução do objetivo principal seja encontrada ou ainda até que se decida que o subobjetivo não poderá ser resolvido (nesse caso, a pesquisa falha).

A aplicação dos métodos de pesquisa é facilitada por causa das suposições feitas com relação aos predicados. Entre esses métodos, utiliza-se o mais simples, chamado método da variável alternativa, que consiste em minimizar com respeito a somente uma variável de entrada por vez. Uma variável  $x_1$  é escolhida, as outras são mantidas constantes até que a solução seja encontrada, ou seja, quando a função associada à aresta se torne negativa. Se o mínimo para a função é um valor de  $x_1$  positivo, então a pesquisa continua da mesma forma com respeito a outra variável  $x_2$ . A pesquisa falhará se todas as variáveis forem analisadas e a função não puder ser decrementada.

O procedimento para resolver um subobjetivo é composto de duas fases: uma pesquisa exploratória e um movimento padrão. Durante a pesquisa exploratória, uma variável  $x_i$  é incrementada ou decrementada. Se em alguma dessas mudanças o caminho  $P$  for executado, encontrou-se a solução. Caso contrário, se  $F_i$  diminui (melhora) quando  $x_i$  aumenta, então deve-se aumentar  $x_j$ ; se diminui quando  $x_i$  diminui, então  $x_i$  deve diminuir. Se o aumento ou diminuição de  $x_i$  não afeta ou aumenta  $F_i$ , então a próxima variável  $x_{i+1}$  é considerada.

Se a pesquisa exploratória realizada indicou uma direção a ser tomada, uma mudança de padrão, que é uma mudança de maiores proporções (aumento ou decremento da variável  $x_i$ ), é executada.

Para exemplificar o método apresentado, considere o Programa 11.6 e seu grafo apresentados na Figura 11.2, que calcula o mínimo e o máximo dos elementos de um vetor  $A$ , determinado pelas variáveis de entrada `low`, `step`, `high`. O objetivo é gerar dados de teste para o caminho  $P = (1, 2, 3, 5, 6, 7, 2, 8)$ .

Para um conjunto inicial  $low = 39$ ,  $high = 93$ ,  $step = 12$ ,  $A[1] = 1$ ,  $A[2] = 2, \dots, A[100] = 100$ , o subcaminho  $(1, 2, 3)$  de  $P$  é executado. O predicado do nó 3, dado por  $max < A[i]$  foi incorretamente avaliado, pois o ramo  $(3, 4)$  foi executado. O primeiro subobjetivo é encontrar uma variável  $x$  tal que  $F_3(x) \leq 0$ , ou seja, fazer com que  $A[i] - max \leq 0$  e que o ramo  $(3, 5)$  seja executado. As variáveis seguem a ordem:  $A[1], A[2], \dots, A[100], high, step, low$ . Se  $A[1]$  é incrementado ou decrementado de 1,  $F$  não se altera. Movimentos exploratórios são realizados para  $A[2], \dots, A[38]$  que não afetam o valor de  $F$ . Quando  $A[39]$

```

1   void max (int low, int step, int high) Programa 11.6
2   {
3       int A[100], i, max, min;
4       /* no' 1 */ leitura (A);
5       /* no' 1 */ min = A [low];
6       /* no' 1 */ max = A [low];
7       /* no' 1 */ i = low + step;
8       /* no' 2 */ while (i < high) {
9           /* no' 3 */ if (max < A[i])
10          /* no' 4 */     max = A [i];
11          /* no' 5 */     if (min > A[i])
12          /* no' 6 */     min = A[i];
13          /* no' 7 */     i = i + step; }
14          /* no' 8 */     printf ("%d, %d",min,max);
15   }

```

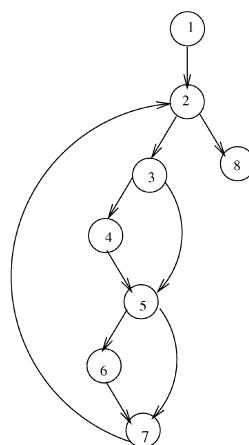


Figura 11.2 – Tratamento de vetores na execução dinâmica.

é incrementado, o valor de  $F$  diminui. Essa é uma boa direção, e a mudança de padrão é efetuada:  $A[39] = 139$ . Para essa entrada, o ramo (3,5) é executado. Outros subobjetivos serão tomados até que a solução seja encontrada.

Em muitos casos, avaliar a função associada ao ramo consome muito tempo e esforço. O ideal seria minimizar o número de avaliações. Korel [220] propõe utilizar análise de fluxo de dados dinâmica para escolher qual variável de entrada será considerada primeiro. Se a variável  $A[39]$  fosse escolhida em primeiro lugar, a solução seria encontrada rapidamente; no entanto, as variáveis foram selecionadas segundo uma dada seqüência.

A análise de fluxo de dados considera definições e usos de variáveis. A cada nó  $i$  do grafo de fluxo de controle são associados o conjunto de variáveis definidas em  $i$  e o conjunto de variáveis usadas em  $i$ . Um nó  $n_j$  influencia diretamente um nó  $n_k$  por uma variável  $x$ , se  $x$  foi definida em  $n_i$  e não existe redefinição de  $x$  em nenhum outro nó no caminho de  $j$  para  $k$ . Uma variável definida em um nó  $l$  influencia diretamente um nó  $k$  se ela não foi definida em nenhum outro nó no caminho de  $j$  para  $k$ . Uma variável de entrada  $x$ , definida no nó  $n_1$  influencia um nó  $n_k$  se no caminho  $(n_1, n_2, \dots, n_k)$  de  $n_1$  para  $n_k$  todos os nós  $n_i$  influenciam diretamente o nó  $n_{i+1}$  por alguma variável  $x_i$ , para  $1 \leq i < k$ . Essas relações são representadas em um grafo de influências. A Figura 11.3 representa como as variáveis de entrada do programa da Figura 11.2, definidas no nó 1, influenciam os valores de  $i$  e  $max$  e, consequentemente, o predicado do nó 3.

Para tornar o processo de resolução de subobjetivos mais eficiente, são escolhidas primeiro as variáveis de entrada que influenciam a função associada ao ramo. Outra questão a ser considerada é o fator de risco, dado pelo número de predicados que a variável influencia no caminho correto executado até onde ocorre a violação do subobjetivo. Entre duas variáveis  $x_1$  e  $x_2$  que influenciam uma função deve-se escolher a que tem menor fator de risco, pois a atribuição de um novo valor a uma variável de baixo fator de risco implica uma menor probabilidade de ocasionar violações em subobjetivos anteriores.

Um cuidado especial deve ser tomado com índices de vetores. Trocas nessas variáveis poderão ocasionar mudanças no conjunto de variáveis que influenciam um dado subobjetivo.

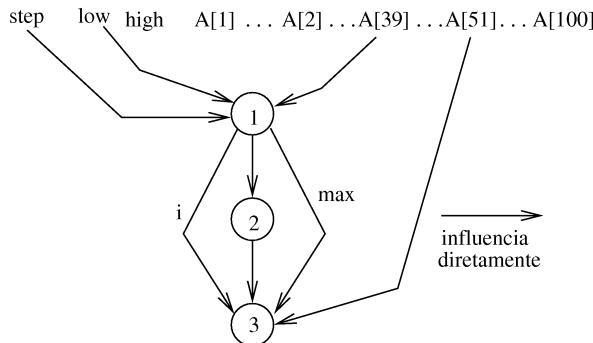


Figura 11.3 – Grafo de influências para o programa da Figura 11.2.

Toda vez que uma variável de entrada receber um valor, um novo conjunto de variáveis que influenciam cada subobjetivo e, consequentemente, novos fatores de risco deve ser derivado.

A técnica dinâmica também permite tratar estruturas de dados dinâmicas, tais como registros e apontadores. Cada campo do registro tem um nome distinto e pode ser tratado como uma variável separada. Entretanto, apontadores representam duas variáveis: o apontador em si e a variável para a qual ele aponta.

Toda estrutura dinâmica criada durante a execução do programa é tratada como uma variável separada. É mantida uma lista dessas variáveis criadas, que serão identificadas por nomes únicos. O tipo de estrutura a ser criada está na declaração do comando.

Dois tipos de subobjetivos são utilizados: o aritmético e o de apontador. O aritmético é resolvido como descrito anteriormente; o de apontador será resolvido utilizando-se um método de pesquisa baseado em análise de fluxo de dados e *backtracking*.

A análise de fluxo de dados dinâmica é utilizada para determinar as variáveis que são apontadores e influenciam esse subobjetivo. Não existe em um subobjetivo do tipo apontador uma função a ser minimizada. Atribuições sistemáticas são feitas às variáveis de entrada até que o subobjetivo seja resolvido. Passa-se então para os próximos subobjetivos. Se algum deles não puder ser satisfeito, um *backtracking* é realizado e novas soluções para os subobjetivos anteriores são derivadas.

## 11.4 Geração com algoritmos genéticos

Alguns autores propuseram o uso de técnicas de computação evolutiva para lidar com as limitações da atividade de geração de dados de teste, dando origem a uma nova área de pesquisa chamada Teste Evolucionário [425]. A maioria desses trabalhos utiliza algoritmos genéticos.

Algoritmos genéticos são métodos de busca e otimização que, aplicando as idéias da Teoria da Evolução de Darwin [104], simulam os processos naturais de evolução; os indivíduos mais adaptados de uma população têm maior probabilidade de sobreviver e, por isso, de passar aos seus descendentes seu material genético. Para utilizar algoritmos genéticos é

necessário: representar o espaço de busca do problema e suas possíveis soluções de forma computacional e encontrar uma função de avaliação (*fitness*) que decida o quão boa é uma dada solução. O algoritmo genético funciona como a seguir: primeiramente, uma população de soluções para determinado problema é gerada. Inicia-se, então, um ciclo de evolução que se repete até a solução desejada ser encontrada, ou até um número máximo de gerações ser alcançado. A cada iteração desse ciclo, novos indivíduos são gerados para compor a nova população e operadores genéticos, tais como seleção, mutação, reprodução e cruzamento, são aplicados.

Na literatura, muitos trabalhos utilizam algoritmos genéticos para a geração de dados de teste. A maioria deles visa à geração de dados de teste para critérios baseados em fluxo de controle [201, 282, 325, 345, 397, 426, 446]. Alguns desejam satisfazer critérios baseados em fluxo de controle e em fluxo de dados [53, 137, 425]. Outros focalizam o critério Análise de Mutantes [42, 80, 137]. Entretanto, esses trabalhos podem ser classificados em duas categorias, dependendo da função de avaliação utilizada. Na primeira categoria, a avaliação é realizada com base na cobertura de cada dado de teste [137]. Na segunda, a função é especificada de acordo com um objetivo, que é, geralmente, a cobertura de um elemento particular [282, 425].

#### 11.4.1 Função de avaliação baseada na cobertura

Na maioria dos trabalhos dessa categoria [137, 201, 345] os indivíduos na população codificam valores possíveis para as variáveis de entrada do programa em teste. No trabalho de Ferreira e Vergilio [137], os indivíduos são representados por uma concatenação de blocos; cada bloco está associado a uma variável de entrada do programa. O formato do bloco varia de acordo com o tipo da variável, e diferentes tipos de valores podem ser gerados. Uma população inicial costuma ser gerada aleatoriamente, mas também pode ser fornecida pelo testador, que poderá utilizar um conjunto de teste existente. A função de avaliação utilizada é dada pela cobertura do critério a ser satisfeito e pelo conjunto de elementos cobertos por indivíduo na população, sendo que cada indivíduo representa um dado de teste. Esse conjunto pode ser representado por uma matriz tal como a da Figura 11.4, na qual o valor X representa que o indivíduo *i* cobre o elemento requerido *j*. Operadores genéticos são aplicados até que a cobertura desejada seja obtida ou que um número estipulado de gerações seja alcançado.

|                |  | Elementos Requeridos |   |   |   |   |   |   |   |   |   |   |
|----------------|--|----------------------|---|---|---|---|---|---|---|---|---|---|
|                |  | X                    | X | - | - | - | - | - | X | - | X | X |
| Dados de Teste |  | X                    | - | - | - | - | X | X | X | X | X | X |
|                |  | X                    | X | - | - | - | X | - | - | - | - | - |
|                |  |                      |   |   |   |   |   |   |   |   |   |   |

Figura 11.4 – Matriz utilizada para avaliar os indivíduos.

Para se alcançar um melhor resultado com esse tipo de função, Ferreira e Vergilio [137] introduziram técnicas de hibridização, tais como o uso de listas *tabu* e elitismo, para garantir que alguns indivíduos que cobrem poucos elementos requeridos sejam mantidos na população; para isso, basta contribuírem para um aumento da cobertura global do critério.

### 11.4.2 Função de avaliação orientada a um objetivo

Nessa categoria, os trabalhos consideram a execução dinâmica de programas e um objetivo que orienta a função de avaliação, o de executar um particular caminho do programa, para cobrir elementos requeridos por um dado critério.

Na abordagem relatada por Michael et al. [282], esse objetivo, por sua vez, pode ser dividido em subobjetivos relacionados à satisfação dos predicados associados a cada aresta que compõe o caminho pretendido. O algoritmo genético é utilizado como uma técnica de minimização de funções (ver Seção 11.3). Um indivíduo na população representa um dado de teste. Sua função de avaliação é calculada, verificando o quão bem o predicado é satisfeito.

Em outro trabalho, Bueno e Jino [53] utilizam uma função de avaliação que diz o quanto o caminho executado pelo indivíduo está próximo do caminho pretendido. Uma medida do número de nós presentes em ambos os caminhos é calculada e utilizada pela função de avaliação.

## 11.5 Geração de dados sensíveis a defeitos

As técnicas de geração de dados sensíveis a defeitos procuram selecionar pontos do domínio que estão relacionados a certos tipos de defeitos e oferecem estratégias de geração de dados de teste que terão alta probabilidade de revelá-los. Essas estratégias foram inicialmente definidas de maneira *ad-hoc*. Posteriormente, foram definidas para serem utilizadas com determinados critérios de teste. Muitas podem ser utilizadas conjuntamente com a execução simbólica e oferecem critérios para a escolha dos valores que satisfazem a condição do caminho. As principais estratégias de geração de dados de teste sensíveis a defeitos são apresentadas nas próximas subseções.

### 11.5.1 Teste de domínios

A técnica chamada teste de domínios foi originalmente desenvolvida por White e Cohen [433], com o propósito de selecionar dados de teste para um conjunto de caminhos de programas, mas o teste de domínios não especifica como os caminhos são selecionados. Nesse sentido, ela deve ser usada conjuntamente com critérios baseados em caminhos (estruturais).

O objetivo é detectar erros de domínio, ou seja, erros que atribuem um domínio incorreto a um dado caminho, selecionando dados de teste no limite do domínio do caminho ou próximos dele. Esse é o fundamento da técnica de teste funcional conhecida como Análise do Valor Limite (Capítulo 2), que diz que um grande número de erros tende a se concentrar em limites dados pelas condições existentes no programa.

Clarke, Hassel e Richardson [84], bem como Chou e Du [78], descreveram alguns problemas encontrados ao se aplicar a técnica proposta por White e Cohen e propuseram alternativas. Zeil et al. [453] propuseram uma extensão para detectar erros lineares em funções de predicados não lineares.

Primeiramente será dada uma descrição da terminologia e das suposições adotadas por White e Cohen [433] para definir a técnica. A técnica original é então apresentada e possíveis extensões são mencionadas.

## Terminologia básica

A técnica analisa os limites do domínio de um caminho para gerar os dados de teste que o executam. O domínio do caminho é definido como o conjunto de dados de entrada que satisfazem a “condição do caminho”. Os limites do domínio do caminho são dados pelas condições associadas às arestas que compõem o caminho. A cada condição corresponde um predicado, que é uma combinação lógica de expressões relacionais.

White e Cohen assumem que os predicados encontrados são simples. Interpretações de predicados em condições de caminhos determinam os limites do domínio do caminho dados por uma borda. Cada borda poderá ser aberta ou fechada, dependendo do operador relacional do predicado associado (aberta:  $>$ ,  $<$ ,  $\neq$  e fechada:  $\leq$ ,  $\geq$ ,  $=$ ).

O número de predicados no caminho é um limite superior do número de bordas do domínio, uma vez que alguns predicados não correspondem necessariamente a uma borda. É o caso de interpretações de predicados independentes das variáveis de entrada e predicados redundantes.

A forma geral de uma interpretação de um predicado linear simples é:  $a_1x_1 + a_2x_2 + \dots + a_nx_n \text{ rop } k$ , em que *rop* é um operador relacional, cada  $x_i$  é uma variável de entrada e cada  $a_i$  e  $k$  são constantes. A borda do domínio é dada pela igualdade  $a_1x_1 + a_2x_2 + \dots + a_nx_n = k$ .

Se as interpretações de predicados forem lineares, os limites são bordas lineares e podem ser de três tipos básicos: igualdade, inequações e desigualdades.

Para uma condição de caminho composta de interpretações lineares que são igualdades, ou inequações, o domínio do caminho é dado por um poliedro convexo. Se uma ou mais desigualdades estiverem presentes, o domínio do caminho consiste na união de um conjunto de poliedros convexos disjuntos.

## Suposições para definir o teste de domínios

A técnica de teste de domínios está baseada na análise geométrica dos limites do domínio. Tem como objetivo detectar erros nas bordas, dados por pequenas diferenças entre a borda correta e a borda do programa em teste (borda dada), embora o teste de domínios não exija que se tenha conhecimento do programa correto.

White e Cohen [433] definiram a técnica para programas que não referenciam apontadores, vetores ou matrizes e que não possuem chamadas de procedimento e/ou funções. Além disso, para simplificar, as seguintes suposições foram feitas:

- correção coincidente não ocorre;
- erro de caminho ausente não ocorre para o caminho em teste;
- cada borda corresponde a um predicado simples (aquele que contém somente operadores relacionais);
- domínios adjacentes computam funções diferentes;
- a borda dada é linear e, se estiver incorreta, a borda correta também é linear;
- o espaço de entrada é contínuo.

A técnica de teste de domínios seleciona dois tipos de pontos de teste. Pontos *on* pertencem à borda dada e pontos *off* ficam a uma distância  $\epsilon$  muito pequena e devem pertencer ao domínio que não contém a borda. Se a borda é fechada, os pontos *on* pertencem ao domínio do caminho que está sendo testado e os pontos *off* pertencem a algum domínio adjacente. Se a borda é aberta, os pontos *on* pertencem a algum domínio adjacente, enquanto os pontos *off* pertencem ao domínio em teste.

### A técnica $N \times 1$

Para programas com interpretações de predicados que resultam em inequações simples e lineares com relação às variáveis de entrada do programa, o domínio de um caminho será um poliedro convexo de dimensão  $N$ , tal que  $N$  é o número de entradas do programa. Em  $N$  dimensões cada borda é dada por um hiperplano com  $(N - 1)$  dimensões. Para testar a borda dada, utiliza-se a propriedade de que um hiperplano com  $(N - 1)$  dimensões é dado por  $N$  pontos linearmente independentes. Por isso, White e Cohen propuseram que sejam escolhidos  $N$  pontos *on* e um ponto *off* para cada borda do domínio do caminho. Por selecionar  $N$  pontos *on* e um ponto *off* a técnica é chamada de técnica  $N \times 1$ .

No teste em duas dimensões, programas com duas variáveis de entrada, cada borda do domínio do caminho é um segmento de reta determinado por dois pontos. Por isso, são selecionados dois pontos *on* e um ponto *off*. Essa técnica é referenciada como técnica  $2 \times 1$ .

A Figura 11.5 apresenta o domínio de entrada para o caminho  $(1, 2, 3, 4, 3, 5, 6)$  do Programa 11.7. Aplicando-se a técnica  $2 \times 1$  na borda que corresponde ao último *if* do programa,  $P = (0, 1.001)$  e  $Q = (1, 1.001)$ , poderiam ser escolhidos como pontos *on* e  $V = (0.5, 1.001 + \epsilon)$ , tal que  $\epsilon$  é o menor valor possível, como ponto *off*.

```

1   void ex (double x, double y)
2   {
3       double d, z;
4       /* no' 1 */      if (x > 0) {
5           /* no' 2 */      z = y - x;
6           /* no' 3 */      d = 0;
7           /* no' 4 */      while (z < d)
8               /* no' 5 */      d = d + 1;
9           /* no' 6 */      if (y <= 1.001 * d)
10          ....
11      }
12  }
    
```

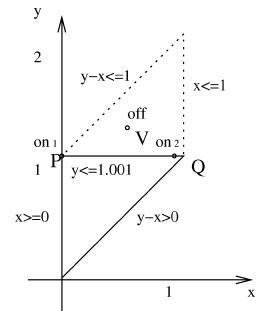


Figura 11.5 – Ilustrando a técnica  $2 \times 1$ .

A Figura 11.6 ilustra a aplicação da técnica para um domínio com três dimensões. Uma recomendação feita por Chou e Du [78] e Clarke et al. [84] é que em duas dimensões os pontos *on* devem ser escolhidos tão próximos quanto possível do final da borda, e os pontos *off* próximos ao seu centro para que um número maior de defeitos seja revelado. Em três dimensões, combinação formada pelos pontos *on* deve conter o centro da borda dada, e o ponto *off* deve pertencer ao centro do hiperplano paralelo à borda dada, que fica a uma distância  $\epsilon$  do lado aberto da borda.

Se o número de dimensões cresce, a aplicação da técnica e a visualização dos domínios podem ficar cada vez mais difíceis.

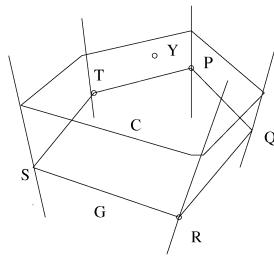


Figura 11.6 – Técnica  $N \times 1$  em três dimensões.

### As técnicas $N \times N$ e $V \times V$

A técnica  $N \times N$  sugere que  $N$  pontos *on* e  $N$  pontos *off* sejam selecionados. Quando aplicada em duas dimensões, os dois pontos *off* são escolhidos nos finais do segmento de linha paralelo à borda dada, como mostrado na Figura 11.7; os pontos *off* são  $U$  e  $V$ .  $P$  e  $Q$  são pontos *on*.

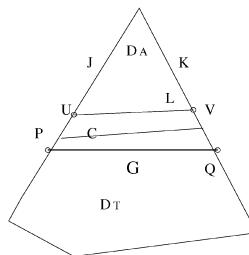


Figura 11.7 – Ilustrando a técnica  $N \times N$ .

Generalizando-se para outras dimensões, a técnica tem duas interpretações. A primeira é a técnica  $N \times N$ ; e a segunda, a estratégia  $V \times V$  (em duas dimensões, elas são idênticas).

A técnica  $N \times N$  interpreta o 2 como sendo a dimensão do domínio. Escolhe  $N$  pontos *on*, contendo o centro de  $G$  e linearmente independentes (portanto, sugere-se que eles estejam próximos aos vértices), e  $N$  pontos *off* que sejam linearmente independentes e formem um hiperplano  $L$  a uma distância  $\varepsilon$  de  $G$ . Sugere-se ainda que esses planos devem conter o centro de  $L$ .

Na Figura 11.8  $S, P$  e  $R$  poderiam ser selecionados como pontos *on*. Se as extensões adjacentes são perpendiculares a  $G$ , então  $L$  e  $G$  são congruentes.  $X, V$  e  $U$  poderiam ter sido escolhidos como pontos *off*.

A utilização da técnica  $N \times N$  tem a vantagem de não requerer que se determinem os vértices da borda dada. Por outro lado, mudanças nos vértices das bordas podem ocorrer sem que isso implique mudanças nos pontos *on* e *off* escolhidos (exceto para  $N = 2$ ). Além

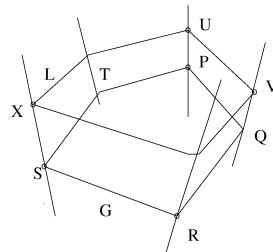


Figura 11.8 – Técnica  $N \times N$  em três dimensões.

disso, quando o número de vértices da borda é muito maior que a dimensão do domínio, alguns vértices consecutivos não serão testados; isso pode permitir que a borda correta não intercepte alguma extensão lateral da borda dada.

Esse problema é resolvido pela técnica  $V \times V$ , em que  $V$  é o número de vértices da borda dada. Se existem  $V$  vértices,  $V$  pontos *on* e  $V$  pontos *off*, próximos aos vértices ou nos vértices do hiperplano  $L$  paralelo à borda dada, são selecionados.

A técnica  $V \times V$  é influenciada pelo domínio; qualquer mudança no domínio implica mudanças nos pontos *off* e *on* escolhidos.

A complexidade das três técnicas é dada pelo número de pontos requeridos. A técnica  $N \times 1$  requer  $(N + 1) * B$  pontos, a técnica  $N \times N$  requer  $(2 * N) * B$  pontos, em que  $B$  é o número de bordas fechadas do domínio do caminho. A técnica  $V \times V$  requer para o domínio de um caminho,  $\sum_{i=1}^B 2 * V_i$ , em que  $V_i$  é o número de vértices da  $i$ -ésima borda. Esse total pode crescer rapidamente.

Outro fator importante a ser considerado é a facilidade para determinar os pontos. A técnica  $V \times V$  exige que todos os vértices sejam determinados e pode parecer que demandaria mais tempo. No entanto, para se garantir uma maior eficácia das técnicas  $N \times 1$  e  $N \times N$ , os vértices também precisam ser determinados para que pontos centrais possam ser escolhidos. Além disso, todas as combinações de vértices precisam ser consideradas para que a melhor possa ser escolhida; isso, em geral, requer tempo não polinomial. A técnica  $V \times V$ , segundo Clarke et al. [84], se mostra mais barata.

Zeil et al. [453] analisaram a capacidade das técnicas em detectar defeitos e sua relação com a confiabilidade. Os autores concluem que esse problema está na formulação das técnicas de teste de domínios e propõem uma nova abordagem que leva em consideração o compartilhamento de comandos por um número potencialmente infinito de caminhos, propondo o teste de domínios de subcaminhos que levam até uma dada interpretação de predicados. Essa nova visão do teste de domínios reduz em muito os custos computacionais. O número de pontos de teste é significativamente reduzido. Cada interpretação é testada só uma vez. Por exemplo, o predicho  $x \geq 0$  da Figura 11.5 pertence a todos os caminhos do programa e será testado só uma vez. Zeil et al. dizem que essa visão resolve os problemas de dependências das bordas adjacentes e não provoca perda da habilidade em detectar erros de bordas do teste de domínios.

Zeil et al. [453] afirmam que as três técnicas apresentadas podem ser consideradas efetivas para detectar erros em bordas. A técnica  $N \times N$  leva alguma vantagem sobre a  $N \times 1$ , por

ser melhor na maioria dos casos. No entanto, a técnica  $V \times V$  raramente seria melhor que a  $N \times N$ , visto que os pontos por ela escolhidos formam um superconjunto dos escolhidos pela  $N \times N$ . Entretanto, a técnica  $N \times 1$  é mais prática e fácil de aplicar.

As suposições feitas para se definirem as técnicas de teste de domínios podem ser suprimidas ou relaxadas, desde que mudanças apropriadas nas técnicas sejam efetuadas. Isso permite que o teste de domínio seja aplicado a uma classe maior de programas. No entanto, para ele se tornar efetivo, é necessário que ele seja aplicado com outros métodos de teste.

### 11.5.2 Teste baseado em restrições

A técnica de teste baseado em restrições foi proposta por DeMillo e Offutt em 1991 [116]. A técnica tem como objetivo auxiliar a geração de dados de teste adequados ao teste de mutação. Ela usa restrições algébricas projetadas para detectar um tipo particular de defeito no programa.

O teste baseado em restrições usa os conceitos de análise de mutantes para gerar dados de testes; os dados são projetados para matar os mutantes, ou seja, para detectar defeitos descritos pelos operadores de mutação. Como apresentado no Capítulo 5, um mutante difere do programa original por uma pequena diferença sintática em um comando  $S$ . Para matar um mutante é necessário que: 1) o comando  $S$  seja executado; 2) imediatamente após a execução de  $S$ , o estado do programa original seja diferente do estado do programa mutante; 3) essa diferença seja propagada até que comandos de saída sejam executados e resultados diferentes sejam produzidos, possibilitando assim a revelação do defeito.

Essas três condições são referenciadas pelos termos: alcançabilidade, necessidade e suficiência, que serão discutidos a seguir.

#### Alcançabilidade

O problema de gerar dados de teste que garantam alcançar um dado comando  $S$  é em geral indecidível. Expressões de caminhos (execução simbólica) descrevem restrições que precisam ser satisfeitas para que um comando seja alcançado. Cada expressão corresponde a um caminho diferente até  $S$ . Desde que o interesse seja apenas alcançar o comando, o conjunto de todos os caminhos que alcançam  $S$  é dado por uma seqüência de restrições conectadas pelo operador “ou”.

Um problema para determinar essas restrições são os laços. Eles podem gerar um número desconhecido ou ilimitado de caminhos. Se a execução do laço for relevante para alcançar o comando e se isso puder ser identificado, deve-se gerar uma restrição que garanta a execução do laço pelo menos uma vez. Caso a execução do laço seja irrelevante para o comando, o laço poderá ser executado zero ou mais vezes. O ideal é que não se exclua qualquer caminho que possa alcançar o comando.

#### Necessidade

A condição de necessidade é a palavra-chave do teste baseado em restrições. Para se matar um mutante é necessário que, imediatamente após a execução de um comando mutado, o

estado do programa mutante seja diferente do estado do programa original. O estado do programa consiste nos valores das variáveis do programa e variáveis internas. O mutante difere do programa original apenas por uma modificação simples em um comando. Se, após a execução desse comando, os estados dos programas mutante e original forem iguais, as saídas produzidas também serão e o mutante não será morto.

A condição de necessidade descreve uma restrição que deve ser satisfeita pelo dado de teste para que estados distintos nos programas mutante e original sejam produzidos.

## Suficiência

Satisfazer a condição de necessidade é necessário para se matar um mutante, mas não é suficiente. Para que o mutante morra, uma saída incorreta deverá ser produzida. O estado final do programa mutante tem de ser diferente do estado final do programa original. Derivar dados de teste que satisfaçam à condição de suficiência é impraticável. Em geral, não se conseguem determinar completamente condições de suficiência [116].

O Programa 11.8 calcula o máximo de dois números  $m$  e  $n$ . Foram criados dois mutantes do programa original correspondentes às duas mutações especificadas entre comentários no programa. Para se alcançar o primeiro comando modificado (linha 3), não é necessária nenhuma restrição; portanto, a condição de alcançabilidade é sempre satisfeita. Para se alcançar o segundo comando (linha 5) é necessário ( $n > m$ ). As condições de necessidade para que estados diferentes sejam produzidos após a execução dos comandos modificados são, respectivamente,  $m \neq \text{abs}(m)$  e  $n \neq 6$ . No entanto, a primeira condição de necessidade não é suficiente, pois, se ( $n > m$ ), o resultado correto ainda pode ser produzido. Portanto, a condição de suficiência para o primeiro mutante é ( $n \leq m$ ).

---

Programa 11.8

---

```
1 void maximo(double m, double n)
2 {
3     max = m; /* mutante 1: modificado para max = abs(m) */
4     if (n > max)
5         max = n; /* mutante 2: modificado para max = 6 */
6     return max;
7 }
```

---

Muitos pesquisadores têm considerado o problema de suficiência [50, 187, 290], relacionando-o ao termo correção coincidente. Determinar condições de suficiência é então uma questão indecível. DeMillo [116] considera um mutante morto quando a saída obtida diverge da saída do programa original. Ele considera que se um dado de teste satisfez as condições de alcançabilidade e de necessidade, um estado incorreto foi produzido, e saídas iguais foram obtidas, três situações poderiam ter ocorrido:

- O programa reconhece o seu estado de erro e se recupera. Trata-se de uma situação de tolerância a falhas. Embora útil, na maioria dos casos isso não ocorre.
- O estado incorreto é irrelevante para alterar o estado final. Trata-se de um mutante equivalente.
- Embora o dado de teste tenha produzido um estado diferente, não foi o suficiente para mudar o estado final.

Em experimentos conduzidos por DeMillo e Offutt [116], notou-se que a terceira situação raramente acontece na prática. A probabilidade de se satisfazer a condição de suficiência, dado que as condições de necessidade e alcançabilidade foram satisfeitas, é muito grande. Em 90% dos mutantes analisados, quando predicados não estavam envolvidos, a produção de um estado intermediário incorreto garantiu um estado final incorreto e, consequentemente, a morte do mutante. A posição adotada por DeMillo e Offutt [116] foi então assumir que a satisfação das condições de necessidade e alcançabilidade implica a satisfação da condição de suficiência.

Por outro lado, apenas 60% dos mutantes que envolviam mutações em predicados foram mortos. Em muitos casos, embora um efeito no estado do mutante fosse registrado, o resultado da avaliação de um predicado continuava o mesmo que o do programa original. Um exemplo desse problema é apresentado na Tabela 11.1. A condição de necessidade gerada para provocar a diferença de estado é  $I \neq 3$ . Observe que o dado de teste  $I = 7, J = 9, K = 7$  satisfaz essa restrição, mas o predicado continua sendo avaliado como verdadeiro, como era esperado.

Tabela 11.1 – Restrições de predicados

|                          |                                    |
|--------------------------|------------------------------------|
| Programa Original        | <code>if ((i + k) &gt;= j))</code> |
| Mutante                  | <code>if ((3 + k) &gt;= j))</code> |
| Restrição de Necessidade | $i \neq 3$                         |
| Restrição de Predicado   | $(I + K > J) \neq (3 + K > J)$     |

Para o problema de suficiência, DeMillo e Offutt propõem restrições de predicados, que são extensões das restrições de necessidade e asseguram diferenças no resultado da avaliação de predicados. Para o exemplo anterior a restrição de predicado é dada por  $(I + K > J) \neq (3 + K > J)$ . O dado de teste  $I = 7, J = 9, K = 7$ , que não mata o mutante, também não satisfaz a restrição:  $(7 + 7 > 9) \neq (3 + 7 > 9) = False$ . Contudo, o dado de teste  $I = 7, J = 10, K = 7$ , que satisfaz a restrição  $(7 + 7 > 10) \neq (3 + 7 > 10)$ , provoca diferença no resultado da avaliação, revelando o defeito e matando o mutante.

Um conjunto de ferramentas chamado Godzilla foi implementado com o objetivo de gerar dados de teste automaticamente. Ele gera e resolve restrições para detectar erros descritos pelos mutantes gerados pelo sistema Mothra [114].

Godzilla representa uma restrição por um par de expressões algébricas relacionadas por um operador condicional ( $>$ ,  $<$ ,  $\leq$ ,  $\geq$ ,  $=$ ). Cada expressão é obtida diretamente do programa e é composta por variáveis, parênteses e operadores de linguagens de programação. Cada restrição pode ser avaliada como verdadeira ou falsa. Uma cláusula é dada por uma lista de restrições conectadas por operadores *and* ou *or*; além disso, uma restrição pode ser alterada pelo operador *not*. Godzilla utiliza somente cláusulas em sua forma normal disjuntiva, ou seja, formadas por operadores *or*, e valores simbólicos para as variáveis. Isso facilita a geração de dados de teste, a restrição disjuntiva representa um único caminho e somente uma cláusula disjuntiva precisa ser satisfeita pelo dado de teste.

A ferramenta Godzilla é composta de três módulos principais que se comunicam por meio de arquivos. O primeiro módulo cria restrições para cada comando do programa original, dadas pelas expressões dos caminhos que alcançam o comando dado. O segundo módulo

constrói as condições de predicados descritos, que são armazenadas em tabelas e passadas ao terceiro módulo. O terceiro módulo utiliza as tabelas geradas e a expressão do caminho para gerar o dado de teste. Se o dado de teste não puder ser gerado porque não foi possível resolver as restrições, isso é notificado ao testador que poderá decidir a questão. Godzilla também permite reduzir o número de dados de teste, verificando se, ao se satisfazer uma condição, outras também o serão. Assim, um dado de teste pode ser utilizado para matar mais de um mutante.

### 11.5.3 Teste baseado em predicados

Estratégias de teste referenciadas como teste baseado em predicados requerem, em geral, que cada predicado, ou condição, do programa seja testado.

Os predicados no programa dividem o domínio de entrada do programa em partições e definem os caminhos do programa. Predicados são classificados em predicados simples e compostos.

A motivação para que técnicas de teste de predicados fossem introduzidas é a determinação de defeitos em predicados.

As técnicas baseadas em predicados foram classificadas por Tai [382] e se dividem em dois grupos: as que testam predicados simples e as que testam predicados compostos. Tai também discute aspectos de eficácia de tais técnicas em determinar defeitos de predicados. Técnicas que consideram apenas predicados simples podem deixar de detectar erros de predicados em expressões booleanas. As técnicas que consideram predicados compostos propõem testes exaustivos para todas as combinações, o que pode torná-las viáveis somente se o predicado contiver um número pequeno de operadores. O problema com predicados compostos é que, se o teste exaustivo não for realizado, poderá acontecer de, embora os predicados individuais sejam avaliados incorretamente, o resultado da avaliação de toda a expressão ser coincidentemente correto. Por exemplo, para o predicado composto  $P = (E_1 = E_2) \& \& (E_3 < E_4)$ , os seguintes testes são requeridos:

$$t_1 : E_1 = E_2 \text{ e } E_3 = E_4$$

$$t_2 : E_1 > E_2 \text{ e } E_3 > E_4$$

$$t_3 : E_1 < E_2 \text{ e } E_3 < E_4$$

Note que, para esse conjunto de testes, todas as possíveis situações para os dois predicados simples foram consideradas. Entretanto, o conjunto não distingue o predicado  $P$  do predicado  $R = (E_1 <> E_2) \& \& (E_3 = E_4)$ .

Essa situação é semelhante à que acontece no teste baseado em restrições, descrita na subseção anterior. Um estado incorreto é produzido mas não é suficiente para que o defeito seja revelado. DeMillo [116] sugeriu o uso de uma restrição adicional chamada restrição de predicado, que é um tipo de condição de suficiência. Tai [382] propõe duas técnicas de teste para predicados compostos. As técnicas são baseadas em erros, pois requerem que dados de teste sejam executados para satisfazer um conjunto de restrições. O conjunto de restrições é projetado para garantir que defeitos em operadores booleanos e defeitos em operadores relacionais sejam detectados, com a suposição de que não existam defeitos de outros tipos. A grande vantagem é que um conjunto de restrições que descrevem vários tipos de erros é gerado de uma só vez. Um número grande de mutantes é morto se as restrições forem

satisfitas; no caso a suposição é satisfeita porque cada mutante difere do original por uma mutação simples.

## Teste baseado em restrições de predicados

Tai [382] propôs duas técnicas para detectar defeitos em predicados compostos. A primeira, chamada teste do operador booleano, ou técnica BOR (*Boolean Operator Testing*), garante a detecção de defeitos em operadores booleanos. Tai afirma que se uma técnica é efetiva em determinar defeitos em operadores booleanos, ela também o será em determinar defeitos em expressões booleanas.

A segunda técnica, chamada teste dos operadores booleano e relacional ou técnica BRO (*Boolean and Relational Operator Testing*), garante a detecção de defeitos em operadores booleanos e relacionais.

Tai utiliza a notação a seguir para definir as restrições que serão requeridas pelas técnicas BRO e BOR.

Para uma variável booleana  $B$ :

- $t$ : denota que  $B$  vale *true*;
- $f$ : denota que  $B$  vale *false*;
- $*$ : denota que não há restrições para o valor de  $B$ .

Para uma expressão relacional ( $E_1 \text{ rop } E_2$ )

- $t$ : denota valor *true* para a expressão;
- $f$ : denota valor *false* para a expressão;
- $>$ : denota que  $(E_1 \text{ rop } E_2) > 0$ ;
- $<$ : denota que  $(E_1 \text{ rop } E_2) < 0$ ;
- $=$ : denota que  $(E_1 \text{ rop } E_2) = 0$ ;
- $+ε$ : denota que  $0 \leq (E_1 \text{ rop } E_2) \leq ε$ ;
- $-ε$ : denota que  $-ε \leq (E_1 \text{ rop } E_2) \leq 0$ ;
- $*$ : denota que não há restrições para o valor da expressão relacional.

Dado o predicado  $C = (E_1 \geq E_2) \parallel !(E_3 > E_4)$ , uma restrição para  $C$  é dada por uma lista de elementos  $l = (v_1, v_2, \dots, v_n)$  que denota os valores para as variáveis booleanas ou expressões relacionais. A restrição dada pela lista  $X = \{(=, <)\}$  requer um teste, fazendo  $E_1 = E_2$  e  $E_3 < E_4$ . Note que o operador “!” não afeta o requisito  $<$ . O valor produzido por  $C$  quando satisfeita a restrição  $X$  é dado por  $C(X)$ . O conjunto de todas as restrições para um predicado  $C$  é dado por  $S$ .  $S$  é dividido em dois conjuntos. O primeiro conjunto  $S_t(C)$

composto por todas as restrições  $X$  em que  $C(X) = \text{true}$ , e o segundo conjunto  $S_f(C)$  composto por todas as restrições  $X$  em que  $C(X) = \text{false}$ . A concatenação de duas restrições (listas)  $l_1$  e  $l_2$  é denotada por  $(l_1, l_2)$ .

Sejam  $A$  e  $B$  dois conjuntos de restrições.  $A \% B$  denota o conjunto mínimo de elementos  $(u, v)$ , tais que  $u(v) \in A(B)$  e cada elemento de  $A(B)$  aparece como  $u(v)$  pelo menos uma vez. Se  $A = \{(=), (>)\}$  e  $B = \{(<), (>)\}$ ,  $A \% B$  tem dois valores possíveis:

1.  $\{(=, <), (>, <)\}$ ; ou
2.  $\{(=, >), (>, <)\}$ .

Se  $A = \{(=), (>)\}$  e  $B = \{(<), (>), (=)\}$ ,  $A \% B$  tem seis valores possíveis:

1.  $\{(=, <), (>, >), (=, =)\}$ ;
2.  $\{(=, <), (>, >), (>, =)\}$ ;
3.  $\{(=, <), (=, >), (>, =)\}$ ;
4.  $\{(>, <), (=, >), (>, =)\}$ ;
5.  $\{(>, <), (=, >), (=, =)\}$ ; ou
6.  $\{(>, <), (>, >), (=, =)\}$ .

Ainda definem-se  $A \$ B$  como união de  $A$  e  $B$  e  $A * B$  como produto de  $A$  e  $B$ .

Para um predicado simples, composto somente de uma variável booleana ou de uma expressão relacional  $E_1 \text{ rop } E_2$ , os seguintes conjuntos de restrições são requeridos:

**BOR:**  $\{(t), (f)\}$

**BRO:**  $\{(>), (=), (<)\}$

Para um predicado com operadores booleanos *bop* ( $\parallel$  ou  $\&\&$ ) que contém pelo menos uma expressão booleana  $C_1$  *bop*  $C_2$ , os conjuntos de restrições são calculados da seguinte maneira:  $\text{BOR} = \text{BRO} = T(C) \$ F(C)$ , sendo que os conjuntos  $T(C)$  e  $F(C)$  são calculados diferentemente para cada operador booleano. Sejam  $S_1$  e  $S_2$  conjuntos de restrições para, respectivamente, os predicados  $C_1$  e  $C_2$ ,  $T(C)$  e  $F(C)$  são calculados de acordo com as seguintes fórmulas:

- $C_1 \parallel C_2$ .

$$F(C) = S_{1f} \% S_{2f}$$

$$T(C) = \{S_{1t} * \{f_2\}\} \$ \{\{f_1\} * S_{2t}\}$$

em que  $f_1 \in S_{1f}$  e  $f_2 \in S_{2f}$  e  $(f_1, f_2) \in F(C)$

- $C_1 \&\& C_2$ .

$$T(C) = S_{1t} \% S_{2t}$$

$$F(C) = \{S_{1f} * \{t_2\}\} \$ \{\{t_1\} * S_{2f}\}$$

em que  $t_1 \in S_{1t}$  e  $t_2 \in S_{2t}$  e  $(t_1, t_2) \in T(C)$

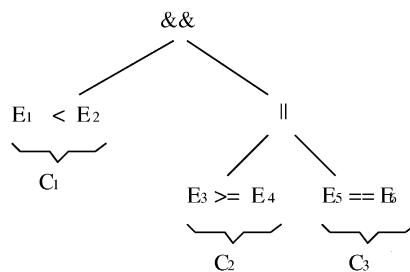


Figura 11.9 – Árvore sintática para um predicado.

Para o predicado  $P = ((E_1 < E_2) \&\& ((E_3 \geq E_4) \parallel (E_5 = E_6)))$ , a construção das restrições é feita de uma forma *bottom-up*, visitando-se a árvore sintática da Figura 11.9.

Ao aplicar as técnicas BOR e BRO geram-se conjuntos de restrições  $S_1, S_2$  e  $S_3$  correspondentes aos predicados  $C_1, C_2$  e  $C_3$ . Gera-se um conjunto  $S_4$  para a expressão  $C_1 \parallel C_3$  utilizando-se  $S_2$  e  $S_3$ . Gera-se, para a expressão  $C_1 \&\& (C_2 \parallel C_3)$ , um conjunto  $S_5$  a partir de  $S_1$  e  $S_4$ . A Figura 11.10 mostra o total de restrições requeridas.

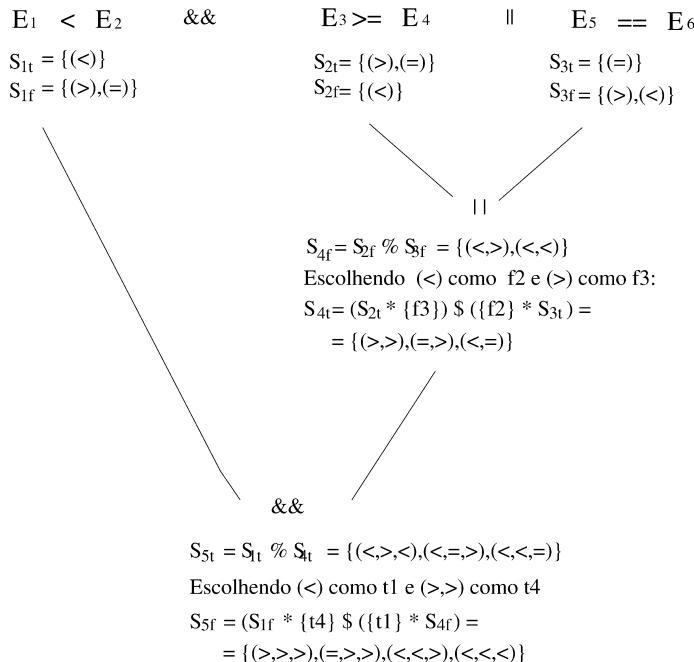


Figura 11.10 – Restrições BRO requeridas para o predicado da Figura 11.9.

As ferramentas BGG [382] e PredTool [358] geram conjuntos BOR e BRO para predicados de programas escritos, respectivamente em Pascal e C. Elas fornecem a cobertura das restrições para um dado conjunto de testes.

As restrições geradas pelas técnicas BRO e BOR podem ser utilizadas para gerar dados de teste adequados à análise de mutantes, como dito anteriormente. Além disso, as restrições poderão ser utilizadas tanto com técnicas de teste funcional quanto com técnicas estruturais. Tai indica duas abordagens para utilizá-las com a técnica de teste baseada em caminhos. A primeira é selecionar um conjunto de caminhos de tal maneira que cada restrição exigida seja coberta pelo menos uma vez. A outra, gerar um dado de teste para executar cada restrição de um caminho dado. Na prática, uma restrição também poderá ser não executável, caso no qual nem todas as restrições derivadas poderão ser satisfeitas e, consequentemente, não se poderá garantir que todos os erros de predicados foram detectados.

Com o objetivo de aumentar a eficácia das técnicas, Tai [382] propôs uma extensão chamada teste de expressões booleanas e relacionais com parâmetro  $\epsilon$ , ou técnica BRE( $\epsilon$ ) (*Boolean and Relational Expression Testing with parameter  $\epsilon$* ). A idéia é substituir ocorrências de  $>$  e  $<$  por  $+\epsilon$  e  $-\epsilon$ , respectivamente, sendo  $\epsilon$  um número muito pequeno, maior que 0. Essa técnica usa fundamentos da técnica teste de domínios e é mais eficaz para determinar defeitos em expressões relacionais que envolvem outros tipos de erros além dos detectados pelas técnicas BOR/BRO. BOR e BRO possuem maior capacidade de revelar defeitos de predicados (defeitos em expressões e operadores booleanas e relacionais); a técnica BRE revela defeitos semelhantes aos revelados pela técnica teste de domínios.

## 11.6 Considerações finais

Neste capítulo foram descritas as principais técnicas de geração de dados de teste que têm o objetivo de satisfazer critérios de teste e de revelar o maior número possível de defeitos.

Gerar dados de teste aleatoriamente não garante a seleção dos melhores pontos do domínio de entrada e também não auxilia a determinação de elementos não-executáveis, nada se aprendendo sobre o programa. A técnica, porém, é defendida por vários autores por ser menos custosa, prática e fácil de automatizar.

Apesar de bastante difundida, a execução simbólica apresenta muitas dificuldades de automatização. Ela é mais custosa do que a geração aleatória, mas comprovou ser mais eficaz em vários experimentos. Permite em alguns casos detectar caminhos não executáveis e, além disso, criar representações simbólicas da execução de um caminho que poderão ser confrontadas com representações simbólicas das saídas esperadas, facilitando a detecção de defeitos. A execução simbólica tem sido utilizada também em outras áreas da computação, tais como depuração [85].

A execução dinâmica tem por fim resolver os problemas da execução simbólica, utilizando técnicas de *backtracking*, análise de fluxo de dados e minimização de funções. O ideal é utilizar conjuntamente execução simbólica e execução dinâmica. Nesse sentido, técnicas de computação evolutiva, tais como Algoritmos Genéticos, podem ser utilizadas de maneira complementar para resolver as restrições estabelecidas com as execuções simbólica e dinâmica ou, ainda, para evoluir um determinado conjunto de dados de teste, melhorando a cobertura obtida com a geração aleatória.

As técnicas de geração de dados sensíveis a defeitos não oferecem uma maneira automática de gerar o dado, mas buscam gerar dados de teste com alta probabilidade de revelar defeitos. Vergilio et al. [407] apresentam resultados de um experimento com as técnicas de geração de dados sensíveis a defeitos.

A técnica teste de domínios deve ser utilizada com um critério de seleção de caminhos. Sugere-se um critério mais exigente porque assim será maior a probabilidade de o caminho que revela o defeito ser selecionado. Essa técnica também é a mais difícil de ser aplicada e automatizada; existem problemas com predicados compostos e não lineares, domínios discretos, etc.

A técnica de teste baseado em restrições tem sua eficácia dependente dos operadores de mutação utilizados, visto que não é possível descrever todo tipo de defeito e que as restrições de predicho são muito importantes por serem aproximações das restrições de suficiência.

As técnicas BOR/BRO são mais práticas e mais fáceis de ser automatizadas, mas são menos poderosas em termos de eficácia. BRE pode ser mais eficaz, mas, por outro lado, existe a dificuldade em se determinar o valor de  $\epsilon$ .

Vergilio et al. [406, 408] propõem critérios de teste, denominados Critérios Restritos, que têm o objetivo de combinar restrições de predicho, tais como as utilizadas no Teste Baseado em Restrições e pelas técnicas BRO/BOR e BRE, para gerar dados de teste adicionais para o teste estrutural. Os dados de teste devem executar o caminho dado para cobrir um determinado elemento requerido por um critério estrutural e satisfazer a restrição que descreve um defeito. Dessa maneira, aumenta-se a capacidade de revelar outros defeitos, geralmente não revelados por critérios estruturais.

Um problema inerente a todas as técnicas de geração de dados sensíveis a erros é que, no final, um conjunto de restrições é derivado. Programas que resolvam eficientemente essas restrições, utilizando execução simbólica ou dinâmica e técnicas de computação evolutiva, são necessários. Entretanto, nem sempre é possível encontrar essa solução; a indecidibilidade permanece como obstáculo à atividade de geração de dados de teste, não sendo possível sua completa automatização.

# Capítulo 12

## Depuração

*Marcos Lordello Chaim (EACH/USP)*

*José Carlos Maldonado (ICMC/USP)*

*Mario Jino (DCA/FEEC/UNICAMP)*

### 12.1 Introdução

A depuração de software é comumente definida como a tarefa de localização e remoção de defeitos [16]. Normalmente, ela é entendida como o corolário do teste bem-sucedido [10], ou seja, ela ocorre sempre que um defeito é revelado. No entanto, defeitos podem ser revelados em diferentes fases do ciclo de vida do software e a depuração possui características diferentes, dependendo da fase em que se encontra o software. Por isso, os processos de depuração que ocorrem durante a codificação, depois do teste e durante a manutenção, podem ser diferenciados.

A depuração durante a codificação é uma atividade complementar à de codificação. Já a depuração depois do teste é ativada pelo teste bem-sucedido, isto é, aquele que revela a presença de defeitos, podendo beneficiar-se da informação coletada nessa fase. A depuração durante a manutenção, por sua vez, ocorre por uma necessidade de manutenção no software, que pode ter sido causada, por exemplo, por um defeito revelado depois de liberado o software ou pela necessidade de acrescentar novas características a ele.

A relevância da atividade de depuração tem dirigido esforços de pesquisa em duas direções: no entendimento do processo de depuração e no desenvolvimento de técnicas que aumentem a sua produtividade. Vários experimentos foram desenvolvidos com o objetivo de entender o processo de depuração de software e estabelecer um modelo de depuração.

Araki et al. [16], utilizando resultados de experimentos anteriores [410], definiram o modelo de depuração chamado de Hipótese-Validação descrito na Figura 12.1. O modelo definido caracteriza a atividade de depuração como um processo interativo de síntese, verificação e refinamento de hipóteses [423]. De acordo com esse modelo, o responsável pela depuração do software (chamado a partir de agora simplesmente de programador) estabelece hipóteses com relação à localização do defeito e à modificação necessária para corrigir o programa.

O processo de depuração é guiado pela verificação e pela refutação das hipóteses levantadas, bem como pela geração de novas hipóteses e refinamento das já existentes [16]. Note-se

que o modelo Hipótese-Verificação é genérico, não sendo vinculado a nenhum tipo particular de depuração. Outros autores [66, 117] também desenvolveram modelos de depuração que são variantes do modelo de Araki et al. [16], vinculados, porém, a técnicas específicas de depuração.

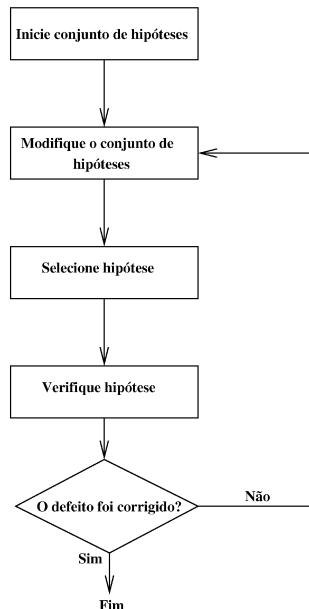


Figura 12.1 – Modelo hipótese-validação para o processo de depuração [16].

Técnicas e ferramentas de depuração propriamente ditas têm sido desenvolvidas para auxiliar a realização de uma ou mais tarefas descritas no modelo Hipótese-Validação. Um exemplo típico são os depuradores simbólicos [4, 379]. Esse tipo de ferramenta permite que a execução do programa pare em pontos determinados pelo programador e que os valores das variáveis e da pilha de execução sejam examinados. Nesse sentido, eles apóiam a verificação e a refutação das hipóteses levantadas pelo programador.

Outras técnicas, por sua vez, apóiam a geração de hipóteses a serem verificadas. Por exemplo, a inclusão de asserções [101, 347] no código-fonte do programa busca identificar pontos do programa que, ao serem atingidos durante a execução do dado de teste, violam a especificação original do software. Esses comandos são hipóteses de possíveis locais onde se encontra o defeito. De maneira similar, a técnica de fatiamento (*slicing*) de programas [9, 221, 427] identifica um conjunto de comandos (chamado de fatia – *slice*) candidatos a conterem o defeito. A fatia é composta de comandos que influenciam o valor de uma variável em determinado ponto do programa. Adicionalmente, a eleição de comandos “suspeitos” pode ser feita por meio de heurísticas que manipulam tanto fatias de programas como informação coletada durante o teste [10, 89, 254, 322]. A técnica de depuração algorítmica [146, 219, 353], por sua vez, apóia duas tarefas do modelo Hipótese-Validação: geração e seleção de hipóteses. Ela apóia as duas tarefas e guia o programador até o local do defeito, usando de uma busca binária em um conjunto de comandos suspeitos.

Como se pode observar, todas as tarefas do modelo Hipótese-Validação são apoiadas por técnicas e ferramentas. No entanto, em ambientes industriais de desenvolvimento de software, é comum o uso apenas de depuradores simbólicos para apoiar a depuração de software. Essa situação é mais dramática quando considerados dois aspectos em relação aos depuradores simbólicos: 1) trata-se de ferramentas disponíveis desde o final dos anos 60; e 2) basicamente apóiam apenas uma tarefa do modelo Hipótese-Validação – a verificação de hipóteses.

Dessas observações, conclui-se que a atividade de depuração é realizada da mesma maneira há pelo menos 30 anos e depende, essencialmente, da experiência do programador, pois, o qual, para a geração e seleção de hipóteses, não conta com o apoio de técnicas e ferramentas. Essa situação foi definida por Lieberman [240] como um escândalo e tem sido creditada ao custo das técnicas de geração e seleção de hipóteses, que, em geral, não são escaláveis para uso em sistemas reais [239, 300, 436]. Todavia, recentemente, progressos têm sido obtidos no desenvolvimento de técnicas de depuração mais escaláveis.

Neste capítulo, procura-se descrever o estado-da-arte das técnicas de depuração de programas e os principais desafios para que essas técnicas passem a fazer parte do estado da prática dos desenvolvedores de software. As técnicas descritas são aplicáveis em programas escritos em linguagens procedimentais. Quando aplicáveis em outros paradigmas de programação, isto é mencionado. Em especial, será dada ênfase às atividades de depuração que ocorrem depois do teste de programas.

Na próxima seção, é apresentado o modelo de Depuração Depois do Teste (DDT). Trata-se de uma variação do modelo de depuração Hipótese-Validação, porém com ênfase na utilização de informação de teste na depuração. Esse modelo será utilizado como guia para a análise de diversas técnicas e ferramentas de depuração que são discutidas na Seção 12.3. Dois aspectos serão considerados nesta análise: tipo de informação de teste utilizada e escalabilidade para programas reais. Na Seção 12.4 é realizada uma comparação entre as diversas técnicas e ferramentas, enfatizando as que utilizam informação de teste sistemática e que são mais promissoras em termos de escalabilidade para programas reais. A Seção 12.5 contém as considerações finais.

## 12.2 Modelo de depuração depois do teste

Nesta seção, são discutidas sucintamente as diferentes ocorrências da atividade de depuração no ciclo de vida de software. É dada ênfase especial à depuração que ocorre depois da atividade de teste. Um modelo que descreve os passos principais da depuração depois do teste é apresentado.

### 12.2.1 Depuração no ciclo de vida

A atividade de depuração ocorre no processo de software em três momentos distintos: durante a codificação, depois do teste e durante a manutenção. Durante a codificação, a depuração é uma ferramenta complementar à programação. O cenário típico ocorre quando o programador codifica parte da especificação e prepara um teste não-sistêmico para verificar o novo código. Em geral, ele executa o programa e verifica o resultado. Se incorreto, o novo código deve ser depurado. Outra possibilidade é não executar o programa de uma vez, mas passo

a passo no trecho relativo ao novo código. Assim, as características da depuração durante a codificação são: 1) uso de testes não-sistemáticos; e 2) ênfase na verificação do novo código introduzido, não na localização de defeitos.

A ferramenta típica utilizada neste cenário é o depurador simbólico [4, 379]. Essa ferramenta tradicional de depuração se presta bem à depuração durante a codificação porque a tarefa de localização do defeito é reduzida, visto que já se sabe que o comportamento incorreto se localiza no novo código. Outro instrumento importante para a verificação do novo código são as ferramentas que detectam usos inválidos de memória [19, 173]. Ferramentas comerciais como PurifyPlus [195] permitem a detecção de vazamento de memória (*memory leaking*) e acesso inválido por uso de apontadores.

Já a depuração que ocorre depois da atividade de teste possui características diferentes. O objeto da depuração, nesse momento, é o software obtido depois de completada a fase de implementação e que, supostamente, já possui todas as funções estabelecidas na especificação. Além disso, a depuração recebe como entrada não somente o código e a especificação, mas também os resultados da atividade de teste. Infelizmente, as ferramentas utilizadas atualmente para a depuração depois do teste são as mesmas utilizadas durante a codificação, ignorando-se a informação obtida durante o teste [6].

Várias técnicas que utilizam informação de teste durante a depuração de programas foram pesquisadas e propostas [10, 6, 74, 89, 117, 146, 221, 254, 427, 436] e pelo menos uma ferramenta comercial resultante desses esforços está disponível (Telcordia xSuds [389]). Alguns fatores, porém, têm dificultado a difusão dessas técnicas na prática, pois: 1) muitas delas utilizam informação trivial de teste (por exemplo, se o caso de teste revela ou não um defeito); 2) em parte em decorrência do primeiro fator, algumas técnicas possuem alto custo em tempo de depuração, isto é, durante a depuração; e 3) não existem técnicas para mapear a informação de teste em informação dinâmica, observável durante a execução do programa. Recentemente, novas técnicas que utilizam mais eficientemente informação de teste [87, 454, 64] a um custo menor [455, 456] foram desenvolvidas.

Durante a manutenção, novamente, há a necessidade de depuração do software. Pressman [329] lista quatro tipos diferentes de manutenção: corretiva, aperfeiçoadora (*perfective*), adaptativa e preventiva. Em todos os tipos de manutenção, com exceção da corretiva, ocorre um novo processo de desenvolvimento, implicando a codificação e o teste das novas funções identificadas; portanto, ocorre recorrentemente depuração durante a codificação e depois do teste. Durante a manutenção corretiva, ocorre essencialmente a depuração depois do teste, mas com dois problemas adicionais: o código precisa ser entendido, visto que o programador original pode não estar mais disponível ou não se lembrar mais do código implementado, e o conjunto de casos de teste é composto apenas pelos casos de teste que provocam a ocorrência de falha no software liberado. O problema de entendimento do programa é inerente a qualquer atividade de manutenção e requer o uso de técnicas de compreensão de programas. Já o conjunto de casos de teste pode ser aumentado juntando-se, quando disponíveis, os casos de teste originalmente desenvolvidos durante o teste.

### 12.2.2 O modelo DDT

Para investigar a atividade de depuração que ocorre depois do teste, foi definido o modelo de Depuração Depois do Teste (DDT). Esse modelo foi desenvolvido a partir de modelos de depuração propostos anteriormente na literatura [5, 16, 66, 117, 410] e tem como objetivo

indicar que tipo de informação deve ser utilizada e quais tarefas devem ser realizadas para apoiar especificamente a depuração depois do teste. Os passos que compõem o modelo são descritos na Figura 12.2.

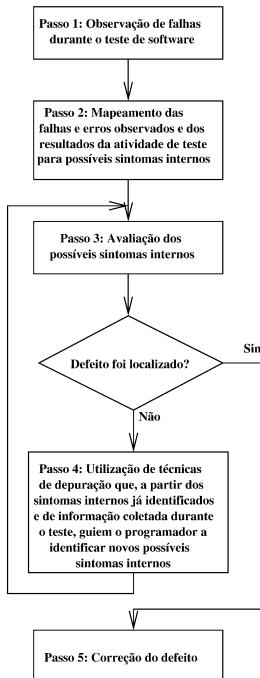


Figura 12.2 – Modelo de depuração depois do teste.

Embora o modelo inclua a observação de falhas (passo 1) e a correção do programa (passo 5), sua ênfase está na atividade de localização do defeito, por sinal, a mais difícil e custosa [294]. Isso porque muitas vezes a correção do defeito é imediata. O modelo indica no Passo 2 que são necessárias técnicas que mapeiem as falhas e erros observados e os resultados coletados durante o teste para possíveis sintomas internos. Os sintomas internos são caracterizados pelo valor de uma ou mais variáveis em determinado ponto de execução. Em outras palavras, os sintomas internos indicam um ponto de execução em que o erro ou a falha podem ser investigados. Os possíveis sintomas internos devem, então, ser avaliados e confirmados como sintomas internos propriamente ditos (Passo 3). No Passo 4, é preconizado o uso de técnicas de depuração que ajudem o programador a refinar os sintomas internos inicialmente identificados e que acabem por levá-lo a localizar o defeito.

O modelo enfatiza o uso de informação de teste durante a depuração. Entretanto, é necessário qualificar o tipo de informação utilizada. Se ela é obtida de testes não-sistemáticos, então é considerada básica. Por exemplo, um relatório de teste que indique tão-somente se os casos de teste revelam a presença de defeitos ou não é uma informação básica. A informação de teste é detalhada quando inclui os produtos gerados durante o teste sistemático do software. Exemplos desses produtos são as entradas e saídas de casos de teste desenvolvidos utilizando técnicas funcionais (Capítulo 2), os requisitos de teste estrutural (Capítulo 4) exer-

citados pelos casos de teste, os programas mutantes “mortos” pelos casos de teste na análise de mutantes (Capítulo 5), etc.

## 12.3 Técnicas de depuração e o modelo DDT

Nesta seção, são analisadas várias técnicas de depuração com respeito ao modelo DDT. As técnicas são apresentadas resumidamente e avaliadas com relação aos aspectos que o modelo mais enfatiza: identificação de possíveis sintomas internos, avaliação dos sintomas, apoio à seleção de novos possíveis sintomas internos e tipo de informação de teste utilizada. As técnicas revisadas são, também, analisadas quanto à escalabilidade para sistemas reais.

### 12.3.1 Depuração baseada em rastreamento e inspeção

A depuração baseada em rastreamento e inspeção é a mais usada na prática. O sucesso dessa técnica deve-se a três fatores: seu uso requer apenas treinamento básico; o custo em tempo de depuração é razoável; e a sua disponibilidade é ampla porque está presente em qualquer ambiente de programação. Esse tipo de depuração envolve o rastreamento de eventos e a inspeção do estado do programa no momento em que ocorre um evento.

Na sua forma mais básica, a depuração baseada em rastreamento e inspeção é realizada com a ajuda de comandos de escrita. O programador coloca em pontos estratégicos do programa comandos para imprimir os valores de determinadas variáveis. O objetivo é rastrear um determinado ponto do programa durante a execução e inspecionar o valor das variáveis escolhidas, ou seja, examinar o estado parcial do programa. Com os depuradores simbólicos [4, 379], o rastreamento de alguns tipos de eventos e a inspeção do estado do programa ficaram extremamente facilitados. Por exemplo, usando os comandos de um depurador simbólico, o programador pode parar a execução do programa ajustando um *breakpoint* no ponto desejado, observar os valores das variáveis, verificar a seqüência de chamadas de procedimentos examinando a pilha de chamadas, atribuir novos valores para variáveis, etc.

Ferramentas mais modernas têm desenvolvido mecanismos de rastreamento de novos eventos e de inspeção de diferentes informações relativas ao estado do programa. DUEL [156] é um depurador simbólico que possibilita a especificação de expressões que avaliam o estado do programa. Por exemplo, DUEL permite que o programador consulte quais elementos de um vetor possuem um valor maior, menor ou igual a uma constante ou ao de outra variável. Lancevicius et al. [238] utilizam uma linguagem de consulta semelhante às utilizadas em banco de dados para inspecionar as relações entre as instâncias das classes em uma linguagem orientada a objetos. A ferramenta Coca [121], por sua vez, permite o acesso a eventos relacionados às construções da linguagem de programação (por exemplo, procedimentos, comandos de controle de fluxo) usando uma linguagem declarativa do estilo Prolog. Por meio da linguagem de consulta de Coca, o programador pode parar a execução do programa em eventos básicos associados à entrada ou à saída dos comandos de controle de fluxo da linguagem C, restringindo, porém, os eventos por meio dos valores das variáveis, do número de ocorrências, etc. Além disso, Coca inclui um mecanismo simplificado de consulta ao estado do programa baseado na linguagem Prolog.

Uma característica interessante de ser incluída nos depuradores simbólicos é a execução em reverso, isto é, permitir que o programador execute o programa em sentido contrário.

Há uma longa série de iniciativas de inclusão dessa característica em ferramentas experimentais [8, 21, 241, 396]. Entretanto, não há depuradores simbólicos disponíveis comercialmente com essa função. O fator limitante é que, mesmo nas implementações mais eficientes, é ainda exigida uma grande quantidade de memória para registrar as alterações tanto no estado do programa quanto nos arquivos de entrada e saída. Esses dois requisitos dificultam sua utilização em programas reais. Algoritmos para execução bidirecional de programas foram propostos [41, 73]. Contudo, eles ainda precisam ser avaliados quanto à sua escalabilidade.

Nem todas as ferramentas modernas de rastreamento de eventos e inspeção do estado do programa operam de forma interativa como os depuradores simbólicos. Algumas ferramentas fornecem um relatório depois de terminada a execução do programa (análise *post mortem*), no qual são relatados os eventos observados juntamente com os respectivos estados do programa. Por exemplo, as ferramentas que permitem verificar o comportamento da memória durante a execução fornecem um relatório que indica os pontos do programa em que ocorreram vazamentos de memória e acessos inválidos [19, 173, 195]. As ferramentas FORMAN [18] e CCI [390], por sua vez, permitem que o programador defina eventos a serem monitorados utilizando um conjunto de eventos básicos. Os eventos monitorados são compostos por eventos básicos associados a operações de atribuição e a operações aritméticas, ao acesso a posições de memória de variáveis em particular, a pontos do programa, etc. O programa é, então, compilado e instrumentado para monitorar os eventos definidos pelo programador. Depois da execução, um relatório é produzido.

Com relação à escalabilidade para programas reais, os depuradores simbólicos, com exceção daqueles que incluem execução em reverso, são ferramentas utilizadas na depuração de programas dos mais diversos tipos, apesar do impacto no tempo de execução causado pela instrumentação introduzida nos programas. A escalabilidade das ferramentas construídas a partir deles (por exemplo, DUEL, Coca) depende de como é feita a comunicação entre essas ferramentas e o depurador simbólico. Se for freqüente a comunicação, a escalabilidade pode ser comprometida. Ferramentas de análise *post mortem* que realizam rastreamento de erros de memória estão disponíveis comercialmente [195] e são importantes para a depuração de sistemas reais, em especial na depuração durante a codificação. Todavia, outras ferramentas do mesmo tipo [18, 390] podem gerar arquivos de dados ou programas excessivamente grandes. Isto ocorre quando os eventos monitorados são muito freqüentes ou quando muitos pontos do programa precisem ser instrumentados.

As ferramentas para rastreamento e inspeção são fundamentais em qualquer abordagem de depuração por permitirem a avaliação dos sintomas internos (Passo 3 do modelo DDT). Além disso, elas também permitem o mapeamento de falhas e erros para sintomas internos (Passo 2), especialmente as ferramentas de análise *post mortem*. Entretanto, essas ferramentas utilizam informação básica de teste para realizar as duas tarefas. Com relação à seleção de novos sintomas internos (Passo 4), nenhum apoio é fornecido. O programador depende essencialmente da sua experiência para realizar essa tarefa.

### 12.3.2 Depuração com asserções

A depuração com asserções é baseada na inclusão de parte da especificação do programa no seu código-fonte, de maneira que uma ação é ativada toda vez que a especificação parcial do programa é violada durante a execução. Segundo von Staa [377, 378], as asserções executáveis reduzem o esforço de localização de defeitos, pois o defeito e seus efeitos estão

próximos. Eles estão próximos porque em geral o número de comandos executados e o tempo entre o alcance do defeito e a observação dos efeitos são pequenos.

Essa técnica de depuração foi desenvolvida no final dos anos 60. Atualmente, algumas linguagens de programação já incluem construções que permitem seu uso como a macro pre-definida `assert` do padrão ANSI da linguagem C. Mais recentemente, algumas ferramentas como ASAP [101] e APP [347] aumentaram ainda mais o poder expressivo das asserções. A seguir, é descrito um exemplo que utiliza asserções e a ferramenta APP.

O Programa 12.1 anotado com asserções realiza a troca de valores de duas variáveis inteiros sem a utilização de uma variável intermediária por meio da operação de ou-exclusivo. As asserções estão incluídas nos indicadores de comentários especiais `/*@ ... @*/`. A cláusula `assume` define uma pré-condição (válida antes da execução do procedimento); a cláusula `promise` especifica uma pós-condição (válida depois de executado o procedimento); e a cláusula `assert` indica uma condição que deve ser verdadeira no corpo. O operador `in` retorna o valor de uma variável antes da execução do procedimento. O Programa 12.2 contém a rotina que é ativada quando a cláusula `promise` é violada.

---

Programa 12.1

---

```

1 void swap(x,y)
2 int * x;
3 int * y;
4
5 /*@
6     assume x && y && x != y;
7     promise *x == in *y;
8     promise *y == in *x;
9 @*/
10 {
11
12     *x = *x ^ *y;
13     *y = *x ^ *y;
14
15 /*@
16     assert *y == in *x;
17 */
18     @*/
19 }
20
21 }
```

---

\*

---

Programa 12.2

---

```

1 promise * x == in * y {
2 printf("%s invalid: file %s, ", __ANNONAME__, __FILE__);
3 printf("line %d, function %s:\n", __ANNOLINE__, __FUNCTION__);
4 printf("out *x == %d, out *y == %d\n", *x, *y);
5 }
```

---

\*

---

A depuração com asserções permite o mapeamento de erros para sintomas internos, pois elas indicam pontos do programa nos quais ocorrem discrepâncias em relação à especificação. Portanto, essa técnica de depuração apóia o Passo 2 do modelo DDT, utilizando informação básica de teste. Segundo Rosenblum [347], o custo em termos de espaço e tempo de execução dos programas anotados com asserções é insignificante, o que viabiliza sua utilização em sistemas reais. Entretanto, essa técnica requer a codificação adicional de parte da especificação, que, por sua vez, poderá, também, conter defeitos.

### 12.3.3 Fatiamento de programas

Nesta subseção são apresentadas diferentes técnicas para “fatiar” programas, isto é, para identificar trechos de código do programa com grande probabilidade de conter defeitos.

#### Fatiamento estático e dinâmico

Fatiamento<sup>1</sup> (*slicing*) de programas é uma técnica cujo objetivo é selecionar fatias (*slices*) do programa. Uma fatia é um conjunto de comandos que afetam os valores de uma ou mais variáveis em determinado ponto do programa. As variáveis e o ponto do programa definem o critério de fatiamento. A fatia pode ser determinada estática ou dinamicamente. No primeiro caso, os comandos selecionados podem afetar as variáveis no ponto especificado para alguma possível entrada do programa. No segundo caso, os comandos selecionados efetivamente afetam os valores das variáveis no ponto especificado para determinada entrada.

Tanto as fatias estáticas quanto as dinâmicas podem ser executáveis ou não. Se executável, a fatia é um subconjunto das instruções do programa original que também é um programa. Esse novo programa fornece os mesmos valores para as variáveis selecionadas no ponto especificado do programa original. Fatias executáveis em geral são maiores porque precisam incluir declarações de variáveis e outros comandos que não afetam o critério de fatiamento, mas são necessários para a execução.

A Figura 12.3 contém um programa que calcula a soma das áreas de N triângulos, bem como diferentes tipos de fatias deste programa. O programa e as fatias estática e dinâmica foram obtidos da tese de doutorado de Agrawal [5]. O critério de fatiamento utilizado para a fatia estática foi a variável `sum` localizada no comando 42. A fatia dinâmica foi determinada para o caso de teste cujos valores de entrada são:  $N = 2$  e os triângulos  $(3, 3, 3)$  e  $(6, 5, 4)$ ; os valores de saída foram:  $N = 2$  e  $sum = 13,90$ . O critério de fatiamento utilizado foi o valor da variável `sum` quando da execução do último comando, no caso o comando 42. Esse caso de teste falha e revela o defeito localizado no comando 18. O valor correto para `sum` é 13,82.

A técnica de fatiamento de programas apóia dois aspectos do modelo DDT: mapeamento de falhas para possíveis sintomas internos (Passo 2) e seleção de novos possíveis sintomas a partir daqueles inicialmente identificados (Passo 4). Nas duas tarefas é utilizada informação básica de teste.

Essa técnica, porém, possui alguns problemas. O primeiro é o tamanho das fatias, tanto estáticas (especialmente) como dinâmicas. O exemplo da Figura 12.3 ilustra esse problema. O número de comandos selecionados pelas fatias estática (SS) e dinâmica (DS) não é muito diferente do número total de comandos do programa. Para programas de grande porte, o número de comandos que afetam um critério de fatiamento pode também ser muito grande, o que pode tornar a técnica pouco atrativa.

Para superar parte desse problema, Korel e Rilling [223] desenvolveram, internamente aos procedimentos, fatias dinâmicas parciais (por exemplo, fatia do código de um laço) de forma a restringir o tamanho do “pedaço” de código que o programador terá de investigar.

<sup>1</sup>Existem muitos artigos que discutem fatiamento de programas além das referências básicas [9, 221, 427]. Em especial, Tip [395] e Kamkar [205] possuem excelentes revisões bibliográficas que discutem vários aspectos dessa técnica (por exemplo, fatiamento estático e dinâmico, algoritmos, estrutura de dados, fatiamento intra e interprocedimental, tratamento de apontadores e de programas não-estruturados, custo, etc.).

| SS | DS | SD | DD | ED | Cmd | Código   |
|----|----|----|----|----|-----|--|
| ✓  | ✓  |    |    |    | 1   | #define MAX 100  |
| ✓  | ✓  |    |    |    | 2   | typedef enum { equilateral, isosceles, right, scalene} |
| ✓  | ✓  |    |    |    | 3   | class_type;  |
| ✓  | ✓  |    |    |    | 4   | main()   |
| ✓  | ✓  |    |    |    | 5   | {  |
| ✓  | ✓  |    |    |    | 6   | int a, b, c;   |
| ✓  | ✓  |    |    |    | 7   | class_type class;                                      |
| ✓  | ✓  |    |    |    | 8   | int a_sqr, b_sqr, c_sqr, N, i;                         |
| ✓  | ✓  |    |    |    | 9   | double area, sum, s;                                   |
| ✓  | ✓  |    |    |    | 10  | printf("Enter number of triangles:");                  |
| ✓  | ✓  |    |    |    | 11  | scanf("%d", &N);                                       |
| ✓  | ✓  |    |    |    | 12  | sum = 0;   |
| ✓  | ✓  |    |    |    | 13  | i=0;   |
| ✓  | ✓  |    |    |    | 14  | while ( i < N) {                                       |
| ✓  | ✓  |    |    |    | 15  | printf("Enter three sides of triangle                  |
| ✓  | ✓  |    |    |    | 16  | in descending order:", i+1);                           |
| ✓  | ✓  |    |    |    | 17  | scanf("%d %d %d", &a, &b, &c);                         |
| ✓  | ✓  |    |    |    | 18  | a_sqr = a * a;   |
| ✓  | ✓  |    |    |    | 19  | b_sqr = b * c; /* <- correct: b_sqr = b * b; */        |
| ✓  | ✓  |    |    |    | 20  | c_sqr = c * c;   |
| ✓  | ✓  |    |    |    | 21  | if((a == b) && (b == c))                               |
| ✓  | ✓  |    |    |    | 22  | class = equilateral;                                   |
| ✓  | ✓  |    |    |    | 23  | else   |
| ✓  | ✓  |    |    |    | 24  | if((a == b)    (b == c))                               |
| ✓  | ✓  |    |    |    | 25  | class = isosceles;                                     |
| ✓  | ✓  |    |    |    | 26  | else   |
| ✓  | ✓  |    |    |    | 27  | if(a_sqr == b_sqr + c_sqr)                             |
| ✓  | ✓  |    |    |    | 28  | class = right;   |
| ✓  | ✓  |    |    |    | 29  | else   |
| ✓  | ✓  |    |    |    | 30  | class = scalene;                                       |
| ✓  | ✓  |    |    |    | 31  | if(class == right)                                     |
| ✓  | ✓  |    |    |    | 32  | area = b * c/2.0;                                      |
| ✓  | ✓  |    |    |    | 33  | else   |
| ✓  | ✓  |    |    |    | 34  | if(class == equilateral)                               |
| ✓  | ✓  |    |    |    | 35  | area = a * a * sqrt(3.0)/4.0;                          |
| ✓  | ✓  |    |    |    | 36  | else   |
| ✓  | ✓  |    |    |    | 37  | s = (a + b + c)/2.0;                                   |
| ✓  | ✓  |    |    |    | 38  | area = sqrt(s*(s-a)*(s-b)*(s-c));                      |
| ✓  | ✓  |    |    |    | 39  | }  |
| ✓  | ✓  |    |    |    | 40  | sum=sum+area;  |
| ✓  | ✓  |    |    |    | 41  | i=i+1;   |
| ✓  | ✓  |    |    |    | 42  | }  |
| ✓  | ✓  |    |    |    | 43  | printf("Sum of areas of the %d triangles is %.2f.",    |
|    |    |    |    |    |     | N, sum);   |
|    |    |    |    |    |     | }  |

SS – static slice; DS – dynamic slice; DD – dynamic dice; ED – execution dice

Figura 12.3 – Exemplos de fatias estáticas e dinâmicas.

Em termos de sistema, os autores desenvolveram fatias de informação relativa à interação dos procedimentos (por exemplo, fatia do grafo de chamada) [224].

O segundo problema da técnica é o custo. O fatiamento dinâmico de programas requer o rastreamento dos comandos e das posições de memória, de maneira a identificar precisamente os comandos que afetam um critério de fatiamento. Esse requisito impõe um custo muito grande em tempo de depuração para programas que possuem longas execuções [300, 436]. Novas técnicas de fatiamento dinâmico [455, 456] conseguiram obter fatias de programas de longa duração a custo bastante reduzido. No entanto, elas realizam um pré-processamento

dos dados coletados durante a execução (comandos executados e posições de memória acessadas) que pode demorar dezenas de minutos. Uma vez realizado esse pré-processamento, diferentes fatias de uma mesma execução podem ser obtidas em segundos.

## Fatiamento de programas usando informação de teste

Pan [117, 321] propõe a identificação de um novo tipo de fatia, chamada fatia crítica, durante o teste baseado em defeitos (análise de mutantes). Esse novo tipo de fatia é definido da maneira a seguir. Suponha-se que um programa  $P$  tenha produzido um valor incorreto para uma variável de saída  $v$ . Seja  $M$  uma versão alterada de  $P$  (mutante), em que apenas um comando  $S$  tenha sido eliminado. Se o valor de  $v$  é diferente quando  $M$  é executado com um caso de teste, então ele é um comando crítico. A fatia crítica é composta pelos comandos críticos do programa  $P$ .

O custo de determinação da fatia crítica isoladamente é muito alto, visto que, para um programa com  $n$  comandos, seria necessário executar  $n$  programas mutantes  $M$  com cada caso de teste [117]. Entretanto, esse custo pode ser amortizado durante o teste se a fatia crítica for obtida durante a análise de mutantes utilizando o operador eliminação de comando. DeMillo et al. [117] descrevem um experimento com programas pequenos em que as fatias críticas selecionaram 25% menos comandos que as fatias dinâmicas.

Agrawal et al. [10, 6] propõem a determinação de uma fatia do programa a partir dos resultados obtidos do teste estrutural do programa. O conjunto de comandos associados aos requisitos de teste estruturais (por exemplo, nós, ramos, c-usos e p-usos) executados por um caso de teste particular é chamado de fatia de execução.

A vantagem das fatias baseadas em informação de teste é que a maior parte do custo para sua obtenção já foi amortizada durante o teste do programa. Entretanto, de modo semelhante às outras fatias, há uma grande probabilidade de essas fatias incluírem um número elevado de comandos. As fatias críticas, apesar da possível redução de 25% trazida em relação às fatias dinâmicas, muito provavelmente incluem uma grande fatia de código quando utilizadas em programas complexos e críticos. Já as fatias de execução são sempre iguais ou maiores que as fatias dinâmicas.

Tanto as fatias críticas quanto as de execução são úteis para mapear informação detalhada de teste para possíveis sintomas internos (comandos suspeitos). Portanto, essas duas técnicas apóiam o Passo 2 do modelo DDT. Contudo, elas não apóiam a seleção de novos sintomas internos (Passo 4). Isso porque as fatias crítica e de execução são determinadas a partir de casos de teste, e não a partir de um critério de fatiamento que pode ser variado. Logo, não é possível refiná-las.

## Heurísticas baseadas em fatias

Heurísticas utilizando fatias têm sido propostas para reduzir o espaço de busca para localização do defeito [5, 10, 89, 74, 254, 322, 321]. A idéia é realizar operações com as fatias para determinar um conjunto menor de comandos com grande probabilidade de conter o defeito. As heurísticas mais simples realizam operações de subtração, intersecção e união de fatias [322, 321].

Lyle and Weiser [254] introduziram o recorte<sup>2</sup> de fatias estáticas. Os autores propõem a subtração das fatias obtidas de variáveis de saída corretas, isto é, variáveis cujos valores estão corretos para todos os casos de teste, das fatias de variáveis de saída incorretas, isto é, que produziram um valor incorreto em pelo menos um caso de teste. A intuição subjacente é que a eliminação dos comandos comuns a ambas as fatias pode levar à identificação de um conjunto menor de comandos com maior chance de conter o defeito.

De maneira análoga, as fatias dinâmicas podem ser utilizadas para a obtenção de fragmentos de recorte [5, 74, 322, 321]. Nesse caso, os fragmentos de recorte podem ser calculados usando fatias das mesmas variáveis, não necessariamente de saída, que tenham produzido valores corretos e incorretos em dois casos de teste diferentes. Fragmentos de recorte podem ser igualmente obtidos a partir de fatias baseadas em informação de teste. A técnica de recorte pode ainda envolver a subtração entre o resultado da união ou intersecção de fatias de variáveis obtidas de casos de teste que falham e o resultado da união ou intersecção de fatias de casos de teste que passam.

Na Figura 12.3, são apresentados exemplos de fragmentos de recorte. Pela utilização do caso de teste que falha apresentado anteriormente, o fragmento de recorte estático (SD) foi determinado fazendo a subtração da fatia estática de  $N$  (variável correta) da fatia estática de  $\text{sum}$  (variável incorreta), ambas as fatias com relação ao comando 42. O fragmento de recorte dinâmico foi igualmente determinado utilizando um caso de teste que passa. Para  $N = 2$  e triângulos  $(4, 4, 4)$  e  $(5, 3, 3)$ , o programa produz o valor correto de  $\text{sum}$  igual a 11,07. As fatias dinâmicas foram determinadas com respeito a  $\text{sum}$  no comando 42 para ambos os casos de teste – que falha e que passa. O fragmento de recorte dinâmico (DD) foi obtido da subtração das fatias dinâmicas por Agrawal [5]. O fragmento de recorte de execução (ED) é calculado subtraindo os nós exercitados pelo caso de teste que passa dos nós exercitados pelo caso de teste que falha.

A Figura 12.3 mostra que as heurísticas baseadas em fatias reduzem o espaço de busca dos defeitos. O fragmento de recorte dinâmico, o qual inclui apenas os comandos que realmente afetam o critério de fatiamento, contém apenas seis comandos e inclui o comando que contém o defeito. No entanto, a Figura 12.3 também ilustra alguns dos problemas relacionados com as heurísticas, em especial com os fragmentos de recorte. Por exemplo, o fragmento de recorte estático é quase igual à fatia estática de  $\text{sum}$ . Isso ocorre porque a intersecção entre as fatias estáticas da variável incorreta e da variável correta contém apenas um comando (11). O fragmento de recorte de execução, por sua vez, não inclui o comando com o defeito, pois o nó no qual o comando “defeitoso” está contido é exercitado pelo caso de teste que falha e pelo que passa. É importante observar que esse problema também pode ocorrer com os fragmentos de recorte dinâmicos.

Outra maneira de identificar, heuristicamente, comandos do programa com grande probabilidade de conter o defeito é estabelecendo um ranking entre eles. Nesse ranking, os comandos ocorridos mais freqüentemente em fatias de casos de teste que manifestam falhas são mais bem classificados do que os não tão freqüentemente ocorridos [322, 321]. Essa idéia foi inicialmente proposta por Collofello e Cousins [89] para evitar que um defeito localizado em um trecho do código executado tanto por casos de teste que falham quanto que não falham fosse eliminado na operação de subtração da técnica de recorte.

<sup>2</sup>Os termos *dicing* (fatiamento em cubos) e *dice* (cubo, dado) foram traduzidos para os termos recorte e fragmento de recorte, respectivamente.

Esses autores definem dez heurísticas que realizam operações de recorte e de ranking de nós (obtidos durante o teste com o critério todos nós) para identificar trechos de código suspeitos [89]. Agrawal et al. [10] revisitaram essa abordagem para a definição das fatias de execução e também de heurísticas baseadas em recorte de outros requisitos de teste (por exemplo, ramos, p-usos, c-usos) executados pelos casos de teste.

O uso de heurísticas impõe alguns riscos. Apesar de a intuição ser de haver grande probabilidade do trecho de código selecionado conter o defeito, também há a chance de ele ser excluído durante as operações envolvendo conjuntos (por exemplo, intersecção, subtração) ou durante a criação do ranking. Nesse último caso, o trecho selecionado pode incluir os efeitos do defeito, mas excluir o próprio defeito. Além disso, as heurísticas que operam sobre fatias possuem as mesmas restrições inerentes à técnica utilizada para obtê-las.

Do ponto de vista do modelo DDT, as heurísticas que utilizam fatias estáticas e dinâmicas, da mesma maneira que a técnica de fatiamento de programas, apóiam as tarefas definidas nos Passos 2 e 4, utilizando informação básica de teste. Já as heurísticas que utilizam fatias baseadas em informação de teste apóiam somente a tarefa de mapeamento para sintomas internos (Passo 2), utilizando informação detalhada de teste.

### Depuração baseada em requisitos de teste

As heurísticas baseadas em requisitos de teste estrutural [10, 89, 322] apóiam apenas o Passo 2 do modelo DDT porque se baseiam em uma informação estática – o trecho de código obtido do mapeamento dos requisitos de teste selecionados. Com o objetivo de tratar esse problema, Chaim et al. [63, 64] propuseram o uso de informação dinâmica de teste para a localização de defeitos.

O pressuposto é que o conjunto de requisitos selecionados pelas heurísticas ainda fornecem indicações úteis para a depuração mesmo quando falham em incluir o defeito no trecho de código selecionado. No entanto, essas informações estão disponíveis em tempo de execução. Os autores desenvolveram estudos empíricos que mostram que esse pressuposto é válido. A partir dele, foi desenvolvida uma estratégia de depuração que combina o uso de heurísticas para seleção inicial de requisitos de teste e mecanismos para o refinamento sucessivo das informações obtidas em tempo de execução até a localização do defeito.

A estratégia de depuração baseada em informação dinâmica de teste parece promissora porque apóia tanto a geração (por meios de heurísticas) como a seleção (utilizando os mecanismos de refinamento) de hipóteses, ou seja, os Passos 2 e 4 do modelo DDT. Adicionalmente, os mecanismos desenvolvidos podem ser implementados em depuradores simbólicos com *overhead* limitado no seu desempenho e utilizam algoritmos de ordem linear em função do número de ramos do programa.

#### 12.3.4 Depuração algorítmica

A técnica de depuração algorítmica objetiva guiar o programador até o sítio do defeito. A técnica utiliza respostas fornecidas pelo programador para restringir o espaço de busca. Nesta subseção são discutidas as depurações inter e intraprocedimental.

## Depuração algorítmica interprocedimental

A técnica de depuração algorítmica foi originalmente desenvolvida para programas sem efeitos colaterais escritos em Prolog [353]. Essa técnica é baseada em um processo interativo durante o qual o programador fornece conhecimento a respeito do comportamento esperado do programa ao sistema de depuração e este, por sua vez, guia o programador durante o processo de localização do defeito [146].

A técnica funciona como a seguir. Para a localização do defeito, o caso de teste que manifestou uma falha deve ser executado sob a supervisão do sistema baseado em depuração algorítmica. O sistema cria, então, uma árvore de execução na qual os nós representam invocações dos procedimentos do programa. Esses nós contêm o nome do procedimento e os valores de entrada e saída utilizados na invocação em particular. O sistema, então, visita, partindo do procedimento de mais alto nível, os nós da árvore de execução, perguntando ao programador se os valores dos parâmetros de entrada e saída estão corretos. Se a resposta for positiva, o processo continua visitando os próximos nós de mesmo nível. Caso contrário, o processo visita os nós dos procedimentos de nível inferior invocados pelo procedimento de nível superior. Esse processo termina quando é identificado um procedimento no qual os parâmetros de entrada estão corretos e os de saída incorretos ou que não faz chamada a nenhum outro procedimento ou, se o faz, os valores dos parâmetros de entrada e saída dos procedimentos invocados estão corretos.

Um exemplo de depuração algorítmica, apresentado por Fritzson et al.[146], para programas escritos em Pascal é descrito utilizando o Programa 12.3. O procedimento *P* possui dois parâmetros de entrada *a* e *c* e calcula os valores de dois parâmetros de saída *b* e *d*. O valor de *b* é calculado chamando-se o procedimento *Q*, e o valor de *d* é obtido da chamada ao procedimento *R*.

---

Programa 12.3

---

```
1  procedure P(a,c:integer; var b, d:integer);
2      procedure Q(a:integer; var b: integer);
3      ...
4      end;
5      procedure R(c:integer; var d: integer);
6      ...
7      end;
8  begin
9      Q(a,b);
10     R(c,d);
11 end;
```

---

Suponha que um defeito esteja localizado no procedimento *R*. O programa é então executado com as entradas *a'* e *c'* e produz as saídas *b'* e *d'*; *d'* é um valor de saída incorreto. O sistema de depuração algorítmica funciona como exemplificado na Figura 12.4.

Caso o programador responda corretamente às questões feitas pelo sistema de depuração algorítmica, o procedimento no qual está o defeito é identificado. Entretanto, a aplicabilidade dessa técnica em programas reais é reduzida. Entre as dificuldades para sua utilização estão: o número de perguntas que o programador deve responder, o tratamento de efeitos colaterais, especialmente os causados pelo uso de apontadores, e o espaço (não-limitado) requerido pela árvore de execução cujo tamanho depende do número de invocações dos procedimentos.

```
P(In a: a', In c: c', Out b: b', Out d: d')?  
no  
Q(In a: a', Out b: b')?  
yes  
R(In c: c', Out d: d')?  
no  
An error is localized inside the body of procedure R.
```

---

Figura 12.4 – Sistema de depuração algorítmica.

Fritzson et al. [146] desenvolveram um sistema de depuração algorítmica para a linguagem Pascal (*GADT — Generalized Algorithmic Debugging and Testing*) no qual a técnica de fatiamento dinâmico de programas e informação de teste funcional [319] são usadas para reduzir o número de perguntas a serem respondidas pelo programador. O GADT funciona da maneira a seguir. Suponha que um procedimento  $P$  tenha produzido um valor incorreto para o parâmetro de saída  $q$ . O GADT determina, usando fatiamento dinâmico [206], as invocações dos procedimentos chamados por  $P$  que influenciaram o valor  $q$ , de forma que somente essas invocações são visitadas durante o processo de depuração. Além disso, antes de perguntar se os parâmetros de entrada e saída de uma invocação de procedimento estão corretos, o GADT verifica em uma base de dados se os valores dos parâmetros de entrada e saída já foram executados por algum caso de teste não-revelador de defeito ou se pertencem a uma categoria que contém apenas esse tipo de caso de teste. Em caso afirmativo, o GADT também ignora esse procedimento e visita o próximo.

O sistema GADT procura reduzir um dos problemas da técnica de depuração algorítmica, que é o número de interações com o programador. Entretanto, outros problemas importantes, como o tamanho da árvore de execução e o tratamento de efeitos colaterais causados por apontadores, não são tratados, o que ainda torna sua aplicabilidade reduzida [239].

Com relação ao modelo DDT, a técnica de depuração algorítmica apóia essencialmente a identificação de novos possíveis sintomas internos até a localização do procedimento defeituoso (Passo 4). A definição da técnica não prevê o uso de informação detalhada de teste, mas ela pode ser útil à depuração algorítmica, como evidenciado pelo sistema GADT.

## Depuração algorítmica intraprocedimental

A técnica de depuração algorítmica como proposta por Shapiro [353] visa à identificação do procedimento no qual se encontra o defeito; entretanto, ela não apóia a localização internamente ao procedimento. Korel [219, 223, 224] utiliza as técnicas de depuração algorítmica e fatiamento de programas para estabelecer uma estratégia para encontrar o defeito dentro do procedimento. A ferramenta PELAS (*Program Error-Locating Assistant System*) implementa essa estratégia. Outras ferramentas como Spyder [5, 423], FIND [354, 355] e ALICE [218] adotam abordagens semelhantes variando o mecanismo de interação com o usuário e a técnica de fatiamento utilizada. A seguir, é mostrado como a ferramenta PELAS é utilizada para descrever a localização de defeitos baseada em depuração algorítmica e fatiamento de programas.

A PELAS analisa estaticamente um procedimento identificado como defeituoso e produz durante a execução do programa um grafo chamado rede de dependência. Os nós da rede de dependência representam os pontos de execução<sup>3</sup> do caso de teste; e os arcos representam as relações entre os nós. Essas relações são descritas a seguir:

1. **Influência de dados:** relação estabelecida entre a definição de uma variável  $v$  no ponto de execução  $X^p$  e o subsequente uso de  $v$  no ponto  $Y^q$ , desde que não haja nenhuma definição de  $v$  nos comandos executados entre  $X^p$  e  $Y^q$ .
2. **Influência de controle:** é a relação entre o ponto de execução  $X^p$ , sendo  $X$  um comando de controle de fluxo, e os demais pontos cuja execução é determinada pelo resultado da avaliação do predicado do comando de controle de fluxo  $X$  no ponto  $X^p$ .
3. **Influência potencial:** considere o ponto de execução  $X^p$ , sendo  $X$  um comando de controle de fluxo, e o ponto de execução  $Y^q$  tal que existe um uso de  $v$  em  $Y^q$  e não há nenhuma definição de  $v$  nos comandos executados entre  $X^p$  e  $Y^q$ . Se existe um caminho alternativo entre  $X$  e  $Y$  tal que, se esse caminho fosse executado, ocorreria uma redefinição de  $v$ , então entre  $X^p$  e  $Y^q$  há uma relação de influência potencial.

Considere o exemplo contido no Programa 12.4 (desenvolvido por Shimomura et al. [354]) para ilustrar o algoritmo de localização implementado na ferramenta PELAS. As entradas  $n = 2$  e  $a = (6, 2)$  produzem a saída incorreta  $s = 4$  (a saída correta é  $s = 8$ ). A Figura 12.5 contém os pontos de execução do programa para esse caso de teste, e a Figura 12.6 contém a rede de dependência.

---

Programa 12.4

---

```

1  get(n,a);
2  t:=1;      { correto: t:=10 }
3  s:=a[1];
4  i:=2;
5  while i <= n loop
6    s:=s-a[i]; { correto: s:=s+a[i]; }
7    i:=i+1;
8  end loop;
9  if s > t then
10   if s mod 2 != 0 then
11     s:=s+1;
12   end if;
13 end if;
14 put(s);

```

---

O sintoma interno relacionado à falha observada é o valor da variável  $s$  no ponto de execução 14<sup>11</sup>. A partir da rede de dependência, a PELAS indica ao programador três pontos de execução que influenciam o resultado de  $s$  em 14<sup>11</sup>: 6<sup>6</sup>, 9<sup>9</sup> e 10<sup>10</sup>. O programador deve então escolher um dos três para continuar a depuração. Suponha que ele escolha o ponto 6<sup>6</sup>. A PELAS recupera o estado do programa nesse ponto, e o programador é questionado se o valor de  $s$  igual a 4 em 6<sup>6</sup> é correto. Nesse caso ele está incorreto, então o sistema pergunta se os valores de entrada ( $s = 2$ ,  $a[2] = 2$ ) em 6<sup>6</sup> estão corretos. A resposta é sim, estão corretos; logo, o defeito foi localizado no comando de número 6 do programa.

<sup>3</sup>Ponto de execução é o par  $X^p$  que indica que o comando  $X$  foi o  $p$ -ésimo comando executado.

| Comando $X^P$    | Código Fonte        | Valores          |
|------------------|---------------------|------------------|
| 1 <sup>1</sup>   | get (n, a);         | {n=2; a=(6, 2)}  |
| 2 <sup>2</sup>   | t:=1;               | {t=1}            |
| 3 <sup>3</sup>   | s:=a[1];            | {s=6}            |
| 4 <sup>4</sup>   | i:=2;               | {i=2}            |
| 5 <sup>5</sup>   | while i ≤ n loop    | {2 ≤ 2}          |
| 6 <sup>6</sup>   | s:=s-a[i];          | {s=2-a[2]=6-2=4} |
| 7 <sup>7</sup>   | i:=i+1;             | {i=3}            |
| 5 <sup>8</sup>   | while i ≤ loop      | {3 ≤ 2}          |
| 9 <sup>9</sup>   | if s > t then       | { 4 > 1}         |
| 10 <sup>10</sup> | if s mod 2 ≠ 0 then | { 4 mod 2 ≠ 0}   |
| 14 <sup>11</sup> | put(s);             | { put(4) }       |

Figura 12.5 – Instância dos comandos executados pelo caso de teste  $n = 2$  e  $a = (6, 2)$ .

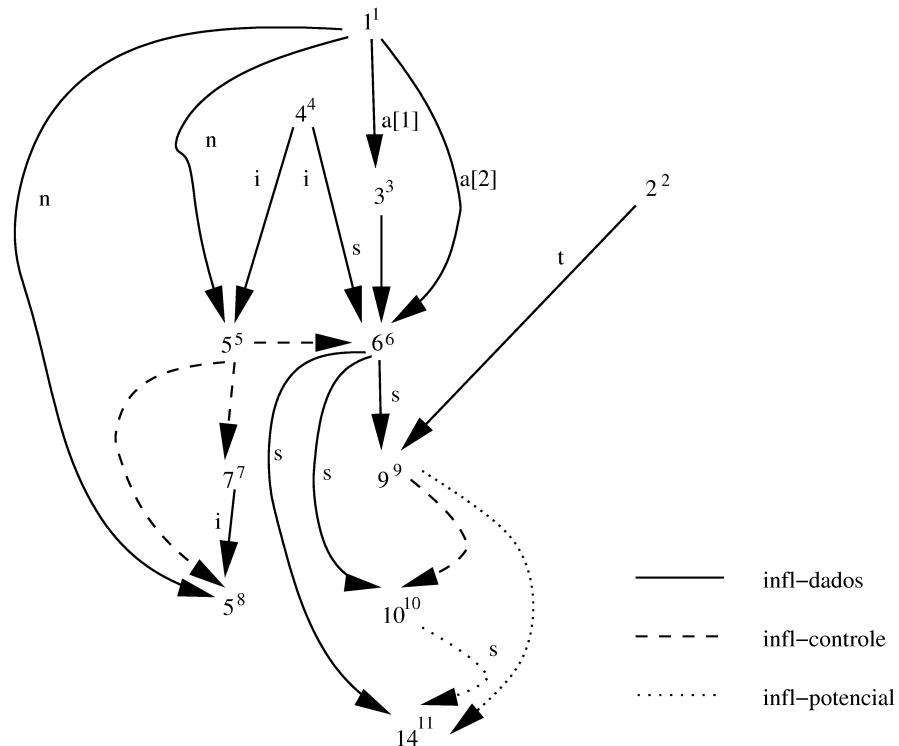


Figura 12.6 – Rede de dependência gerada pela ferramenta PELAS durante a execução do programa.

A estratégia de depuração implementada pelas ferramentas PELAS, Spyder, FIND e ALICE depende essencialmente da construção de um grafo em tempo de depuração que registra cada ponto de execução, bem como suas relações. A complexidade espacial para a construção do grafo é não-limitada, visto que depende do número de vezes que os coman-

dos do procedimento são executados. Essa questão é importante para programas com longas execuções, pois a memória disponível pode ser totalmente consumida [225].

Os grafos construídos pelas ferramentas PELAS, Spyder e FIND tinham inicialmente o objetivo de determinar fatias dinâmicas do programa. Novos algoritmos para a determinação de fatias dinâmicas que não dependem da construção do grafo foram desenvolvidos [225, 158]. Entretanto, essas soluções, por evitar a construção do grafo, perdem as instâncias dos comandos que influenciam determinado ponto da execução, o que impede a seleção de novos sintomas internos (pontos de execução) a serem investigados.

Some-se a isso o fato de que essas ferramentas baseiam-se em execução em reverso, o que compromete sua escalabilidade. Os novos algoritmos de fatiamento dinâmico e de execução em reverso, se confirmados como escaláveis para programas reais, poderão tornar a depuração algorítmica intraprocedimental viável.

A estratégia de depuração implementada nas ferramentas PELAS, Spyder, FIND e ALICE expandem a idéia de depuração algorítmica (originalmente utilizada para a determinação do procedimento defeituoso) para localização de defeitos internamente aos procedimentos. Portanto, ela apóia o Passo 4 do modelo DDT utilizando informação básica de teste.

### 12.3.5 Depuração delta

A depuração delta foi criada inicialmente por Zeller [454] com o objetivo de reduzir o tamanho da entrada de casos de teste que provocam a ocorrência de falhas. A idéia é determinar o dado de entrada diretamente responsável pela falha do programa. Um problema típico de utilização da depuração delta é determinar qual dos comandos HTML provoca a falha de carregamento de uma página HTML. A depuração delta soluciona essa questão fazendo o teste de carregamento de páginas que são resultantes da diferença entre a página que provocou a falha e uma página que não provocou (no caso, uma página vazia).

Considere uma página que contenha metade do texto da página que provoca a falha. Se a falha continua a ocorrer, então o trecho que induz à falha está na metade incluída. Se, por outro lado, a falha não ocorre mais, então o trecho omitido é o que contém a falha. O caminho inverso pode ser também percorrido partindo da página que não provoca a falha e incluindo trechos da página errônea. Ao repetir esse processo inúmeras vezes, pode-se chegar ao comando HTML que provoca a ocorrência da falha.

O algoritmo da depuração delta é inspirado na busca binária, porém, tratando situações em que a página carregada produz um resultado indeterminado. A depuração delta pode ser utilizada para identificar outras circunstâncias causadoras de falhas. Por exemplo, considere que um programa teve 10.000 linhas alteradas e uma falha ocorre quando executado por um determinado caso de teste. Qual dessas 10.000 linhas causa a falha? Fazendo a execução repetida de versões do programa com diferentes conjuntos de linhas modificadas e utilizando o algoritmo de depuração delta, podem-se identificar as linhas que dão origem à falha.

Zeller et al. [87, 454] propuseram a utilização da depuração delta para identificação de trechos do programa que causam a ocorrência da falha. Isso é realizado comparando-se o estado do programa em execuções que manifestam a falha e execuções que não manifestam. São feitas alterações no estado do programa utilizando o algoritmo de depuração delta para identificar os comandos que causam a manifestação da falha. Os resultados obtidos são pro-

missores, mas o custo para utilizar essa técnica em programas com longas execuções ainda é uma questão em aberto.

Do ponto de vista do modelo DDT, a depuração delta auxilia o Passo 2, pois ajuda no mapeamento das entradas em sintomas internos, seja por meio de simplificação da entrada, seja pela identificação de trechos do programa candidatos a conter o defeito. O tipo de informação utilizada é simples, daí a necessidade de repetidas execuções para obter informação útil para a depuração.

## 12.4 Comparação das técnicas e o modelo DDT

A Tabela 12.1 apresenta os passos do modelo DDT que cada técnica de depuração apóia, bem como o tipo de informação de teste que utiliza e sua escalabilidade para situações reais. O símbolo “✓” indica a presença da característica (Passo do modelo, escalabilidade), enquanto o símbolo “⊕” indica que a técnica possui indicações de escalabilidade que precisam ser confirmadas por novos experimentos.

Tabela 12.1 – Adequação das técnicas de depuração ao modelo DDT

| Técnica de Depuração                   | Passo do P. DDT |   |   | Inf. de Teste | Esc. |
|--|-----------------|---|---|---------------|------|
|  | 2               | 3 | 4 |               |      |
| Rastreamento e inspeção                | ✓               | ✓ |   | básica        | ✓    |
| Asserções                              | ✓               |   |   | básica        | ✓    |
| Fatiamento estático                    | ✓               |   | ✓ | básica        |      |
| Fatiamento dinâmico                    | ✓               |   | ✓ | básica        | ⊕    |
| Fatiamento informação de teste         | ✓               |   |   | detalhada     | ✓    |
| Heur. baseada em fatias estáticas      | ✓               |   | ✓ | básica        |      |
| Heur. baseada em fatias dinâmicas      | ✓               |   | ✓ | básica        | ⊕    |
| Heur. em fatias de informação de teste | ✓               |   |   | detalhada     | ✓    |
| Dep. baseada em requisitos de teste    | ✓               | ✓ | ✓ | detalhada     | ⊕    |
| Dep. algorítmica interprocedimental    | ✓               |   | ✓ | detalhada     |      |
| Dep. algorítmica intraprocedimental    | ✓               |   | ✓ | básica        | ⊕    |
| Depuração delta                        | ✓               |   |   | básica        |      |

A técnica de depuração baseada em rastreamento e inspeção é fundamental ao modelo DDT, pois a avaliação dos possíveis sintomas internos (Passo 3) é apoiada apenas pelas ferramentas que apóiam essa técnica. Entretanto, essas ferramentas utilizam informação básica de teste. Nesse sentido, elas não são completamente adequadas à depuração que ocorre depois do teste, visto que não tiram proveito da informação coletada durante essa atividade. No contexto do modelo DDT, as ferramentas de rastreamento e inspeção devem ser capazes de rastrear eventos relacionados com os resultados obtidos do teste sistemático para permitir a análise dos possíveis sintomas internos indicados por eles. Por isso, Chaim et al. [64] propõem alterações no depurador simbólico GDB para que seja possível rastrear eventos associados a requisitos de teste estrutural.

As técnicas de fatiamento estático e dinâmico de programas apóiam o mapeamento de falhas e erros para possíveis sintomas internos (Passo 2) e a seleção de novos possíveis sintomas (Passo 4). Todavia, essas técnicas possuem alguns problemas. Um dos problemas é o número grande de comandos suspeitos contidos nas fatias selecionadas. O uso de diferentes tipos de fatias [223, 224] e heurísticas [74, 322] pode diminuir esse problema. O segundo problema, relacionado com o custo de obtenção das fatias, pode ser resolvido pelos novos algoritmos de fatiamento dinâmico, caso se confirmem escaláveis para situações reais. O fato de não utilizar informação de teste, porém, torna as fatias estáticas e dinâmicas menos adequadas ao modelo DDT.

As fatias de programas obtidas de informação de teste são as mais adequadas ao modelo DDT, visto que são determinadas por meio da utilização de resultados coletados durante o teste, e o modelo preconiza o uso desse tipo de informação. A vantagem é que elas podem ser obtidas quase que diretamente durante a depuração, pois o custo de obtenção dos resultados de teste já foi amortizado. Os algoritmos para determinação das fatias são baratos e basicamente realizam o mapeamento da informação de teste para um trecho de código. Nesse sentido, os fatiamentos baseados em informação de teste são escaláveis para programas reais [436]. Entretanto, eles não apóiam a seleção de novos sintomas internos (Passo 4).

Esse problema é importante, visto que o trecho de código associado às fatias baseadas em informação de teste pode ser grande. A utilização de heurísticas diminui o tamanho do trecho de código a ser examinado, mas existe sempre a possibilidade de o programador ter sua atenção direcionada para um ponto do programa que não contém o defeito ou que indica os efeitos do defeito, e não o próprio defeito. Chaim et al. [64] propõem mecanismos para refinamento dos requisitos de teste selecionados por heurísticas em tempo de execução para superar esse problema. O custo desses mecanismos é pequeno, o que indica que eles podem ser escaláveis para programas reais.

As técnicas de depuração algorítmica, como as técnicas de fatiamento estático e dinâmico, apóiam os Passos 2 e 4. Contudo, tanto a depuração algorítmica interprocedimental quanto a intraprocedimental têm sua aplicabilidade restringida pelo grafo de tamanho não-limitado utilizado nessas técnicas, o que as torna inviáveis em um contexto industrial de produção de software.

A depuração delta é uma técnica promissora que permite simplificar condições causadoras de falha. O principal apelo dessa técnica é que, a partir de informações triviais como um caso de teste que apresenta uma falha, podem-se simplificar automaticamente circunstâncias causadoras de falhas como a entrada de um programa. Nesse sentido, é uma técnica eficaz de mapeamento de falhas em sintomas internos (Passo 2). No entanto, por utilizar um tipo de informação de teste simples, a depuração delta requer várias reexecuções do programa para obter informação útil para a depuração. A utilização da depuração delta para identificar trechos de programas candidatos a conter o defeito está sendo pesquisada, porém ela ainda não é escalável para programas reais.

A análise realizada permite constatar que o uso de informação detalhada de teste na depuração resulta em técnicas de baixo custo em tempo de depuração, desde que o teste tenha sido realizado de maneira sistemática. Nesse sentido, elas apresentam possibilidades de utilização em ambientes industriais que realizam teste sistemático. Todavia, apenas recentemente têm surgido técnicas que utilizam informação de teste para apoiar todos os passos do modelo DDT. Isto corrobora a afirmação de Harrold [166] de que o uso de informação de teste na depuração encontra-se ainda na infância.

## 12.5 Considerações finais

Neste capítulo, a atividade de depuração que ocorre depois do teste de software foi analisada. As principais técnicas de depuração de programas procedimentais foram avaliadas tendo como guia o modelo DDT e sua escalabilidade para sistemas reais. O modelo DDT ressalta os seguintes aspectos (passos): identificação de possíveis sintomas internos; avaliação dos sintomas identificados; apoio à seleção de novos sintomas; e tipo de informação de teste utilizada. Dessa avaliação, observou-se que os resultados do teste sistemático de software, quando utilizados na depuração, resultam em técnicas de baixo custo e com perspectivas de escalabilidade para programas reais. Estudos experimentais [202] mostram que técnicas baseadas em heurísticas e informação de teste são eficazes e eficientes em comparação com outras técnicas.

No entanto, vários problemas em depuração demandam esforços de pesquisa. A escalabilidade das técnicas ainda é uma questão relevante. É verdade que algoritmos promissores foram obtidos para técnicas como o fatiamento dinâmico de programas [456, 455]. O ponto, porém, é verificar se esses algoritmos são escaláveis em um ambiente de programação em que o programador invoca os algoritmos várias vezes para diferentes casos de teste. Outras técnicas, por sua vez, demandam soluções mais eficientes como a depuração delta, quando utilizada para selecionar um trecho de código. Outra questão a ser investigada é a combinação de técnicas de depuração. Recentemente, foi proposta a combinação de depuração delta e o fatiamento de programas [157]; outras combinações são possíveis e precisam ser avaliadas experimentalmente. Finalmente, é importante tornar disponíveis as informações de teste e de depuração de uma maneira útil ao programador. Técnicas de visualização de informação de teste e de depuração precisam ser desenvolvidas.

# Capítulo 13

## Confiabilidade

*Adalberto Nobaito Crespo (CENPRA, USF)  
Mario Jino (DCA/FEEC/UNICAMP)*

### 13.1 Introdução

Com o constante desenvolvimento da tecnologia, os sistemas computacionais têm sido requisitados em quase todas as áreas da atividade humana. Especificamente nos últimos anos, softwares específicos foram desenvolvidos para aplicações críticas, como sistemas de controle de usinas nucleares, sistemas de controle aeroespacial, controle de processos na área médica e muitas outras áreas de risco no campo industrial. A natureza de sistemas computacionais em termos de precisão de tempo e em termos do comportamento repetitivo faz do software a solução ideal para áreas nas quais um simples engano pode causar efeitos extremamente danosos.

Essa crescente dependência em relação ao software tem conscientizado tanto os usuários, que cada vez mais exigem softwares confiáveis, como também a indústria de software, no sentido de desenvolver produtos de alta qualidade. No entanto, além de freqüentemente o software constituir a parte mais dispendiosa para a solução de um problema que envolve o computador, desenvolver software com qualidade tem exigido um enorme esforço na atividade de teste e também tem sido uma tarefa extremamente difícil. A principal razão dessa dificuldade é que o software é uma entidade lógica, diferentemente de muitos outros sistemas em que os componentes têm alguma forma física para os quais um valor concreto de qualidade pode ser alcançado. Uma outra evidência dessa dificuldade é que o software é um artefato extremamente complexo que não pode ser analisado por formalismos matemáticos bem estruturados [258].

Nesse sentido, as evidências apontam a necessidade de pesquisas na área, tendo por fim também um melhor entendimento do que vem a ser a qualidade de software.

Em um contexto mais amplo, qualidade de software é uma propriedade multidimensional ainda difícil de ser medida e, conforme a norma ISO9126, é constituída pelas características: funcionalidade, confiabilidade, eficiência, portabilidade, usabilidade e manutenibilidade. Todavia, a confiabilidade, ao contrário de outras características, é comumente aceita como um

fator-chave da qualidade, uma vez que pode ser medida e estimada usando dados históricos, qualificando, assim, as falhas do software.

A confiabilidade é uma característica que tem sido extensivamente considerada na análise da qualidade do software, pois se um software não é confiável, pouco importa se outras características da qualidade são aceitáveis. Por outro lado, medir a confiabilidade de um software tem-se mostrado uma tarefa desafiadora.

A preocupação com a confiabilidade de software teve início por volta de 1967 com Hudson [193]. A partir dos anos 70, fundamentados na teoria sobre confiabilidade de hardware, surgiram os primeiros estudos e os primeiros modelos de confiabilidade de software [199, 356]. Na década de 1980 ampliaram-se os estudos e surgiram vários outros modelos.

Atribuir-se um grau de confiabilidade ao software, o objetivo é quantificar alguns aspectos desse software que, devido à presença inevitável de defeitos, está sujeito a falhas durante seu período de utilização. Assim, o estudo da confiabilidade de software caracteriza-se como uma abordagem analítica que está baseada nos conceitos de métricas, medidas e modelos.

A confiabilidade representa a qualidade do ponto de vista do usuário. Sabe-se que uma confiabilidade de 100%, mesmo para programas de baixo nível de complexidade, é praticamente impossível de ser obtida. Do ponto de vista das organizações que desenvolvem software, a confiabilidade é uma referência para a avaliação do software. Nesse contexto, a indústria analisa o software para garantir que o produto atingiu um certo nível de confiabilidade como um critério para sua liberação. Para isso, o teste é extensivamente conduzido tanto para remover defeitos quanto para determinar o nível de confiabilidade.

De uma maneira geral, um sistema de software é dito confiável se desempenha corretamente suas funções especificadas por um longo período de execução e em uma variedade de ambientes operacionais [155, 292]. Essencialmente, existem três maneiras de se alcançar um alto nível de confiabilidade:

- evitar a introdução de defeitos durante o projeto e o desenvolvimento dos programas;
- fazer uso de estruturas tolerantes a defeitos; e
- remover defeitos durante a fase de teste e depuração.

As duas primeiras maneiras enquadram-se na abordagem construtiva: consistem em desenvolver software utilizando os métodos, as técnicas e as ferramentas disponíveis. Finalmente, a confiabilidade do software pode ser melhorada pelo teste e depuração, abordagem em que o software é submetido a um intenso processo dinâmico que objetiva a detecção e a remoção dos defeitos restantes.

A primeira consideração na análise de confiabilidade de software é como medir a confiabilidade. Várias métricas foram propostas para responder essa questão. Alguns autores sugerem associar a confiabilidade diretamente ao número de defeitos restantes no software. Assim, a atividade de teste de software é um método primário para se obter a medida de confiabilidade. Littlewood e Verall [245] discutiram a impossibilidade de uma definição rigorosa do conceito de defeitos no software e, dessa forma, defendem uma abordagem de modelos de falhas de software. Em geral, muitos artigos sobre confiabilidade têm considerado esse problema e modelado as falhas do software em vez de defeitos no software.

Para tanto, torna-se necessária a coleta de dados de falhas. Essa coleta compreende: 1) contagem de falhas para o rastreamento da quantidade de falhas observadas por unidade de tempo; 2) tempo médio entre falhas que faz o rastreamento dos intervalos entre falhas consecutivas. De posse desses dados, a engenharia de confiabilidade de software pode desenvolver atividades de estimação e previsão.

Para melhor entendimento do significado de confiabilidade consideremos, como exemplo, um automóvel. Por melhor que seja o processo de fabricação, o carro deve passar por períodos de manutenção para reparos de funcionalidades ou substituição de peças desgastadas. Dessa maneira, o carro é considerado confiável se, por longos períodos, apresentar um comportamento desejável e consistente entre os períodos de manutenção.

Em relação ao software, a confiabilidade é definida como a probabilidade de que o software não falhe em um dado intervalo de tempo, em um dado ambiente [292]. Logo, é uma medida importante para decidir sobre a liberação do software. A probabilidade de ocorrência de falha serve também como um preditor útil da confiabilidade corrente para o software em operação. Um software é considerado altamente confiável quando pode ser utilizado sem receio em aplicações críticas. O contínuo crescimento de tamanho e complexidade do software projetado atualmente torna a confiabilidade o aspecto indiscutivelmente mais importante de qualquer sistema de software.

Apesar de se utilizar o carro como exemplo para o entendimento da confiabilidade, a confiabilidade de software é completamente diferente da confiabilidade de hardware. Ao contrário do hardware, o software não envelhece e não sofre desgaste por ação do uso ou do tempo. No hardware os defeitos são causados por peças de baixa qualidade ou por desgastes naturais das peças. No software os defeitos podem ser introduzidos nas várias fases do desenvolvimento e são causados principalmente por problemas no projeto. A confiabilidade do hardware pode ser aumentada por substituição de material de melhor qualidade e práticas de projeto mais aperfeiçoadas. O crescimento da confiabilidade do software resulta da descoberta e da eliminação de defeitos no software obtidos pela aplicação de um teste intensivo e de qualidade.

Devido a essas diferenças, a confiabilidade de hardware é tratada de maneira diferente da confiabilidade de software. Quando um hardware é reparado, ele retorna ao seu nível de confiabilidade anterior, ou seja, a confiabilidade do hardware é mantida. Entretanto, quando um software é reparado, a sua confiabilidade pode tanto aumentar como diminuir, caso novos defeitos sejam inseridos durante o reparo. Assim, o objetivo da engenharia de confiabilidade de hardware é manter a estabilidade, enquanto o objetivo da engenharia de confiabilidade de software é melhorar a confiabilidade.

## 13.2 Fundamentos de confiabilidade de software

A ocorrência de falhas em um software é um evento totalmente imprevisível e, desse modo, deve ser considerada um processo aleatório. Por isso, o estudo da confiabilidade de um sistema resume-se em aplicar a teoria de probabilidades na modelagem do processo de falhas do sistema e na predição da probabilidade de ocorrência de um sucesso. Em relação a software, a ocorrência do evento “sucesso” é entendida como a execução do software com um dado de entrada sem que ocorram falhas.

A teoria de confiabilidade está estreitamente ligada à teoria de probabilidades, e, assim, a confiabilidade é uma característica da qualidade do software que pode ser expressa por meio de números entre 0 (não confiável) e 1 (totalmente confiável). Nesse contexto, para se atribuir a confiabilidade, o software deve ser examinado em termos de falhas. O número de defeitos no software que provocaram as falhas determina uma confiabilidade estável no software. À medida que os defeitos são detectados e removidos, sem a inserção de novos defeitos, o software passa a ter uma menor intensidade de falhas e, por consequência, há um aumento na confiabilidade.

Considerando a ocorrência de falhas no software como um evento aleatório, a atribuição da confiabilidade está associada ao estudo de algumas variáveis aleatórias do processo de falhas. Como exemplo, a confiabilidade de um software pode estar associada ao tempo médio entre a ocorrência das falhas. Nesse caso, para a atribuição da confiabilidade do software, a variável aleatória de interesse é o tempo  $T$  decorrido para a ocorrência de uma falha.

A confiabilidade informa se o software está sendo aperfeiçoado (fallando cada vez menos), na medida em que se detectam e removem os defeitos. Se os tempos entre as falhas permanecem os mesmos, então a confiabilidade do software continua estável. Se o tempo entre as falhas aumenta, então tem-se um crescimento na confiabilidade do software.

### 13.2.1 Definições

A teoria sobre confiabilidade de software lida com métodos probabilísticos aplicados para analisar a ocorrência aleatória de falhas em um dado sistema de software. Nesta seção são apresentados alguns conceitos que formalizam a teoria da confiabilidade, tais como: função Confiabilidade, função Taxa de Falhas, função de Falhas Acumuladas, função Intensidade de Falhas, Tempo Médio para Falhas (MTTF) e Tempo Médio entre Falhas (MTBF).

No estudo sobre confiabilidade de software, a variável aleatória de interesse pode ser o tempo  $T$  decorrido para a ocorrência de uma falha do software. Por consequência, a base dos resultados sobre confiabilidade estará relacionada ao estudo da variável aleatória  $T$ , mais especificamente, ao estudo das funções associadas à variável aleatória  $T$ , tais como função densidade de probabilidade  $f(t)$  e função distribuição de probabilidades acumuladas  $F(t)$ .

A probabilidade de que haja uma falha no software em decorrência do tempo  $t$  é definida como:

$$F(t) = P[0 \leq T \leq t] = \int_0^t f(x)dx \quad (13.1)$$

em que  $f(t)$  é a função densidade de probabilidade e  $F(t)$  é a função distribuição de probabilidades, da variável aleatória  $T$ . Por consequência, a probabilidade de que não ocorra falha no software até o tempo  $t$  é definida como:

$$1 - F(t) = 1 - P[0 \leq T \leq t] = P[T > t]$$

## Função Confiabilidade

Na literatura clássica, a definição sobre confiabilidade amplamente adotada é: probabilidade de operação livre de falhas de um software, em um tempo e ambiente especificados [292].

Assim, a função Confiabilidade, também chamada de função de sobrevivência de um software, é definida como:

$$R(t) = P[T > t] = 1 - F(t) = \int_t^{+\infty} f(x)dx \quad (13.2)$$

Quando uma base de tempo é determinada, as falhas no software podem ser expressas por várias funções, como: função Taxa de Falhas, função de Falhas Acumuladas, função Intensidade de Falhas, tempo médio para falhas (MTTF) e tempo médio entre falhas (MTBF).

Para determinar o comportamento de falhas no software, basta observar o comportamento de uma dessas funções. Ou seja, para determinar a forma funcional do modelo que representa o comportamento das falhas no software, é só determinar a forma funcional de apenas uma dessas funções. O comportamento das demais funções pode ser explicitamente determinado. Alguns modelos de confiabilidade de software são determinados por suposições no comportamento da taxa de falhas, outros são determinados por suposições no comportamento da função de Falhas Acumuladas.

## Função Taxa de Falhas

Para descrever o ritmo de ocorrência das falhas em um sistema, a taxa de falhas é um conceito bastante utilizado. Teoricamente, a taxa de falhas é definida como a probabilidade de que uma falha por unidade de tempo ocorra num intervalo  $[t, t + \Delta t]$ , dado que o sistema não falhou até o tempo  $t$ . Na prática, a taxa de falhas é a razão entre o incremento do número de falhas e o incremento de tempo correspondente. Assim, usando a probabilidade condicional, tem-se que:

$$\begin{aligned} \text{Taxa de falhas} &\equiv \frac{P[t \leq T \leq t + \Delta t | T > t]}{\Delta t} = \frac{P[t \leq T \leq t + \Delta t]}{\Delta t P[T > t]} \\ &= \frac{F(t + \Delta t) - F(t)}{\Delta t R(t)} \end{aligned}$$

A taxa de falhas instantânea,  $Z(t)$ , também conhecida como taxa de risco associada à variável aleatória  $T$ , é definida como:

$$Z(t) = \lim_{\Delta t \rightarrow 0} \frac{F(t + \Delta t) - F(t)}{\Delta t R(t)} = \frac{f(t)}{R(t)}$$

Ou seja, a taxa de risco é definida como o limite da taxa de falhas quando o intervalo  $\Delta t$  tende a zero ( $\Delta t \rightarrow 0$ ).

A taxa de risco é uma taxa de falhas instantânea no tempo  $t$ , dado que o sistema não falhou até o tempo  $t$ . Embora haja uma pequena diferença entre a taxa de falhas e a taxa de risco, normalmente usa-se  $Z(t)$  como taxa de falhas [255].

As funções  $f(t)$ ,  $F(t)$ ,  $R(t)$ , e  $Z(t)$  matematicamente fornecem especificações equivalentes sobre a distribuição da variável aleatória  $T$  e podem ser derivadas uma da outra por simples operações algébricas.

Observa-se que:

$$Z(t) = \frac{f(t)}{R(t)} = \frac{dF(t)}{dt} \frac{1}{R(t)} \quad (13.3)$$

Observa-se também que da Equação (13.2) pode-se obter:

$$\frac{dF(t)}{dt} = -\frac{dR(t)}{dt} \quad (13.4)$$

Substituindo a Equação (13.4) na Equação (13.3), tem-se:

$$-Z(t) = \frac{dR(t)}{dt} \frac{1}{R(t)} \quad (13.5)$$

Integrando a Equação (13.5) em relação a  $t$  em ambos os lados, obtém-se:

$$-\int_0^t Z(x)dx + c = \int \frac{dR(t)}{R(t)} \Rightarrow \ln R(t) = -\int_0^t Z(x)dx + c$$

Da Equação (13.2) tem-se a condição inicial que  $R(0) = 1$ , e portanto  $c = 0$ . Logo, a relação da função confiabilidade  $R(t)$  com a taxa de falhas  $Z(t)$  é dada por:

$$R(t) = e^{-\int_0^t Z(x)dx} \quad (13.6)$$

Utilizando a Equação (13.3) tem-se que a relação entre a função densidade de probabilidade  $f(t)$  e a taxa de falhas  $Z(t)$  é dada por:

$$f(t) = Z(t)e^{-\int_0^t Z(x)dx} \quad (13.7)$$

A taxa de falhas se altera ao longo do tempo de vida do sistema. A Figura 13.1 ilustra o comportamento da taxa de falhas de alguns sistemas.

A Região I, conhecida como fase de depuração, representa as falhas iniciais do sistema. Nessa região a taxa de falhas tende a decrescer com o tempo.

A Região II, conhecida como período de vida útil do sistema ou fase de operação, representa as falhas causadas por eventos aleatórios ou por condições de *stress* do sistema. Nessa região a taxa de falhas permanece constante com o tempo.

A Região III representa a fase de desgaste do sistema, caracterizada pelo crescimento na taxa de falhas em função do tempo.

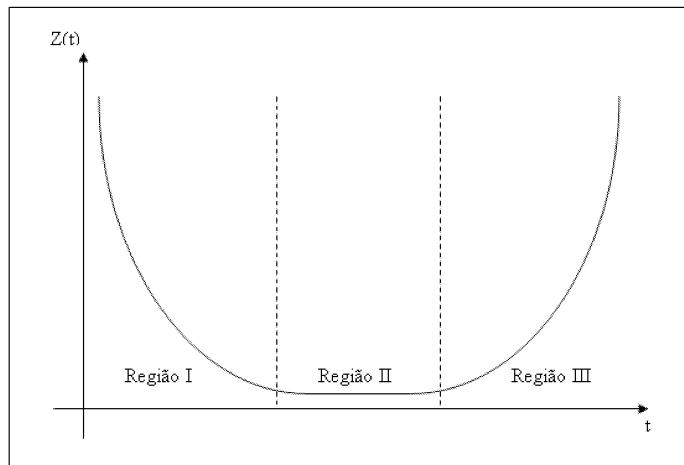


Figura 13.1 – Taxa de falhas típica de um sistema.

A confiabilidade de software é semelhante à confiabilidade de hardware, tendo em vista que ambas são processos probabilísticos e podem ser descritas por distribuições de probabilidades. Contudo, a confiabilidade de software é diferente da confiabilidade de hardware no sentido de que o software não se desgasta com o tempo, ou seja, a confiabilidade não decresce com o tempo. Conseqüentemente, a Região III não se aplica à confiabilidade de software. No software, geralmente, a confiabilidade cresce na fase de teste e na fase de operação, desde que as falhas sejam removidas quando detectadas. No entanto, pode ocorrer um decréscimo na confiabilidade devido a alterações abruptas no ambiente de operação do sistema ou modificações incorretas na manutenção.

Considerando-se que o software é constantemente modificado em seu ciclo de vida, torna-se inevitável a consideração de que a taxa de falhas seja variável.

Diferentemente dos defeitos de hardware que, na maioria das vezes, são defeitos físicos, os defeitos de software são defeitos de projeto, difíceis de visualizar, classificar, detectar e remover. Como resultado, a confiabilidade de software é muito mais difícil de se medir e analisar. Normalmente, a teoria de confiabilidade de hardware está fundamentada na análise de processos estacionários porque somente os defeitos físicos são considerados. Contudo, com o crescimento da complexidade dos sistemas e a introdução de defeitos no projeto, a teoria de confiabilidade de software baseada em processos estacionários torna-se inconveniente. Isso torna a confiabilidade de software um problema desafiante que requer o emprego de vários métodos.

A taxa de falhas ideal para o software é decrescente. A Figura 13.2 dá uma idéia de seu comportamento em função do tempo.

Ao se observar o comportamento da taxa de falhas em algum ponto, por meio de evidência estatística, é possível prever o comportamento da taxa de falhas em um tempo futuro. Com isso, os modelos de confiabilidade de software podem prever o tempo adicional necessário

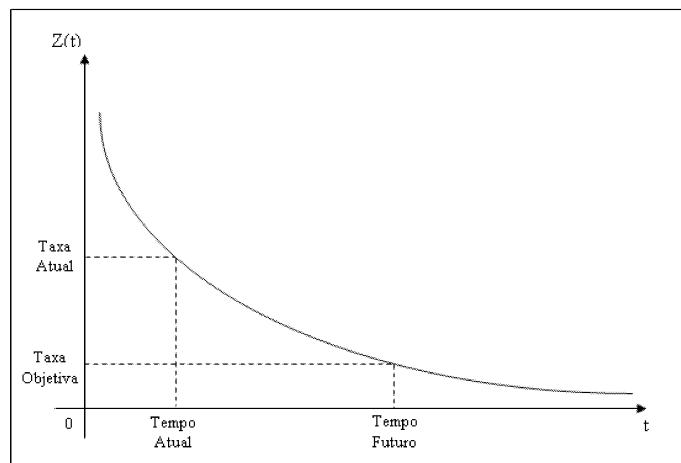


Figura 13.2 – Taxa de falhas do software.

para o teste do software até que se atinja o objetivo especificado – a taxa de falhas desejada. Pode-se também estimar a confiabilidade ao término do teste.

Quando se considera crescimento de confiabilidade, uma medida usual é a confiabilidade condicional ([154, 293]). Dado que o sistema teve  $n - 1$  falhas, a confiabilidade condicional é a função de sobrevivência associada à  $n$ -ésima falha do sistema. A confiabilidade condicional é de interesse quando o sistema está em fase de desenvolvimento, período em que se observa o tempo para a próxima falha. Quando o sistema está liberado e em fase operacional, o interesse passa a ser o intervalo de tempo livre de falhas e, nesse caso, os instantes de falha não são necessariamente condicionados às falhas anteriores. O interesse é a confiabilidade em um dado intervalo de tempo, independentemente do número de falhas ocorridas anteriormente.

A Tabela 13.1 ilustra as relações existentes entre a função densidade de probabilidade  $f(t)$ , a função de distribuição de probabilidades acumulada  $F(t)$ , a função confiabilidade  $R(t)$  e a função taxa de falhas  $Z(t)$ .

Seja  $T'_i (i = 1, 2, 3, \dots)$  a variável aleatória que representa o  $i$ -ésimo intervalo de tempo entre falhas e seja  $T_i (i = 1, 2, 3, \dots)$  a variável aleatória que representa o  $i$ -ésimo tempo de falha. A Figura 13.3 ilustra o comportamento dessas variáveis aleatórias, mostrando que  $T_i = \sum_{j=1}^i T'_j = T_{i-1} + T'_i$  para  $i = 1, 2, 3, \dots$ , e  $T_0 = 0$ .

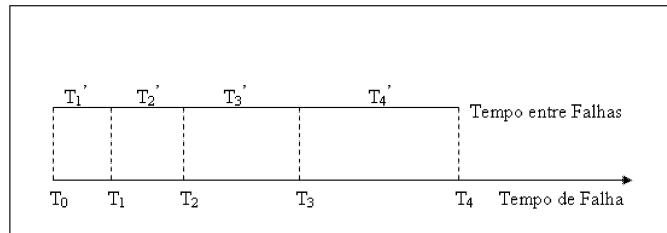
Assim, a confiabilidade condicional é definida como:

$$R(t'_i | t_{i-1}) = P[T'_i > t'_i | T_{i-1} = t_{i-1}]$$

As relações anteriores mostradas na Tabela 13.1 são também válidas para a confiabilidade condicional.

Tabela 13.1 – Relações entre as Funções  $f(t)$ ,  $F(t)$ ,  $R(t)$  e  $Z(t)$ 

|        | $f(t)$                                 | $F(t)$                              | $R(t)$                             | $Z(t)$                     |
|--------|--|-------------------------------------|------------------------------------|----------------------------|
| $f(t)$ | -                                      | $\frac{dF(t)}{dt}$                  | $-\frac{dR(t)}{dt}$                | $Z(t)e^{-\int_0^t Z(x)dx}$ |
| $F(t)$ | $\int_0^t f(x)dx$                      | -                                   | $1 - R(t)$                         | $1 - e^{\int_0^t Z(x)dx}$  |
| $R(t)$ | $\int_0^{+\infty} f(x)dx$              | $1 - F(t)$                          | -                                  | $e^{-\int_0^t Z(x)dx}$     |
| $Z(t)$ | $\frac{f(t)}{\int_t^{+\infty} f(x)dx}$ | $\frac{dF(t)}{dt} \frac{1}{1-F(t)}$ | $-\frac{dR(t)}{dt} \frac{1}{R(t)}$ | -                          |


 Figura 13.3 – Variáveis aleatórias Tempo de Falha ( $T_i$ ) e Tempo entre Falhas ( $T'_i$ ).

## Função de Falhas Acumuladas

A função de Falhas Acumuladas, também denominada função Valor Médio, é uma maneira alternativa de caracterizar a ocorrência aleatória das falhas em um software. A função de Falhas Acumuladas descreve o crescimento da curva de Falhas Acumuladas e, assim, considera o número de falhas ocorridas no software até um tempo  $t$ . A função Valor Médio é uma maneira alternativa de representar o processo de falhas no software, e, assim, o comportamento futuro pode ser estimado por uma análise estatística do comportamento dessa curva de crescimento.

Teoricamente, o número de falhas acumuladas até o tempo  $t$  pode ser representado por uma variável aleatória  $M(t)$  com um valor médio  $\mu(t)$ . Isto é,  $\mu(t)$  representa a função valor médio do processo aleatório.

Dessa forma, tem-se que:

$$\mu(t) = E[M(t)]$$

em que  $E$  representa a média da variável aleatória  $M(t)$ .

A Figura 13.4 representa um comportamento típico da função Valor Médio.

O processo aleatório pode ser completamente especificado, assumindo-se uma distribuição de probabilidades para a variável aleatória  $M(t)$  para algum  $t$ .

Como  $M(t)$  assume somente valores inteiros, as correspondentes distribuições de probabilidades devem ser do tipo discretas. As distribuições de probabilidades Poisson e Binomial são bastante utilizadas para descrever o processo aleatório  $M(t)$ .

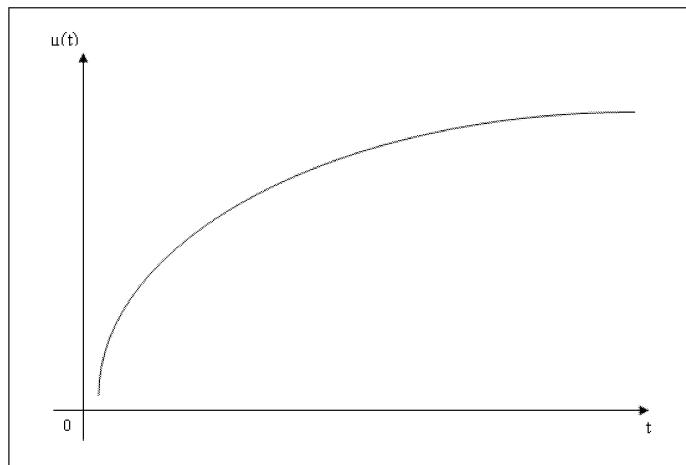


Figura 13.4 – Função Valor Médio.

Assim, considerando as distribuições mais utilizadas, pode-se calcular a probabilidade de que o processo tenha um certo número  $k$  de falhas acumuladas em determinado tempo  $t$ , da seguinte forma:

Se a distribuição de probabilidades assumida for a de Poisson, tem-se

$$P[M(t) = k] = \frac{[\mu(t)]^k}{k!} e^{-\mu(t)}$$

Se a distribuição de probabilidades assumida for a Binomial, tem-se

$$P[M(t) = k] = \binom{N}{k} p(t)^k [1 - p(t)]^{N-k}$$

em que  $N$  é o número de defeitos no software e  $p(t)$  é a probabilidade de falhas. Nesse caso, tem-se que  $E[M(t)] = Np(t)$ .

A função Valor Médio  $\mu(t)$  deve ser não decrescente com  $t$ . Alguns modelos de confiabilidade de software admitem uma função valor médio  $\mu(t)$  limitada. Outros modelos admitem a suposição de que na remoção de um defeito existe a possibilidade da introdução de novos defeitos e, assim, admitem que a função Valor Médio  $\mu(t)$  seja ilimitada.

A Tabela 13.2 ilustra as principais distribuições de probabilidades do número acumulado de falhas, utilizadas em confiabilidade de software.

Tabela 13.2 – Distribuição do número acumulado de falhas no tempo  $t$ 

|  | Poisson                           | Binomial                               |
|--|-----------------------------------|--|
| $E[M(t)]$  | $\mu(t)$                          | $Np(t)$                                |
| $P[M(t) = k]$  | $\frac{\mu(t)^k}{k!} e^{-\mu(t)}$ | $\binom{N}{k} p(t)^k [1 - p(t)]^{N-k}$ |
| $\mu(t)$ : Função Valor Médio do processo de falhas<br>$p(t)$ : Probabilidade de ocorrência da falha |                                   |  |

## Função Intensidade de Falhas

A função Intensidade de Falhas  $\lambda(t)$  representa a taxa de variação instantânea da função Valor Médio. Isto é, representa o número de falhas ocorridas por unidade de tempo. Por exemplo, pode se dizer que houve 0,01 falha/hora ou então que houve uma falha a cada 100 horas.

A função Intensidade de Falhas representa a derivada da função Valor Médio e é um valor instantâneo. A Figura 13.5 ilustra o comportamento da função Intensidade de Falhas.

Observa-se que no início do teste do software a ocorrência de falhas é maior em uma unidade de tempo e, consequentemente, a função de Falhas Acumuladas ou função Valor Médio cresce rapidamente. À medida que o teste prossegue, o número de falhas por unidade de tempo tende a diminuir, ou seja, a função Valor Médio cresce mais lentamente. A taxa de variação da função Valor Médio tende a decrescer rapidamente conforme o teste procede. Com isso, a função Intensidade de Falhas tem um comportamento decrescente no tempo.

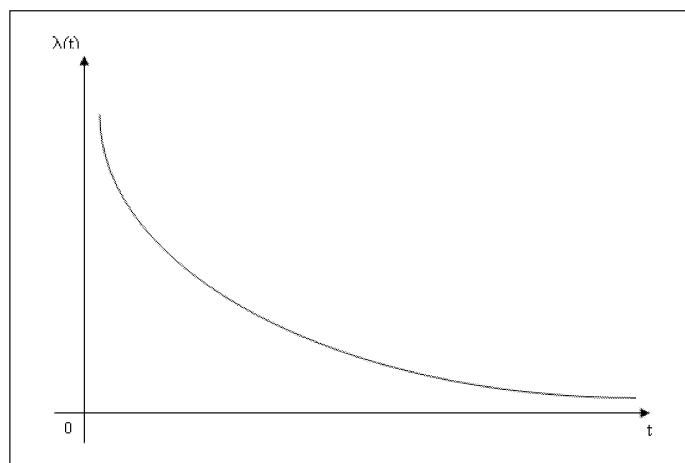


Figura 13.5 – Função Intensidade de Falhas.

A função Intensidade de Falhas  $\lambda(t)$  é, então, obtida da função Valor Médio como:

$$\lambda(t) = \frac{d\mu(t)}{dt} = \frac{d(E[M(t)])}{dt} \quad (13.8)$$

Para que haja crescimento da confiabilidade, é necessário que  $\frac{d\lambda(t)}{dt} < 0$  para qualquer  $t \geq t_0$  para algum  $t_0$ .

## Tempo Médio para Falhas

A função Tempo Médio para Falhas, MTTF (*Mean Time to Failure*), representa o tempo esperado para a ocorrência da próxima falha, ou seja, é o tempo durante o qual o software funciona sem falhas. O MTTF é uma medida que pode ser utilizada para caracterizar o modelo de falhas de um sistema de software.

Suponha que um software esteja em teste e que tenham sido encontradas  $i - 1$  falhas. Registrando-se os tempos entre as falhas como  $t_1, t_2, t_3, \dots, t_{i-1}$ , a média desses valores é o tempo médio antes de ocorrer a próxima falha. Um MTTF de 500 significa que uma falha pode ser esperada a cada 500 unidades de tempo.

Considere que cada defeito identificado tenha sido corrigido e que o sistema esteja novamente em execução. Pode-se utilizar  $T_i$  para denotar o tempo antes da próxima falha, que ainda será observada.  $T_i$  é uma variável aleatória e, quando se fazem declarações sobre a confiabilidade do software, fazem-se declarações de probabilidades sobre  $T_i$ .

De uma maneira geral, se  $T$  é uma variável aleatória que representa o tempo para a ocorrência da próxima falha com uma distribuição de probabilidades, então o tempo médio para falhas pode ser calculado como:

$$MTTF = E[T] = \int_0^{\infty} tf(t)dt \quad (13.9)$$

em que  $f(t)$  é a função Densidade de Probabilidades da variável aleatória  $T$ .

### 13.2.2 Medição da confiabilidade

Um meio simples de se medir a confiabilidade de um software é observar o tempo para a ocorrência da próxima falha. Certamente existe uma relação entre a confiabilidade do software e o tempo médio para a ocorrência da próxima falha. Pode-se imaginar que, à medida que o software se torna mais confiável, o tempo médio para a ocorrência da próxima falha tende a aumentar. Da mesma forma, se os tempos para a ocorrência da próxima falha são pequenos, isso significa que o software está falhando bastante e, por consequência, sua confiabilidade é baixa. Por outro lado, se os tempos para ocorrência da próxima falha são grandes, o software falha pouco e, por consequência, sua confiabilidade é alta. Logicamente, nessa maneira simples de medir a confiabilidade de um software, não se dá a devida atenção às suposições no comportamento das falhas no software.

Nesse contexto, o MTTF é próximo de zero quando a taxa de falhas do software é grande e próximo de 1 quando a taxa de falhas do software é pequena. Utilizando essa relação, pode-se calcular a medida da confiabilidade de um software como:

$$C = \frac{MTTF}{1 + MTTF}$$

Em um processo de teste do software, à medida que ocorrem as falhas e removem-se os defeitos, a confiabilidade é uma medida que informa se o software está sendo ou não aperfeiçoado. Se os tempos entre as falhas permanecem os mesmos, então tem-se uma confiabilidade estável. Se os tempos entre as falhas aumentam, então tem-se um crescimento da confiabilidade do software.

A confiabilidade do software também pode ser medida quando se conhece a distribuição de probabilidades de ocorrência das falhas. Isto é, quando o comportamento da ocorrência das falhas pode ser descrito com uma função Densidade de Probabilidade  $f(t)$ . Conhecida a função  $f(t)$ , a confiabilidade pode ser calculada pela Equação (13.2).

Pode-se desejar, também, saber a probabilidade de que o software opere sem falhas em um intervalo de tempo  $[t_1, t_2]$ . Assim, a confiabilidade do software nesse intervalo pode ser calculada como:

$$C_{t_1}^{t_2} = 1 - P[t_1 < T < t_2] = \int_{t_2}^{t_1} f(t)dt$$

Pode ser mostrado que:

$$C_{t_1}^{t_2} = R(t_2) - R(t_1)$$

### 13.2.3 Funções de confiabilidade

A modelagem sobre a confiabilidade de software normalmente envolve a utilização de uma distribuição de probabilidades do tempo para ocorrência das falhas no software. Dependendo do conjunto de suposições, a função de confiabilidade do modelo pode envolver uma distribuição de probabilidades desconhecida sobre o tempo de falhas no software. Contudo, de uma maneira geral, a maioria dos modelos é fundamentada em distribuições de probabilidades conhecidas na literatura [286].

Assim, nesta subseção são apresentadas as principais distribuições de probabilidades utilizadas em modelos de confiabilidade de software.

#### Distribuição exponencial

A distribuição exponencial é a distribuição de probabilidades mais conhecida e mais amplamente utilizada devido à sua simplicidade e grande aplicabilidade. A exponencial é um bom modelo de distribuição de probabilidades para descrever o tempo  $T$  entre a ocorrência de dois eventos consecutivos.

As funções Densidade de Probabilidade  $f(t)$  e Distribuição de Probabilidade  $F(t)$  têm a forma:

$$f(t) = \lambda e^{-\lambda t}$$

e

$$F(t) = 1 - \lambda e^{-\lambda t}$$

em que  $t > 0$  e  $\lambda > 0$ ,  $\lambda$  é um parâmetro. A Figura 13.6 ilustra o comportamento da função Densidade de Probabilidades Exponencial.

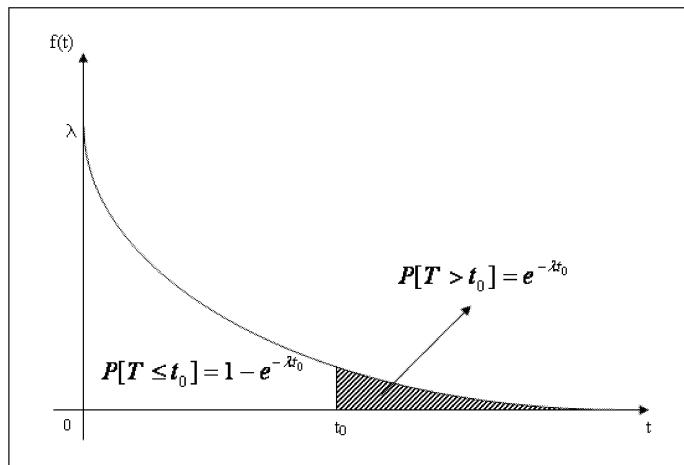


Figura 13.6 – Função Densidade de Probabilidades Exponencial.

A região hachurada na Figura 13.6 representa a probabilidade de não ocorrência de falhas até o tempo  $t_0$ . Observa-se que para pequenos intervalos de tempo é grande a probabilidade de não ocorrência de falhas. Por outro lado, para grandes intervalos de tempo é pequena a probabilidade de não ocorrência de falhas.

A função Confiabilidade pode ser obtida da Equação (13.2) e tem a forma:

$$R(t) = \int_t^{+\infty} \lambda e^{-\lambda x} dx = e^{-\lambda t}$$

A função Taxa de Falhas pode ser obtida da Equação (13.3) e tem a forma:

$$Z(t) = \frac{f(t)}{R(t)} = \lambda$$

O emprego da distribuição exponencial deve ter a hipótese de que o sistema tem uma taxa de falhas constante. Seu uso é recomendado quando o software tem uma taxa de falhas constante, isto é, na Região II da Figura 13.1 anteriormente discutida.

Pela Equação (13.9), o tempo médio para falhas tem a forma:

$$MTTF = E[T] = \int_0^{\infty} tf(t) dt = \frac{1}{\lambda}$$

Para uma certa classe de modelos de confiabilidade de software, a suposição é que o número de falhas acumuladas em um tempo  $t$  segue uma distribuição de Poisson, ou seja, é regida por um processo de Poisson [286], e que o tempo entre falhas segue uma distribuição exponencial. Dessa forma, muitos modelos de confiabilidade de software admitem que o tempo de falha do software ou então o tempo entre falhas segue uma distribuição exponencial.

Matematicamente, isso significa que se  $\lambda$  for a taxa de falhas, isto é,  $Z(t) = \lambda$ , e  $P[T < t]$ , a probabilidade de que o tempo de falha seja menor do que  $t$ , então tem-se que:

$$F(t) = P[T < t] = 1 - e^{-\lambda t}$$

para  $t > 0$ .

A Figura 13.7 ilustra a distribuição de probabilidades do tempo entre falhas para dois valores de taxa de falhas  $\lambda_1$  e  $\lambda_2$  com  $\lambda_1 < \lambda_2$ .

Ambas as curvas da Figura 13.7 aproximam-se da linha horizontal  $F(t) = 1$  à medida que o tempo  $t$  aumenta. Para valores maiores de  $\lambda$  (taxa de falhas) a curva aproxima-se da unidade mais rapidamente e para valores menores de  $\lambda$  a curva aproxima-se mais lentamente.

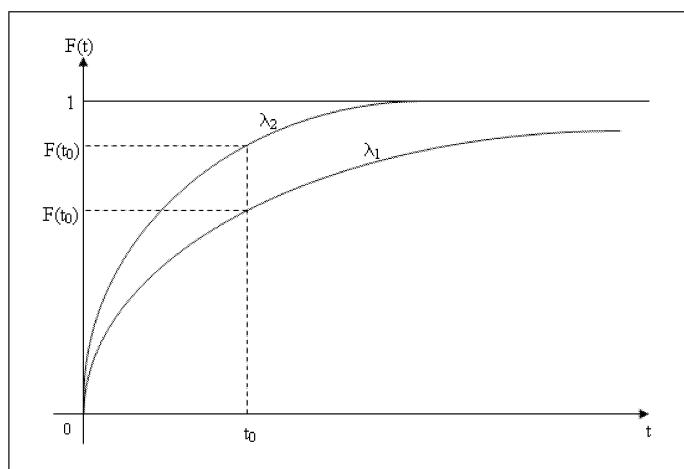


Figura 13.7 – Distribuição de probabilidades do Tempo entre Falhas.

Na Figura 13.7 observa-se que a probabilidade de ocorrência de falhas até um tempo  $t_0$  é maior quando a taxa de falhas é maior.

Para exemplificar, considere-se uma situação na qual o tempo é medido em horas e  $\lambda = 0,5$ . Isso significa que há a ocorrência de uma falha a cada duas horas, ou melhor, 0,5 falha por hora. A distribuição de probabilidades que representa essa situação tem a forma:

$$F(t) = P[T < t] = 1 - e^{-0,5t}$$

A Tabela 13.3 ilustra os valores dessa distribuição de probabilidades no tempo  $t$  em horas.

Ressalta-se que  $\lambda = 0,5$  não significa que as falhas ocorrem regularmente a cada duas horas, mas que ocorrem aleatoriamente na média de uma falha a cada duas horas. Em outras palavras,  $\lambda = 0,5$  é o valor médio da variável aleatória tempo  $T$  entre falhas, isto é,  $E[T] = 0,5$ .

Tabela 13.3 – Função Densidade de Probabilidades  $F(t)$

| Tempo $t$ em horas | $F(t) = P[T < t]$ |
|--------------------|-------------------|
| 0                  | 0                 |
| 1                  | 0,393             |
| 2                  | 0,632             |
| 3                  | 0,777             |
| 4                  | 0,865             |
| 5                  | 0,918             |
| 10                 | 0,993             |
| 20                 | 0,9995            |

Observando a Tabela 13.3, nota-se que para  $t = 1$  tem-se  $P[T < t] = 0,393$ . Isso significa que há uma probabilidade de 0,393 de o tempo entre a ocorrência de falhas ser menor que uma hora, ou seja, quase 40% dos tempos entre as falhas são menores que uma hora. Do mesmo modo, 0,918 é a probabilidade de o tempo entre falhas ser menor que 5 horas, ou seja, quase 92% dos tempos entre as falhas são menores que 5 horas, ou, de outra forma, apenas 8% dos tempo entre falhas excedem 5 horas.

É evidente que a distribuição de probabilidades não diz quando as falhas ocorrem. No entanto, ela fornece informações sobre o processo aleatório de ocorrência das falhas.

## Distribuição de Weibull

A distribuição de probabilidades de Weibull [286], como outras distribuições, tem uma grande aplicação em confiabilidade devido à sua adaptabilidade. Dependendo dos valores dos parâmetros, pode-se ajustar a muitos conjuntos de dados sobre falhas. Ao utilizar essa distribuição, assume-se que, no processo de falhas do software, a variável aleatória  $T$  (tempo entre falhas) segue a distribuição de Weibull.

A função Densidade de Probabilidade de Weibull,  $f(t)$ , tem a forma:

$$f(t) = \alpha\beta t^{\alpha-1}e^{-\beta t^\alpha}$$

em que  $\alpha > 0$ ,  $\beta > 0$  e  $t > 0$ .

Os parâmetros  $\alpha$  e  $\beta$  são parâmetros de forma e escala da função Densidade de Probabilidades, respectivamente.

A função Distribuição de Probabilidades tem a forma:

$$F(t) = 1 - e^{-\beta t^\alpha}$$

A função Taxa de Falhas, que pode ser obtida pela Equação (13.3), tem a forma:

$$Z(t) = \alpha \beta t^{\alpha-1}$$

Observe que se  $\alpha = 1$ , a taxa de falhas é constante e se reduz à distribuição exponencial.

A função Confiabilidade, que pode ser obtida pela Equação (13.2), tem a forma:

$$R(t) = e^{-\beta t^\alpha}$$

A função Tempo Médio para Falhas, que pode ser obtida pela Equação (13.9), tem a forma:

$$MTTF = \frac{\Gamma[\frac{1}{\alpha} + 1]}{\beta^{\frac{1}{\alpha}}}$$

em que  $\Gamma[.]$  é a função Gamma, definida como:

$$\Gamma(y) = \int_0^\infty x^y e^{-x} dx$$

para  $y > 0$ .

Casos especiais:

1. Se  $\alpha = 1$ , tem-se a taxa de falhas  $Z(t) = \beta$  (constante). A distribuição reduz-se à distribuição exponencial com parâmetro  $\beta$ .
2. Se  $\alpha = 2$ , tem-se a taxa de falhas  $Z(t) = 2\beta t$  que é linearmente crescente com o tempo.

Assim, tem-se:

$$f(t) = 2\beta t e^{-\beta t^2}$$

e ainda

$$R(t) = e^{-\beta t^2}$$

ou seja, reduz-se à distribuição Rayleigh [286].

## Distribuição Gamma

A distribuição Gamma [286] tem propriedades semelhantes às da distribuição de Weibull. Com variações nos parâmetros, essa distribuição pode-se ajustar a vários conjuntos de dados de falhas. A função Densidade de Probabilidades  $f(t)$  tem a forma:

$$f(t) = \frac{\beta^\alpha}{\Gamma(\alpha)} t^{\alpha-1} e^{-\beta t}$$

em que  $\alpha$  e  $\beta$  são parâmetros de forma e escala, respectivamente, com  $\alpha > 0$ ,  $\beta > 0$  e  $t > 0$ .

A função Confiabilidade, que pode ser obtida da Equação (13.2), tem a forma:

$$R(t) = \int_t^\infty \frac{\beta(\beta x)^{\alpha-1}}{\Gamma(\alpha)} e^{-\beta x} dx$$

A função Tempo Médio para Falha, que pode se obtida pela Equação (13.9), tem a forma:

$$MTTB = \frac{\alpha}{\beta}$$

Casos especiais:

1. Se  $\alpha = 1$ , a função Densidade de Probabilidade reduz-se à distribuição exponencial com parâmetro  $\beta$ . A taxa de falhas é constante.
2. Se  $\alpha = n$ , tal que  $n$  é um número inteiro, a função Densidade de Probabilidade tem a forma:

$$f(t) = \frac{\beta(\beta t)^{n-1}}{(n-1)!} e^{-\beta t}$$

$$n = 1, 2, 3, \dots$$

Essa distribuição de probabilidade é conhecida como distribuição especial de Erlang [286]. Pode ser mostrado que a distribuição de Erlang é a soma de  $n$  distribuições exponenciais com parâmetro  $\beta$ .

A função Confiabilidade tem a forma:

$$R(t) = \sum_{k=0}^{n-1} \frac{(\beta t)^k}{k!} e^{-\beta t}$$

A Tabela 13.4 ilustra as distribuições de probabilidade do tempo de falha mais utilizadas em confiabilidade de software.

Essas informações formam o conjunto de requisitos necessários para o bom entendimento dos modelos de confiabilidade que serão descritos na próxima seção.

Tabela 13.4 – Distribuições de probabilidades do Tempo de Falhas Exponencial, Weibull e Gamma

|            | Exponencial            | Weibull   | Gamma  |
|------------|------------------------|---|--|
| $Z(t)$     | $\alpha$               | $\alpha\beta t^{\alpha-1}$                                    | $\frac{f(t)}{R(t)}$  |
| $f(t)$     | $\alpha e^{-\alpha t}$ | $\alpha\beta t^{\alpha-1}e^{-\beta t^\alpha}$                 | $\frac{\beta^\alpha}{\Gamma(\beta)}t^{\alpha-1}e^{-\beta t}$                     |
| $F(t)$     | $1 - e^{-\alpha t}$    | $1 - e^{-\beta t^\alpha}$                                     | $\int_0^t \frac{\beta^\alpha}{\Gamma(\alpha)} x^{\alpha-1} e^{-\beta t} dx$      |
| $R(t)$     | $e^{-\alpha t}$        | $e^{-\beta t^\alpha}$   | $\int_t^\infty \frac{\beta^\alpha}{\Gamma(\alpha)} x^{\alpha-1} e^{-\beta t} dx$ |
| MTTF       | $\frac{1}{\alpha}$     | $\frac{\Gamma[\frac{1}{\alpha}+1]}{\beta^{\frac{1}{\alpha}}}$ | $\frac{\alpha}{\beta}$   |
| Restrições | $\alpha > 0, t > 0$    | $\alpha > 0, \beta > 0, t > 0$                                | $\alpha > 0, \beta > 0, t > 0$   |

## 13.3 Modelos de confiabilidade

Na seção anterior foram vistos os fundamentos da teoria de confiabilidade de software, requisito necessário para o bom entendimento dos modelos de confiabilidade de software. Nesta seção são discutidos aspectos gerais sobre modelos de confiabilidade de software, iniciando com a fundamentação básica da modelagem de confiabilidade de software. As principais classificações dos modelos de confiabilidade de software existentes na literatura são apresentadas de acordo com a visão de cada autor. As três principais abordagens de modelagem de confiabilidade de software são também apresentadas.

### 13.3.1 Modelagem

Um dos aspectos particulares da engenharia de confiabilidade de software que tem recebido a maior atenção é a modelagem da confiabilidade de software. Para se modelar a confiabilidade de software é necessário considerar os principais fatores que afetam a confiabilidade, que são: a introdução de defeitos, a remoção de defeitos e, finalmente, o ambiente no qual o software é executado.

A introdução de defeitos depende das características do código desenvolvido (código criado inicialmente ou código modificado) e das características do processo de desenvolvimento. As características do processo de desenvolvimento incluem as técnicas de desenvolvimento de software, as ferramentas utilizadas e, por fim, o nível de experiência da equipe de desenvolvimento. Além disso, como Musa e Okumoto [293] observaram, defeitos em software somente podem ser definidos quando são descobertos por meio de uma falha e, assim, não faz sentido contar o número de defeitos em um programa sem que tal programa tenha sido executado por um longo período de tempo. Em geral, muitos pesquisadores de confiabilidade de software têm considerado esse problema modelando a falha no software em vez de defeitos no software.

A remoção de defeitos no software depende do tempo de operação do software, do perfil operacional e da qualidade da atividade de reparos. Alguns pesquisadores contestam essa dependência em relação ao tempo, afirmando que observar o comportamento do software em função do tempo não faz sentido se não houver o controle do que está sendo executado no código. A observação do tempo é uma herança da confiabilidade de hardware, na qual o tempo é um fator importante do comportamento. O ambiente de operação do software depende diretamente do perfil operacional do usuário.

Dado que alguns fatores são de natureza probabilística e operam no tempo, os modelos de confiabilidade de software geralmente são formulados em termos de processos aleatórios. Em termos gerais, os modelos se distinguem pela distribuição de probabilidade do tempo entre falhas ou distribuição do número de falhas observadas e, também, pela natureza da variação do processo aleatório com o tempo.

Um modelo matemático é chamado de modelo de confiabilidade de software se for utilizado para obter uma medida da confiabilidade do software [447]. Todos os modelos matemáticos de confiabilidade de software são de natureza probabilística e, assim, tentam, de algum modo, especificar a probabilidade de ocorrência de falhas do software. O objetivo final é quantificar a confiabilidade do software de uma maneira tão precisa quanto possível.

Um modelo de confiabilidade de software especifica a forma funcional da dependência do processo de falhas sobre os fatores mencionados, isto é, faz uma descrição probabilística precisa da confiabilidade baseada nas suposições *a priori* sobre esses fatores que afetam a confiabilidade e, também, baseada nos resultados obtidos pelos dados experimentais. Várias são as formas matemáticas para se descrever o processo de falhas. Uma forma específica pode ser determinada de uma maneira geral ao se estabelecerem os valores dos parâmetros do modelo por estimação – aplicação de procedimentos de inferência estatística aplicados aos dados de falhas; ou por predição – determinação das propriedades do software e do processo de desenvolvimento (pode ser feita antes da execução do software).

Sempre existe uma incerteza na determinação de uma forma específica. Assim, geralmente se expressam os parâmetros em termos de intervalos de confiança. Uma vez que a forma específica do modelo foi determinada, algumas características do processo de falhas podem ser determinadas. Alguns modelos apresentam uma expressão analítica para essas características, que são:

1. o número médio de falhas observadas em algum ponto no tempo;
2. o número médio de falhas em um intervalo de tempo;
3. a intensidade de falhas em algum ponto no tempo;
4. a distribuição de probabilidade do tempo entre falhas.

A predição do comportamento futuro pressupõe que os valores dos parâmetros do modelo não se alteram nos próximos períodos. De um modo geral, os modelos de confiabilidade de software estão baseados em uma execução estável do software em um ambiente constante.

As medidas de confiabilidade de software podem ser de grande valor para os engenheiros de software, para os gerentes e para os usuários. Com essas medidas, pode-se avaliar o status de desenvolvimento durante as fases de um projeto e tem-se a possibilidade de monitorar o desempenho operacional do software, podendo-se, ainda, controlar as alterações feitas no

software. Finalmente, o entendimento quantitativo da qualidade do software e dos vários fatores que afetam a qualidade enriquece o conhecimento dos processos de desenvolvimento e dos produtos de software.

No entanto, deve ser ressaltado que muitos dos modelos existentes na literatura são testados com dados simulados ou com dados reais que, na maioria das vezes, não foram coletados para tal propósito. O resultado é que algumas suposições básicas desses modelos ou abordagens são violadas. Além disso, é importante salientar que a modelagem de confiabilidade de software é apenas uma das muitas ferramentas da engenharia de software. Como tal, a modelagem não responde a todos os questionamentos relativos à gerência do software. A modelagem de confiabilidade deve ser encarada como uma técnica adicional que ajuda a fazer um julgamento realístico sobre o status do software. Devido às controvérsias existentes sobre qual é o melhor modelo e por causa da incerteza sobre o desempenho das abordagens sobre a modelagem da confiabilidade, deve ser enfatizado que, entre os modelos existentes, recomenda-se aplicar aquele que melhor se ajusta aos dados. A estimativa de confiabilidade pode ser utilizada como mais uma fonte de informação na determinação do status do software.

### 13.3.2 Classificação de modelos de confiabilidade

Nesta subseção são apresentadas algumas formas de classificação dos modelos de confiabilidade de software. Cada autor adota um ponto de vista diferente para fazer sua classificação, mas algumas concordâncias têm estreita relação entre si.

Schick e Wolverton [352] distinguem duas abordagens na modelagem de confiabilidade de software:

- abordagem baseada no domínio dos dados de entrada;
- abordagem baseada no domínio do tempo.

Ramamoorthy e Bastani [334] apresentam um esquema de classificação no qual os modelos se aplicam nas fases do ciclo de vida do sistema. Isto é, modelos aplicáveis na fase de desenvolvimento, na fase de validação, na fase de operação, na fase de manutenção e, finalmente, modelos que se aplicam para medir a correção do software. Nesta classificação, alguns modelos se aplicam a mais de uma fase.

Goel [155] classifica os modelos segundo:

1. A natureza do processo de falhas
  - (a) Modelos baseados no tempo entre falhas;
  - (b) Modelos baseados no número de falhas num intervalo de tempo.
2. A forma de aplicação dos modelos
  - (a) Modelos de implante de defeitos;
  - (b) Modelos baseados no domínio dos dados de entrada.

Mellor [280] classifica os modelos de confiabilidade de acordo com as hipóteses sobre o mecanismo de falhas e com a estrutura matemática: modelos estruturais, que permitem a combinação da confiabilidade dos módulos na obtenção da confiabilidade global do sistema, e modelos que levam em conta somente o comportamento global do sistema. Nessa segunda classificação os modelos se dividem em modelos de tempo entre falhas e modelos que interpretam falhas como a manifestação de defeitos do sistema, supondo que cada defeito dá origem a uma falha com uma certa taxa.

Musa e Okumoto [293] fazem uma classificação dos modelos em função de cinco atributos diferentes:

1. domínio do tempo: tempo cronológico e tempo de execução;
2. categoria finita ou infinita: característica do modelo que se relaciona ao número total de falhas que pode ser observado em um tempo infinito de teste;
3. tipo: característica do modelo que se relaciona à distribuição do número de falhas observadas no tempo  $t$ . Dois tipos de distribuição importantes são Binomial e Poisson;
4. classe (somente para modelos de categoria finita): característica do modelo que se relaciona com a forma funcional da intensidade de falhas expressa em termos do tempo (vide a função  $\lambda(t)$  – função Intensidade de Falhas na Equação (13.8)). As principais classes são: Exponencial, Weibull, Pareto e Gamma;
5. família (somente para modelos de categoria infinita): característica do modelo que se relaciona à forma funcional da intensidade de falhas expressa em termos do número esperado de falhas observadas ( $\lambda(\mu(t))$ ). As principais famílias são: Geométrica, Linear Inversa, Polinomial Inversa e a família Potência.

As Tabelas 13.5 e 13.6 ilustram os principais modelos existentes na literatura, de acordo com a classificação de Musa e Okumoto. Independentemente de qualquer classificação, três abordagens são identificadas na literatura: modelos de implante de defeitos, modelos baseados no domínio dos dados e modelos baseados no domínio do tempo.

## 13.4 Principais modelos de confiabilidade

Nesta seção são apresentados alguns dos mais importantes modelos de confiabilidade de software amplamente citados na literatura, tanto do ponto de vista histórico quanto de suas aplicações. Os modelos são apresentados considerando-se as três principais abordagens anteriormente identificadas. São apresentados também os modelos baseados na cobertura do teste, uma abordagem recente de modelos de confiabilidade de software.

O propósito desta seção é apresentar um levantamento dos principais modelos, considerando as abordagens existentes de modelagem e estimativa da confiabilidade de software. Não se pretende esgotar o assunto, mas, em uma forma sucinta, apresentar as suposições fundamentais do modelo, os dados necessários para sua aplicação e, finalmente, sua forma funcional. Não se apresentam os detalhes sobre a estimativa dos parâmetros de cada modelo nem a dedução matemática de sua forma funcional.

Tabela 13.5 – Modelos de confiabilidade de categoria de falhas finita

| CLASSE  | TIPO                        |                            |                              |
|---|-----------------------------|----------------------------|------------------------------|
|   | Poisson                     | Binomial                   | Outro Tipos                  |
| Exponencial   | Musa<br>(1975)              | Jelinski-Moranda<br>(1972) | Goel-Okumoto<br>(1978)       |
|   | Moranda (1975)              | Shooman (1972)             | Musa (1979a)                 |
|   | Schneidewind<br>(1975)      | -                          | Keiller-Littlewood<br>(1983) |
|   | Goel-Okumoto (1976b)        | -                          | -                            |
| Weibull   | -                           | Schick-Wolerton<br>(1973)  | -                            |
|   | -                           | Wagoner (1973)             | -                            |
|   | -                           | Littlewood (1978)          | -                            |
| Pareto  | -                           |                            |                              |
| Gamma   | Yamada-Ohba-Osaki<br>(1983) | -                          | -                            |
| Tipo: Distribuição do número de falhas observadas                   |                             |                            |                              |
| Classe: Forma funcional da intensidade de falhas em função do tempo |                             |                            |                              |

Tabela 13.6 – Modelos de confiabilidade de categoria de falhas infinita

| FAMÍLIA   | TIPO              |                             |                             |                |
|---|-------------------|-----------------------------|-----------------------------|----------------|
|   | T1                | T2                          | T3                          | Poisson        |
| Geométrica  | Moranda<br>(1975) | -                           | -                           | -              |
| Linear Inversa  | -                 | Littlewood-Verral<br>(1973) | -                           | -              |
| Polinomial Inversa  | -                 | -                           | Littlewood-Verral<br>(1973) | -              |
| Potência  | -                 | -                           | -                           | Crow<br>(1974) |
| Tipo: Distribuição do número de falhas observadas   |                   |                             |                             |                |
| T1, T2, T3: Outras distribuições de probabilidades  |                   |                             |                             |                |
| Família: Forma funcional da intensidade de falhas em função do número esperado de falhas observadas |                   |                             |                             |                |

### 13.4.1 Modelos de implante de defeitos

Nesta subseção são apresentados os modelos que se baseiam na inserção de defeitos no software para estimar sua confiabilidade. Essa abordagem, proposta inicialmente por Mills [284], envolve implantar, em um dado programa, um certo número de defeitos. A suposição é que a distribuição dos defeitos implantados é a mesma dos defeitos inerentes do programa. Assim,

o programa é entregue a uma equipe de teste para validação e verificação. No procedimento de teste, alguns dos defeitos descobertos são defeitos implantados e outros são defeitos reais do programa. Se for feita a contagem desses defeitos, o número total de defeitos inerentes pode ser estimado. Em particular, suponha que 100 defeitos foram implantados em um programa. Após um período de teste, 20 defeitos implantados e 10 defeitos inerentes foram detectados. Os defeitos implantados que foram descobertos representam 20% do total. Assume-se que os 10 defeitos inerentes descobertos também representam 20% do total. Portanto, o número total de defeitos inerentes do programa será estimado em 50.

Uma outra técnica proposta por Rudner [350] emprega o procedimento de teste de dois estágios. O programa é entregue a uma equipe de teste que descobre  $n$  defeitos. Uma segunda equipe testa novamente o programa e descobre  $r$  defeitos, sendo que um número  $k$  desses defeitos também foi detectado pela primeira equipe. Assim, usando uma distribuição Hipergeométrica, o estimador de máxima verossimilhança para o número total,  $N$ , de defeitos no programa tem a forma:

$$\hat{N} = \lceil \frac{nr}{k} \rceil$$

em que  $\lceil \cdot \rceil$  denota a função maior inteiro.

Basin [27] propõe a seguinte técnica: supõe que um programa consiste em  $M$  comandos, dos quais  $n$  são aleatoriamente selecionados para se introduzirem defeitos. Se  $r$  comandos são escolhidos ao acaso e testados, sendo  $k_1$  com defeitos inerentes e  $k_2$  com defeitos implantados, então pode ser mostrado que o estimador de máxima verossimilhança do número total  $N$  de defeitos no programa é dado por:

$$\hat{N} = \lceil \frac{k_1(M - n + 1)}{r - k_2} \rceil$$

em que  $\lceil \cdot \rceil$  denota a função maior inteiro.

Algumas críticas são feitas a essas técnicas. Na prática, não é tão simples implantar defeitos artificiais que sejam equivalentes aos defeitos inerentes em termos de dificuldade de detecção. Geralmente, é muito mais simples detectar os defeitos que foram implantados. Com isso, o número de defeitos inerentes pode ser subestimado. Outro problema é que dificilmente pode-se dar ao luxo de dispor de duas equipes de teste para aplicar a técnica de teste de dois estágios. Além disso, essas técnicas não proporcionam uma medida de confiabilidade para o software dependente do tempo. A inconveniência, ou não, dessa dependência do tempo será discutida posteriormente em modelos baseados em cobertura do teste.

### 13.4.2 Modelos baseados no domínio dos dados

Nesta subseção são apresentados os modelos que se baseiam no particionamento do domínio dos dados de entrada do software para se estimar a sua confiabilidade. Essa abordagem inclui procedimentos que estimam a confiabilidade corrente do programa estritamente baseada no número observado de execuções com sucesso (execuções sem falhas), em relação ao número total de execuções do programa. Nesta categoria, incluem-se, também, os procedimentos que usam dados de teste selecionados de acordo com a distribuição de probabilidades do perfil operacional de uso do programa. O domínio de entrada do programa é dividido em classes,

e as probabilidades de cada classe são fixadas de acordo com o perfil de uso do programa. Como exemplo, vamos supor que o domínio de entrada de um programa seja o conjunto dos números inteiros positivos. Sabe-se antecipadamente que 25%, 35%, 30% e 10% são, respectivamente, as porcentagens dos dados de entradas referentes aos intervalos [0 – 1500], [1501 – 2500], [2501 – 3500] e [3501 e mais ]. Assim, em uma amostra aleatória de 200 casos de teste, 50, 70, 60 e 20 devem ser os números de casos de teste, respectivamente selecionados, para representar cada um dos intervalos. Ou seja, a distribuição de probabilidades de seleção seria 0,25, 0,35, 0,30 e 0,1. A confiabilidade estimada para o programa será o número de execuções com sucesso sobre o valor 200. De uma forma geral, se  $N$  entradas são selecionadas de acordo com o perfil operacional e  $S$  são as execuções com sucesso (sem falhas), então a estimativa da confiabilidade do programa é dada por:

$$\hat{R} = \frac{S}{N}$$

Nesta abordagem, vários pesquisadores propuseram variações na forma de se estimar a confiabilidade.

Hecht [175] propôs os estimadores

$$\hat{R}_1 = \frac{S}{NL}$$

e

$$\hat{R}_2 = \frac{S}{NLW}$$

em que  $L$  é o número de instruções de máquina submetido e  $W$  é o número médio de instruções por bits. Essa modificação normaliza o estimador pelo tamanho do programa e pelo tipo de máquina utilizada.

Nelson [299] propôs um modelo no qual  $n$  entradas são aleatoriamente selecionadas do domínio de entrada  $E = E_i, i = 1, 2, 3, \dots, N$ , sendo cada  $E_i$  o conjunto de dados necessários para se fazer uma execução do programa. A amostra aleatória das  $n$  entradas é feita de acordo com a distribuição de probabilidades  $P_i$ . O conjunto das probabilidades  $P_i; i = 1, 2, 3, \dots, N$  é o perfil operacional do usuário. Se  $n_e$  é o número de entradas cujas execuções resultam em falhas, então um estimador não viciado para a confiabilidade do software será:

$$\hat{R}_1 = 1 - \frac{n_e}{n}$$

Brown e Lipow [47] sugerem uma modificação na qual o espaço de entradas é dividido em regiões homogêneas,  $E_i; i = 1, 2, 3, \dots, k$ . A homogeneidade das regiões é no sentido de geração de defeitos. Supõe-se que  $N_j$  execuções sejam efetuadas e  $F_j$  falhas sejam detectadas para a região  $E_j$ . Assim,  $F_j/N_j$  é uma estimativa da probabilidade de falhas da região  $E_j$ . De acordo com o perfil operacional, se  $PE_j$  é a probabilidade de seleção da região  $E_j$ , então a probabilidade de falha do software é estimada por:

$$\sum_{i=1}^k \frac{F_i}{N_i} PE_i$$

Assim, a confiabilidade do software será estimada por

$$\hat{R} = 1 - \sum_{i=1}^k \frac{F_i}{N_i} PE_i$$

A grande dificuldade dessa técnica é a determinação das regiões homogêneas e as respectivas probabilidades de seleção.

Corcoran, Weingarten e Zehna [93] propõem um modelo que é mais aplicável a hardware, mas que também pode ser aplicado a software. Supõe-se  $M$  tipos de defeitos possam ocorrer em um software. Supõe-se ainda que  $a_i$  seja a probabilidade de remoção do defeito tipo  $i$ , dado que o mesmo tenha sido detectado. Isto é, a probabilidade condicional de corrigir o defeito dado que o defeito tipo  $i$  ocorreu é igual a  $a_i$ , ou:

$$P[\text{corrigir o defeito} \mid \text{defeito do tipo } i \text{ ocorreu}] = a_i, i = 1, 2, 3, \dots, M$$

Se  $N$  execuções são feitas e  $F_i$  defeitos do tipo  $i$  ocorrerem, então um estimador da confiabilidade do programa é dado por:

$$\hat{R} = \frac{S}{N} + \sum_{i=1}^M Y_i \frac{F_i}{N}$$

em que  $S$  é o número total de execuções com sucesso e

$$Y_i = \begin{cases} a_i & \text{se } F_i > 0 \\ 0 & \text{se } F_i = 0 \end{cases}$$

Pode ser mostrado que essa estimativa é assintoticamente não viciada e sua variância tende a zero quando  $N$  é grande. A dificuldade em se aplicar esse modelo é conhecer os  $M$  tipos de defeitos do software e as probabilidades  $a_i$ .

### 13.4.3 Modelos baseados no domínio do tempo

Nesta subseção são apresentados os modelos que se baseiam na ocorrência de falhas do software ao longo do tempo para se estimar a confiabilidade.

A modelagem da confiabilidade baseada no domínio do tempo é a abordagem que tem recebido maior ênfase na pesquisa. Essa abordagem utiliza o tempo de ocorrência entre falhas ou o número de falhas ocorridos em um intervalo de tempo, para modelar o processo de falhas no software. Em geral, os modelos podem ser utilizados para predizer o tempo até a ocorrência da próxima falha ou o número esperado de falhas no próximo intervalo de tempo. Originalmente, esses modelos foram baseados nos conceitos sobre confiabilidade

de hardware e, assim, muitos termos usados em confiabilidade de hardware são, também, usados em confiabilidade de software.

Nesta abordagem muitos modelos já foram propostos e várias extensões foram sugeridas. No entanto, existe uma grande controvérsia sobre qual é o melhor modelo para um conjunto de dados de falhas em um software. Alguns estudos foram realizados usando dados simulados sobre falhas, outros foram realizados com dados reais. A conclusão é que se deve aplicar o maior número possível de modelos em um conjunto de dados e escolher o melhor com base em critérios estatísticos.

Os conceitos para a modelagem de confiabilidade de hardware foram adaptados à modelagem de confiabilidade de software, o que não implica que o comportamento do software seja semelhante ao comportamento do hardware. Pelo contrário, o software não se desgasta com o uso. No processo de reprodução do software não existe a geração aleatória de novos defeitos nas cópias. As duplicatas são idênticas. Além disso, o software não se altera durante o uso nem se desgasta pela ação do tempo. O decorrer do tempo não causa defeitos no software e, assim, a geração de defeitos ou mesmo a ocorrência de falhas no software independem do tempo.

Por essas razões, um grande número de pesquisadores tem questionado fortemente a abordagem que utiliza a variável tempo no processo de modelagem da confiabilidade de software [43, 217].

Nesta abordagem, os modelos encontram-se classificados em duas classes básicas, dependendo do tipo de dados de falhas que se utiliza:

- falhas por intervalo de tempo;
- tempo entre falhas.

Essas classes, contudo, não são disjuntas. Existem modelos que aceitam qualquer um dos dois tipos de dados. Além disso, os dados podem ser transformados de um tipo para outro, adaptando-se a qualquer uma das classes de modelos.

De acordo com a classificação de Musa [293] existem dois tipos importantes de modelos cuja categoria de falhas é finita: o tipo Poisson e o tipo Binomial.

Para os modelos do tipo Poisson, considera-se que o processo de falhas no software segue um processo de Poisson no tempo. Assim, a variável aleatória  $M(t)$  representa o número de falhas observadas no tempo  $t$  com um valor médio dado por  $\mu(t)$ . Ou seja,  $\mu(t) = E[M(t)]$ .

Considerando-se  $t_0 = 0, t_1, t_2, \dots, t_{i-1}, t_i, \dots, t_n = t$  uma partição no intervalo  $[0, t]$ , tem-se um processo de Poisson se as variáveis  $f_i$ ,  $i = 1, 2, \dots, n$  (representando o número de falhas detectadas no intervalo  $[t_{i-1}, t_i]$ ) forem independentes com uma distribuição de probabilidade Poisson cujo valor médio é dado por  $E[f_i] = \mu(t_i) - \mu(t_{i-1})$ .

Assim, para cada variável aleatória  $f_i$ , a função Densidade de Probabilidade é dada por:

$$P[f_i = x] = \frac{[\mu(t_i) - \mu(t_{i-1})]^x}{x!} e^{-[\mu(t_i) - \mu(t_{i-1})]}$$

para  $x = 0, 1, 2, \dots$

Observa-se que se  $\mu(t)$  é uma função linear do tempo, então tem-se um processo de Poisson homogêneo (HPP), isto é,  $\mu(t) = \alpha t$ ,  $\alpha > 0$ . Se  $\mu(t)$  não é uma função linear no tempo, tem-se um processo de Poisson não homogêneo (NHPP).

Musa [292] demonstra as seguintes relações para modelos tipo Poisson:

a)  $Z(t_i|t_{i-1}) = \lambda(t_i)$ , ou seja, a taxa de falhas do processo é igual à função de intensidade de falhas no intervalo  $t_i$ ,

$$\text{b)} \quad \mu(t_i) = \alpha F_a(t_i)$$

$$\text{c)} \quad \lambda(t_i) = \mu'(t_i) = \alpha F_a(t_i)$$

$$\text{d)} \quad R(t_i|t_{i-1}) = e^{-[\mu(t_i) - \mu(t_{i-1})]}$$

em que  $\alpha$  é uma constante que representa o número de defeitos detectados no software,  $f_a(t)$  e  $F_a(t)$  são, respectivamente, a função Densidade de Probabilidade e a função Distribuição de Probabilidades acumuladas do tempo de falha de um defeito “a”, e  $R(t)$  é a função Confiabilidade.

Para os modelos do tipo Binomial, as seguintes suposições são consideradas:

1. existe um número fixo  $N$  de defeitos no software no início do teste;
2. quando um defeito é detectado, é imediatamente removido;
3. usando a notação de Musa [292], se  $T_a$  é a variável aleatória que denota o tempo de falha de um defeito “a”, então as variáveis  $T_a$  são independentes e identicamente distribuídas para todos os defeitos.

Nota-se que, nesse caso, a função de Distribuição de Probabilidades  $F_a(t)$ , a função Densidade de Probabilidades,  $f_a(t)$  e a Taxa de Falhas  $Z_a(t)$  são as mesmas para todos os defeitos.

Musa [292] mostra que, para esse tipo de modelo, a Taxa de Falhas  $Z(t)$ , a função Intensidade de Falhas  $\lambda(t)$ , a função Valor Médio do processo  $\mu(t)$  e a função Confiabilidade têm, respectivamente, as formas:

$$\text{a)} \quad Z(t_i|t_{i-1}) = (N - i + 1)Z_a(t_i)$$

$$\text{b)} \quad \lambda(t) = Nf_a(t)$$

$$\text{c)} \quad \mu(t) = NF_a(t)$$

$$\text{d)} \quad R(t_i|t_{i-1}) = e^{[-(N-i+1) \int_{t_{i-1}}^{t_i} Z_a(t)dt]}$$

Observa-se a similaridade entre as funções  $\lambda(t) = \alpha f_a(t)$  e  $\lambda(t) = Nf_a(t)$  para os modelos dos tipos Poisson e Binomial, respectivamente. O mesmo acontece para as funções  $\mu(t) = \alpha F_a(t)$  e  $\mu(t) = NF_a(t)$ .

O parâmetro  $N$  nos modelos tipo Binomial representa o número de defeitos no software no início do teste, enquanto o parâmetro  $\alpha$  no tipo Poisson representa o eventual número de defeitos que podem ser descobertos em um tempo infinito de teste.

A Tabela 13.7 ilustra as relações derivadas para os modelos tipo Binomial e Poisson.

Tabela 13.7 – Relações derivadas para os mdelos do tipo Binomial e Poisson

|                                       | TIPO  |                                   |
|---------------------------------------|---|-----------------------------------|
| Relações Derivadas                    | Binomial                                      | Poisson                           |
| Falhas Acumuladas<br>$P[M(t) = k]$    | $\binom{N}{k} F_a(t)^k [1 - F_a(t)]^{N-k}$    | $\frac{\mu(t)^k}{k!} e^{-\mu(t)}$ |
| Valor Médio<br>$\mu(t)$               | $N F_a(t)$                                    | $\alpha F_a(t)$                   |
| Intensidade de Falhas<br>$\lambda(t)$ | $N f_a(t)$                                    | $\alpha f_a(t)$                   |
| Taxa de falhas<br>$Z(t_i t_{i-1})$    | $(N - i + 1) Z_a(t_i)$                        | $\alpha f_a(t_i)$                 |
| Confiabilidade<br>$R(t_i t_{i-1})$    | $e^{-(N-i+1) \int_{t_{i-1}}^{t_i} Z_a(t) dt}$ | $e^{-[\mu(t_i) - \mu(t_{i-1})]}$  |

$f_a(t)$ : função Densidade de Probabilidade do tempo de falha do defeito “a”  
 $F_a(t)$ : função Distribuição de Probabilidade do tempo de falha do defeito “a”

Cada um dos modelos que serão descritos nesta abordagem foram criados com base em suposições específicas, mas existem algumas suposições padrão comuns à maioria dos modelos, tais como:

1. o software é operado de uma maneira semelhante em que as previsões da confiabilidade são feitas;
2. todos os defeitos de uma classe de dificuldade têm chance idêntica de serem encontrados;
3. as falhas são independentes.

A suposição 1 é para garantir que as estimativas do modelo, usando-se os dados coletados no ambiente de teste, sejam válidas quando utilizadas no ambiente de operação do software. A suposição 2 garante que todas as falhas têm as mesmas propriedades em suas distribuições. Finalmente, a suposição 3 permite que os estimadores dos parâmetros sejam calculados utilizando-se o método da máxima verossimilhança.

## Modelo de Weibull

Um dos modelos mais amplamente utilizados para se modelar a confiabilidade de hardware é a distribuição de Weibull. Desde que vários conceitos foram adaptados para software, esse foi um dos primeiros modelos usados para se modelar a confiabilidade de software. Devido à natureza da distribuição de Weibull, este modelo pode ser usado em situações nas quais a taxa de falhas seja crescente, decrescente ou mesmo constante, dependendo do valor do parâmetro da distribuição. Segundo a classificação adotada por Musa [293], este modelo pertence à categoria de falhas finita, de classe Weibull e de tipo Binomial.

Suposições do modelo:

1. o software é operado de maneira semelhante àquela em que as previsões da confiabilidade são feitas;
2. todos os defeitos de uma classe de dificuldade têm chance idêntica de serem encontrados;
3. as falhas, quando os defeitos são detectados, são independentes;
4. existe um número N de defeitos no software no início do teste;
5. o tempo de falha de um defeito “a”, denotado como  $T_a$ , tem uma distribuição de probabilidades Weibull com parâmetros  $\alpha$  e  $\beta$ ;
6. os números de defeitos  $f_1, f_2, f_3, \dots, f_n$  detectados em cada um dos intervalos de tempo  $(t_0 = 0, t_1), (t_1, t_2), \dots, (t_i, t_{i-1}), \dots, (t_{n-1}, t_n)$  são independentes.

Conforme a suposição 5, a taxa de falhas tem a forma  $Z_a(t) = \alpha\beta t^{\alpha-1}$ , em que  $t \geq 0$  e  $\alpha$  e  $\beta$  são constantes. Observa-se que se  $\alpha > 1$ , então a taxa de falhas cresce com o tempo t; se  $\alpha < 1$ , então a taxa de falhas decresce com o tempo t e se  $\alpha = 1$ , a taxa de falhas é constante com o tempo t.

A função Densidade de Probabilidade para o tempo de falha  $T_a$  do defeito “a” é a distribuição de Weibull, ou seja:

$$f_a(t) = \alpha\beta t^{\alpha-1} e^{(-\beta t^\alpha)}$$

em que  $t \geq 0$ ,  $\alpha > 0$  e  $\beta > 0$

A função de distribuição de probabilidade é dada por:

$$F_a(t) = \int_0^t f_a(x)dx = 1 - e^{(-\beta t^\alpha)}$$

em que  $t \geq 0$ ,  $\alpha > 0$  e  $\beta > 0$

Os dados necessários para a aplicação desse modelo são:

- a) o número de defeitos em cada intervalo de tempo, ou seja, os  $f_i$ ; e
- b) o tamanho de cada intervalo de tempo em que o software é testado, isto é, os  $t_i$ .

Desde que o modelo de Weibull é do tipo Binomial, então a função Intensidade de Falhas  $\lambda(t)$  e a função Valor Médio  $\mu(t)$  do processo de falhas são dadas por:

$$\lambda(t) = Nf_a(t) = N\alpha\beta t^{\alpha-1} e^{(-\beta t^\alpha)}$$

$$\mu(t) = NF_a(t) = N[1 - e^{(-\beta t^\alpha)}]$$

Observa-se que  $\lim_{t \rightarrow \infty} \mu(t) = N$  é o número de falhas que podem ser detectadas no software.

A função Confiabilidade é obtida da função Distribuição de Probabilidade como:

$$R(t) = 1 - F(t) = e^{(-\beta t^\alpha)}$$

e o tempo médio de falhas MTTF é dado por:

$$MTTF = \int_0^{\infty} R(t)dt = \Gamma\left(\frac{1}{\alpha} + 1\right) \frac{1}{\beta^{\frac{1}{\alpha}}}$$

em que  $\Gamma(\cdot)$  é a função Gamma.

Coutinho [106] mostra que os parâmetros  $\alpha$  e  $\beta$  podem ser estimados pelo método dos momentos, método dos mínimos quadrados, método da máxima verossimilhança ou mesmo pelo método gráfico.

## Modelo de Jelinski-Moranda

Um dos primeiros modelos propostos e ainda muito utilizado é o de Jelinski-Moranda [289], desenvolvido na McDonnell Douglas Astronautics Company. Esse modelo foi criado para o projeto Apollo, e vários outros modelos são pequenas variações desse modelo inicial.

A idéia básica é que o tempo entre falhas segue uma distribuição exponencial cujo parâmetro é proporcional ao número de falhas restantes no software. Assim, o tempo médio entre a  $i - 1$  e a  $i$ -ésima falha é dado por  $\frac{1}{\theta(N-i+1)}$ , em que  $N$  é o número de falhas no software no início do teste e o parâmetro  $\theta$  é a constante de proporcionalidade. Isso indica que o impacto da remoção de cada defeito é sempre o mesmo.

De acordo com a classificação de Musa [292], esse modelo é de categoria de falhas finita, de classe exponencial e do tipo Binomial. O modelo de Jelinski-Moranda está baseado nas seguintes suposições:

1. o software é operado de maneira semelhante àquela em que as previsões da confiabilidade são feitas;
2. todos os defeitos de uma classe de dificuldade têm chance idêntica de serem encontrados;
3. as falhas, quando os defeitos são detectados, são independentes;
4. a taxa de detecção de defeitos é proporcional ao número de defeitos correntes no software;
5. a taxa de detecção de defeitos permanece constante no intervalo entre a ocorrência de falhas;
6. um defeito é instantaneamente removido sem a introdução de novos defeitos no software.

Os dados necessários para se utilizar esse modelo são:

- a) o intervalo de tempo entre falhas  $x_1, x_2, \dots, x_n$ ; ou
- b) o tempo em que o software falhou,  $t_1, t_2, \dots, t_n$ , para  $x_i = t_i - t_{i-1}$ ,  $i = 1, 2, \dots, n$ , em que  $t_0 = 0$ .

De acordo com as suposições, se o tempo de ocorrência de falhas é representado pela variável  $X_i = T_i - T_{i-1}$ ,  $i = 1, 2, \dots, n$ , então as variáveis aleatórias  $X_i$  são independentes e exponencialmente distribuídas, com média  $\frac{1}{\theta(N-i+1)}$ .

A função Densidade de Probabilidade da variável aleatória X é dada por:

$$f(x_i|t_{i-1}) = \theta(N-i+1)e^{[-\theta(N-i+1)]}$$

Desde que o modelo é do tipo Binomial, tem-se que a função Intensidade de Falhas  $\lambda(t)$  e a função Valor Médio  $\mu(t)$  são dadas por:

$$\lambda(t) = N f_a(t) = N \theta e^{(-\theta t)}$$

$$\mu(t) = NF_a(t) = N[1 - e^{(-\theta t)}]$$

Observando-se que  $\lim_{t \rightarrow \infty} \mu(t) = N$ , confirma-se que esse modelo é de categoria de falhas finita.

Os estimadores de máxima verossimilhança dos parâmetros  $\theta$  e  $N$  são dados por:

$$\begin{aligned} \hat{\theta} &= \frac{n}{\hat{N} \sum_{i=1}^n X_i - \sum_{i=1}^n (i-1)X_i} \\ \sum_{i=1}^n \left[ \frac{1}{\hat{N} - i + 1} \right] &= \frac{n}{\hat{N} - \left[ \frac{1}{\sum_{i=1}^n X_i} \right] \sum_{i=1}^n (i-1)X_i} \end{aligned}$$

A segunda equação deve ser resolvida por técnicas numéricas para se encontrar a estimativa de N. Ao se substituir o valor na primeira equação, tem-se a estimativa de  $\theta$ .

## Modelo geométrico

O modelo geométrico foi proposto por Moranda [288] e é uma variação do modelo De-Eutrophication de Jelinski-Moranda. Este é um modelo interessante porque, de modo diferente dos modelos anteriormente discutidos, não assume um número fixo de defeitos no software nem assume que as falhas tenham a mesma probabilidade de ocorrência. O modelo assume que, com o progresso da depuração, os defeitos tornam-se mais difíceis de ser detectados. O tempo entre falhas é considerado como tendo uma distribuição exponencial cuja média decresce em uma forma geométrica.

Inicialmente, a taxa de falhas assume o valor de uma constante  $D$  e decresce geometricamente quando as falhas ocorrem. O modelo reflete o grande impacto das primeiras falhas e a difícil redução da taxa nas últimas falhas. O decréscimo na taxa de falhas torna-se menor à medida que os defeitos são detectados.

O modelo geométrico está baseado nas seguintes suposições:

1. o software é operado de uma maneira semelhante em que as previsões da confiabilidade são feitas;
2. todos os defeitos de uma classe de dificuldade têm chance idêntica de serem encontrados;
3. as falhas, quando os defeitos são detectados, são independentes;
4. a taxa de detecção de defeitos forma uma progressão geométrica e é constante entre a detecção dos defeitos, isto é,  $Z(t) = D\theta^{i-1}$ , em que  $0 < \theta < 1$  e  $t_{i-1} < t < t_i$ , com  $t_{i-1}$  sendo o tempo inicial do intervalo de ocorrência da  $(i-1)$ -ésima falha;
5. existe um número infinito de defeitos no software, ou seja,  $\lim_{n \rightarrow \infty} \mu(t) = \infty$ , em que  $\mu(t)$  é a função Valor Médio do processo; e
6. o tempo entre a detecção de defeitos segue uma distribuição exponencial.

Observa-se que, de acordo com a classificação de Musa [293], esse modelo é de categoria de falhas infinita e da família geométrica.

Os dados necessários à aplicação deste modelo são:

- a) os tempos de ocorrência das falhas,  $t_i$ ; ou
- b) os tempos entre as ocorrências de falhas,  $x_i (x_i = t_i - t_{i-1})$ .

Conforme a suposição 6, a função Densidade de Probabilidade do tempo entre a ocorrência de falhas é dada por:

$$f(x_i) = D\theta^{i-1}e^{-D\theta^{i-1}x_i}$$

A função Intensidade de Falhas  $\lambda(t)$  e a função Valor Médio  $\mu(t)$  são dadas por:

$$\lambda(t) = \frac{De^\beta}{[D\beta e^\beta]t + 1}$$

$$\mu(t) = \frac{1}{\beta} \ln[(D\beta e^\beta)t + 1]$$

em que  $\beta = \ln(\theta)$  para  $0 < \theta < 1$ .

Os estimadores de máxima verossimilhança dos parâmetros  $D$  e  $\theta$  são dados por:

$$\begin{aligned}\hat{D} &= \frac{n\hat{\theta}}{\sum_{i=1}^n (\hat{\theta})^i x_i} \\ \frac{\sum_{i=1}^n i(\hat{\theta})^i x_i}{\sum_{i=1}^n (\hat{\theta})^i x_i} &= \frac{n+1}{2}\end{aligned}$$

O tempo médio entre falhas é dado por:

$$MTBF = \frac{1}{\hat{D}(\hat{\theta}^n)} = \hat{E}[X_{n+1}]$$

O modelo não estima o número de defeitos no software.

#### 13.4.4 Modelos baseados em cobertura de teste

A estimativa de confiabilidade de software tem a sua importância por várias razões bem conhecidas na literatura. Nesse sentido, vários são os modelos criados para a estimativa de confiabilidade de software. No entanto, todos os modelos propostos são formulados e fundamentados em uma abordagem de teste funcional, ou teste caixa preta, nos quais a maior preocupação é a obtenção de uma forma funcional que explique o comportamento das falhas no software. Nenhum dos modelos até agora apresentados utiliza a informação sobre a cobertura do código.

A utilização da análise de cobertura dos elementos requeridos de um critério de teste estrutural tem a vantagem de que, no decorrer do teste, obtém-se a informação sobre o percentual do código exercitado durante o teste. Uma outra vantagem é a possibilidade de avaliar o problema da superestimação da confiabilidade do software criado pelo efeito de saturação do critério de teste.

Nesse contexto, os modelos de confiabilidade apresentados a seguir utilizam a informação da cobertura do critério de teste como um parâmetro próprio do modelo, isto é, a informação da cobertura é utilizada diretamente na forma funcional do modelo de confiabilidade.

Os modelos de confiabilidade anteriormente apresentados baseiam-se no tempo de teste do software. Nos modelos de confiabilidade baseados em cobertura supõe-se que a execução de um dado de teste corresponde a uma unidade de tempo de execução do software. A informação da cobertura obtida é diretamente utilizada no processo de modelagem da confiabilidade.

Malaya [259] faz essa mesma suposição quando cria um modelo que relaciona a cobertura do código com número de dados de teste para avaliar a confiabilidade do software. Trata-se de um modelo que explica a cobertura em função dos dados de teste. Da mesma forma, Chen [68] também faz essa suposição quando utiliza a informação da cobertura para definir um fator a ser utilizado nos tradicionais modelos de confiabilidade com a finalidade de corrigir a confiabilidade estimada por esses modelos. Chen et al. [70] também relatam o desenvolvimento de um trabalho que envolve a relação entre cobertura e confiabilidade.

Trabalhos recentes também abordam o tratamento da confiabilidade de software por meio da informação sobre a cobertura atingida durante a execução do teste. Malaiya et al. [260] propõem um modelo que relaciona uma medida da cobertura do teste diretamente com a cobertura de defeitos. Chen, Lyu e Wong [71] propõem uma abordagem para a predição de falhas de software durante sua operação por meio de medições de tempo entre casos de teste e de cobertura de código. Pham e Zhang [328] propõem um modelo de confiabilidade de software baseado em um processo não homogêneo de Poisson que incorpora a informação de cobertura de teste para estimar e prever quantitativamente a confiabilidade de produtos de software.

A seguir é descrito um modelo de confiabilidade de software que faz uso da informação da cobertura do código atingida durante o teste para estimar e prever diretamente a confiabilidade do software.

### **Modelo tipo Binomial baseado em cobertura – MBBC**

O modelo de crescimento de confiabilidade tipo Binomial baseado em cobertura foi proposto em 1997 por Crespo [98, 100]. Os modelos tipo Binomial são caracterizados pela forma funcional da taxa de falhas por defeito “*a*”,  $Z_a(n)$ , em que  $n$  é o número de dados de teste aplicados para se revelar o defeito “*a*”.

Neste modelo, a taxa de falhas  $Z_a(n)$  é proporcional às seguintes medidas:

- a) Número de dados de teste aplicados no software até a ocorrência da falha provocada pelo defeito “*a*”.
- b) Complemento da cobertura alcançada no teste com a aplicação dos dados de teste até a ocorrência da falha provocada pelo defeito “*a*”.
- c) Peso do critério de teste utilizado como estratégia para a geração dos dados de teste.

O modelo tipo Binomial está fundamentado nas seguintes suposições:

1. o software é testado nas mesmas condições quando utilizado pelo usuário;
2. todos os defeitos de uma classe de dificuldade têm chance idêntica de serem encontrados;
3. os defeitos  $1, 2, \dots, k$  detectados, respectivamente, em cada um dos intervalos  $(0; n_1)$ ,  $(n_1; n_2), (n_2; n_3), \dots, (n_{k-1}; n_k)$  são independentes;
4. existe um número  $N$  de defeitos no software no início do teste;
5. a cobertura dos elementos requeridos pelo critério de seleção utilizado na avaliação dos dados é calculada à medida que os dados de teste são aplicados, a cada ocorrência de falha;
6. a taxa de falhas condicional tem a seguinte forma funcional:

$$Z(k_i|n_i) = [N - i]\alpha_i(n_i + k_i)^{\alpha_i - 1}$$

em que:

- $\alpha_i = \alpha_0 + \alpha_1 c_i$  é a cobertura normalizada,  $c_i$  é o complemento da cobertura medida atingida com a aplicação dos  $n_i$  dados de teste,  $0 \leq c_i \leq 1$ ;
- $N$  é o número de defeitos no software no início do teste;
- $i$  é a ordem de ocorrência das falhas, isto é,  $i = 1, 2, 3, \dots, N$ .

A Figura 13.8 ilustra o processo de teste do software para a avaliação da confiabilidade.

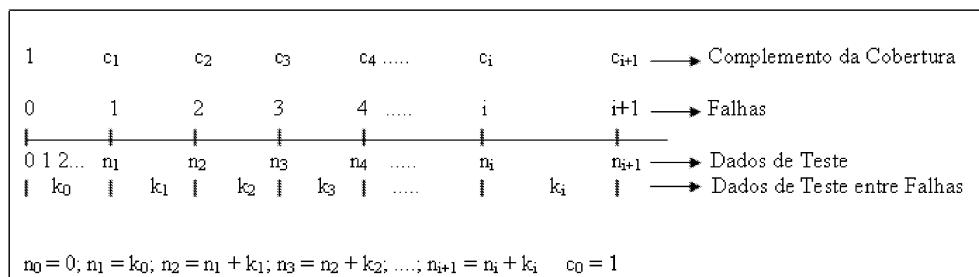


Figura 13.8 – Processo de teste do software.

O peso do critério de seleção utilizado no teste condiciona o comportamento da taxa de falhas do software. Sendo a função Confiabilidade relacionada à taxa de falhas, significa que a adoção de diferentes critérios conduziria a diferentes previsões do comportamento futuro do software. Dessa forma, objetivando padronizar as estimativas geradas pelo modelo, independentemente de qualquer critério, é justificado o uso da cobertura normalizada no lugar da cobertura medida no teste.

O objetivo de se utilizar a cobertura normalizada no lugar da cobertura medida é para garantir a padronização dos resultados obtidos pelo modelo quando se aplica qualquer um dos critérios de teste. Em outras palavras, qualquer que seja o critério de teste utilizado, a cobertura medida será sempre normalizada pelos parâmetros  $\alpha_0$  e  $\alpha_1$ , garantindo a obtenção dos mesmos resultados nas previsões do modelo ao se utilizar qualquer um dos critérios de teste para se medir a cobertura. As estimativas dos parâmetros  $\alpha_0$  e  $\alpha_1$  serão distintas para diferentes critérios de teste adotados.

Após a remoção do  $i$ -ésimo defeito, a confiabilidade do software pode ser estimada pela seguinte equação:

$$R(k_i | n_i) = e^{-[N-i][(n_i+k_i)^{\alpha_i} - (n_i)^{\alpha_i}]}$$

em que:

- $n_i$  é o número de dados de testes aplicados até a ocorrência da  $i$ -ésima falha;

- $k_i$  é o número de dados de teste aplicados após a ocorrência da  $i$ -ésima falha;
- $\alpha_i$  é a cobertura normalizada atingida até a aplicação de  $n_i$  dados de teste.

Os parâmetros  $N$ ,  $\alpha_0$  e  $\alpha_1$  da equação  $\alpha_i = \alpha_0 + \alpha_1 c_i$  são estimados pelo método da máxima verossimilhança que dá origem a um sistema de equações não lineares. Utilizando-se métodos numéricos, os valores que satisfazem simultaneamente as equações do sistema maximizam a função de verossimilhança e são, portanto, os valores estimados para os parâmetros.

O sistema de equações que descreve a estimativa dos parâmetros pode ser visto em Crespo [98].

### 13.4.5 Limitações dos modelos de confiabilidade

Apesar do grande esforço de pesquisa na área de modelagem, os modelos fornecem apenas uma estimativa grosseira da confiabilidade do software. As principais aplicações dos modelos têm se limitado ao suporte à gerência dos projetos e ao seu uso como critério de parada para a atividade de teste.

No procedimento sobre a modelagem de confiabilidade de software o programa é executado várias vezes usando os dados de teste selecionados aleatoriamente, de acordo com o perfil operacional. Quando uma falha ocorre, pára-se o teste e, em seguida, inicia-se o procedimento de detecção e remoção do defeito que causou a falha. Novamente, inicia-se o teste até que uma nova falha ocorra. Em alguns modelos utilizam-se os resultados do teste para se estimar o número de defeitos restante e obter a taxa de falhas em função desse número. Em outros modelos utiliza-se a seqüência do tempo entre falhas para se medir a confiabilidade corrente e, também, estimar o crescimento da confiabilidade com a execução de novos dados de teste. A informação sobre a cobertura de elementos requeridos, considerando-se um teste caixa branca, geralmente não é utilizada, mesmo sabendo que a ocorrência de falhas, na maioria das vezes, está relacionada ao exercício de elementos requeridos do teste.

Uma outra restrição nos modelos de confiabilidade de software são os critérios de teste. Todo critério de teste tem um limite na sua capacidade de detectar a validade dos dados que têm alguma chance de revelar defeitos em um programa [68]. Quando um critério de teste atinge seu limite, não se consegue mais detectar a validade de um novo dado de teste e, assim, o tempo entre falhas pode aumentar consideravelmente. Logo, a estimativa da confiabilidade, produzida pelos modelos baseados no domínio do tempo, cresce sem que haja a remoção de algum novo defeito. Os testadores que não estiverem conscientes do nível de saturação do critério de teste podem obter uma superestimação da confiabilidade.

A estimativa do perfil operacional é um dos fatores importantes no uso de modelos de crescimento de confiabilidade. Um perfil operacional é a função Densidade de Probabilidade, que melhor representa como os dados de entrada são selecionados durante o tempo de teste do software [293]. Sem um perfil operacional preciso, as estimativas dos modelos certamente serão incorretas. No entanto, o perfil operacional pode ser difícil de ser estimado, principalmente em softwares utilizados em controle de processos. Em outros casos, um único perfil operacional de um software pode não ser suficiente para os diferentes usuários deste software. Além disso, um perfil operacional pode ser alterado durante o processo de manutenção do software. Todas essas causas podem conduzir a erros na estimativa do per-

fil operacional e esses erros certamente afetam a sensibilidade dos modelos de crescimento de confiabilidade [99, 326].

Os modelos baseados no domínio do tempo não consideram o critério de teste utilizado. Experimentos realizados [72] evidenciam que diferentes técnicas de teste resultam em diferentes estimativas da confiabilidade, o que, evidentemente, afeta o desempenho desses modelos. Dessa maneira, esses modelos apenas tentam simular a forma de uma função matemática, ou seja, a forma de uma função que representa o crescimento da confiabilidade. As formas funcionais propostas não consideram aspectos que afetam o comportamento da confiabilidade. Além dessas limitações, existem outros problemas referentes às suposições que são feitas em muitos modelos de crescimento da confiabilidade e que não podem ser consideradas em aplicações práticas [393, 394].

Várias suposições específicas limitam a aplicabilidade e a eficácia dos modelos, tais como:

- O tempo é utilizado como base para o cálculo da taxa de falhas.

Nesta suposição, está implícito que o esforço do teste é proporcional ao tempo. Todavia, o tempo de calendário e o tempo de relógio de parede somente captam o instante da falha. Em geral, a distribuição da carga de trabalho do sistema (*workload*) é altamente desigual. Assim, dados baseados nesses tempos tornam-se impróprios [393, 394]. Isso conduz à preferência por dados baseados em tempo lógico ou tempo de execução, para medir ou modelar a confiabilidade. Uma outra restrição com referência ao tempo, como variável de controle do teste, é que não se pode garantir que o esforço do teste está sendo adequado se as entradas selecionadas do domínio não executam pelo menos as principais funções do software em teste. Imaginando-se que fosse possível a realização de um teste no qual o tempo  $t$  tende ao infinito, de nada adiantaria o esforço desse teste se as entradas exercitassem sempre as mesmas funções do software ou, equivalente, sempre os mesmos elementos, no caso de um teste estrutural. Nesse ponto, ressalta-se a importância de se observar a cobertura de elementos requeridos, quando se utiliza um teste caixa branca, para garantir que pelo menos uma certa porcentagem dos elementos requeridos seja exercitada. Nesse contexto, o tempo não é importante. Muitos modelos de confiabilidade consideram o tempo como variável de controle do teste. Essa utilização do tempo deve-se à herança da teoria sobre confiabilidade de hardware, na qual a variável tempo para o teste é de grande significância.

- Independência de tempo entre falhas.

Essa suposição implica a utilização de uma seleção aleatória dos dados de teste. No entanto, muitas vezes se utiliza uma seleção dirigida ou agrupada com a finalidade de se conseguir uma boa cobertura dos requisitos funcionais [393]. No caso de seleção não aleatória não se pode garantir a independência de tempo entre falhas.

- A confiabilidade como uma função do número de defeitos restantes no software.

Essa suposição implica uma distribuição homogênea dos defeitos no software e, também, uma distribuição uniforme na taxa de detecção de defeitos. Ocorre que, geralmente, a distribuição dos defeitos é desigual, confirmada por estudos recentes [394]. Essas evidências explicam porque, na maioria das vezes, os modelos de crescimento de confiabilidade não funcionam corretamente, ou então, se são aplicados a um software, não se aplicam a outros. Acredita-se que ainda seja preciso um grande esforço de pesquisa para se chegar à solução de todos esses problemas.

Nesse sentido, a utilização da cobertura do critério obtida no teste é uma informação adicional que pode ser utilizada na estimativa da confiabilidade do software. A cobertura do teste estrutural e a confiabilidade de software estão estreitamente relacionadas [148, 97]. O uso da cobertura de elementos requeridos, no estudo da confiabilidade, está apoiado na existência de uma forte correlação com a confiabilidade [402]. Pesquisas, tanto no campo teórico como no experimental, comprovam a existência de alguma relação entre a confiabilidade e a cobertura de elementos requeridos de um teste estrutural. A abordagem que utiliza a cobertura do código como informação relacionada à confiabilidade é uma alternativa consistente à tradicional abordagem caixa preta de teste, já que esta não considera a estrutura do código para a estimativa de confiabilidade.

## 13.5 Cálculo da confiabilidade de software

O tema confiabilidade de software é alvo de grande atenção pela comunidade de desenvolvedores de software. Com o crescimento da utilização do software em todas as áreas da atividade humana, e mantida essa tendência, surge uma questão bastante relevante. Qual é o esforço a ser empregado na fase de teste de um software para que seja liberado para seu usuário? Certamente a resposta para essa questão requer uma análise mais profunda, uma vez que o grau de confiança desejado pelo usuário para um software depende do tipo de software e de sua área de aplicação.

A medida da confiabilidade de software proporciona uma resposta quantitativa para essa questão. Nesse sentido, mais de 70 modelos de confiabilidade baseados em diversas abordagens podem ser encontrados na literatura sobre confiabilidade de software. Esses modelos, na grande maioria, requerem um sofisticado procedimento numérico para o cálculo das estimativas de seus parâmetros.

### 13.5.1 Procedimento geral

Como visto anteriormente, existem vários modelos de confiabilidade de software que podem ser utilizados na tomada de decisões. Além da simples satisfação dos requisitos básicos para a utilização de determinado modelo, não existe um critério que possa ser utilizado para a seleção de um modelo de confiabilidade de software antes da realização dos testes do software.

Contudo, a seleção de um modelo de confiabilidade pode seguir um procedimento geral, compreendendo os passos descritos a seguir:

1. Teste do software: esta fase inicial não é trivial nem a mais rápida, pois trata da realização dos testes do software. Requer o planejamento dos testes, a realização dos testes e o registro das falhas.
2. Coleta dos dados: nesta fase os resultados dos testes são coletados, anotados e armazenados. Normalmente, são anotados dados como: o tempo entre a ocorrência das falhas, o tempo acumulado de falhas, o número acumulado de falhas em um período de tempo e a cobertura do teste.
3. Seleção do modelo de confiabilidade: nesta fase verificam-se os modelos que satisfazem as condições em que os testes foram realizados e estimam-se os parâmetros dos modelos.

4. Verificação dos modelos de confiabilidade: entre os modelos selecionados no passo anterior, verifica-se a adequação dos modelos com o uso de testes estatísticos (paramétricos e não paramétricos).
5. Validação do modelo de confiabilidade: entre os modelos verificados, seleciona-se o modelo que melhor se ajusta aos dados, com base no valor crítico do teste estatístico utilizado no passo anterior.
6. Utilização do modelo de confiabilidade: nesta fase utiliza-se o modelo de confiabilidade selecionado para a tomada de decisões sobre o software. Podem ser calculadas estimativas da confiabilidade corrente do software ou estimativas do tempo para a ocorrência das próximas falhas.

Em geral, a seleção de um modelo de confiabilidade do software requer o uso de uma ferramenta. Existem algumas ferramentas de domínio público que podem ser utilizadas para a seleção dos modelos e a estimação de seus parâmetros.

Uma ferramenta bastante conhecida é o SMERFS (Statistical Modeling and Estimation of Reliability Functions for Systems) [136]. É uma ferramenta que teve seu desenvolvimento iniciado em 1981, patrocinado pelo departamento de pesquisa naval do governo dos Estados Unidos (NSWC – Naval Surface Weapons Center). Atualmente, é utilizada por diversas empresas e instituições como a NASA (Agência Espacial Norte-Americana) em seus projetos de exploração espacial que exigem softwares com um padrão elevado de confiabilidade. O SMERFS, por meio de técnicas estatísticas, permite avaliar o ajuste de diversos modelos de confiabilidade, assim como estimar os parâmetros desses modelos. Inicialmente, a versão foi desenvolvida para o ambiente DOS, mas já existe uma versão para o ambiente operacional Windows.

Outra ferramenta também de domínio público muito conhecida é o CASRE (Computer Aided Software Reliability Estimation) [255]. É uma ferramenta caracterizada pela sua fácil utilização e grande interação com o usuário. A ferramenta CASRE incorpora praticamente todos os modelos implantados no SMERFS.

A SoRel (Software Reliability) é uma ferramenta desenvolvida pelo LAAS, um laboratório da “National Center for Scientific Research” em Toulouse na França [255]. Essa ferramenta foi primeiramente desenvolvida em 1991 para ser operada em uma plataforma Macintosh II com um coprocessador matemático. A SoRel é composta por duas partes. A primeira parte permite aplicar vários testes de tendência da confiabilidade, tais quais: teste aritmético, o teste de Laplace, o teste de Kendall e o teste de Sperman. Esses testes permitem uma análise para identificar se os dados apontam para uma confiabilidade crescente ou decrescente e, então, aplicar um modelo de confiabilidade apropriado. A escolha de um modelo é validada por meio de três critérios estatísticos. A segunda parte permite a aplicação do modelo de confiabilidade escolhido. Uma limitação é que apenas quatro modelos de confiabilidade estão implantados na ferramenta.

Existem outras ferramentas menos conhecidas [255], como: a SRMP (Statistical Modeling and Reliability Program), desenvolvida por consultores de estatística e confiabilidade para ser utilizada em uma plataforma UNIX, e a SARA (Software Assurance Reliability Automation), um sistema que incorpora a modelagem do crescimento da confiabilidade e métrica do código para analisar o tempo entre falhas do software.

Uma análise detalhada de ferramentas sobre confiabilidade de software pode ser vista na publicação de Lyu [255].

### 13.5.2 Aplicações de medidas da confiabilidade

Basicamente, existem duas situações importantes de tomadas de decisão que precisam ser investigadas com a ajuda dos modelos de confiabilidade de software, isto é, situações em que o uso dos modelos de confiabilidade de software é indispensável:

- Liberação do software

O software está pronto para ser liberado? O nível de confiabilidade do software atingido nos testes já realizados pode ser um critério para a liberação do software.

No início do teste, ocorre um número significativo de falhas. A remoção dos defeitos que provocaram essas falhas pode gerar um crescimento significativo da confiabilidade do software. Após essa fase inicial de aumento substancial da confiabilidade, atinge-se um patamar em que o aumento da confiabilidade do software ocorre de forma muito lenta. O processo de remoção de defeitos prossegue até atingir um nível de confiabilidade desejado.

- Teste

Duas questões podem ser feitas em relação ao teste. Se o teste do software é baseado no tempo: quanto tempo de teste ainda é necessário para se atingir a confiabilidade desejada no software? Se o teste do software é baseado no número de dados de teste: quantos dados de teste ainda são necessários para se atingir a confiabilidade desejada no software?

O crescimento da confiabilidade considerada como função do tempo ou como função do número de dados de teste indica que qualquer aumento desejado na confiabilidade do software pode requerer um tempo de teste excessivamente longo ou um grande número de dados de teste.

A Figura 13.9 ilustra o crescimento da confiabilidade do software como uma função do tempo de teste.

O critério de confiabilidade mínima desejada para o software é um critério extremamente simples e fácil de ser aplicado. Entretanto, não considera o custo dos testes adicionais necessários para se atingir a confiabilidade desejada. O custo dos testes adicionais para se atingir um nível desejado de confiabilidade para o software pode não ser viável para o projeto em questão.

Devem-se levar em conta dois fatores para a avaliação de custos. O custo do teste do software e o custo de manutenção ou custo de falha no caso de uma liberação do software sem que se tenha atingido o nível de confiabilidade desejado. Esses dois tipos de custos variam em sentidos opostos. O ideal é fazer uma avaliação global que envolva o custo dos testes e o custo de falha.

A Figura 13.10 ilustra o comportamento da função Custo Global. O ideal é a localização do ponto mínimo, ou pelo menos a região próxima ao ponto mínimo do custo global.

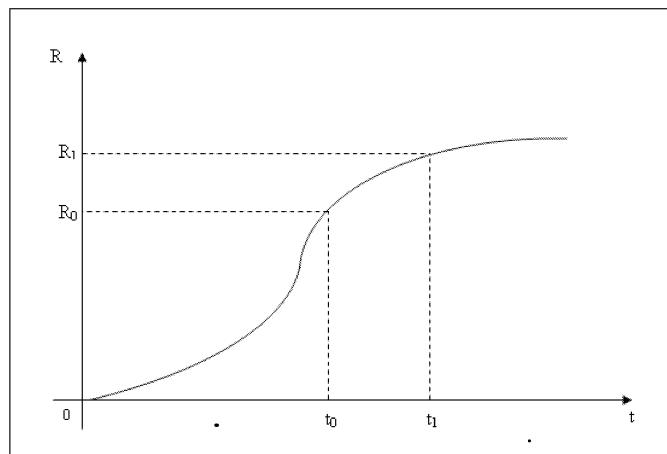


Figura 13.9 – Confiabilidade de software como função do tempo de teste.

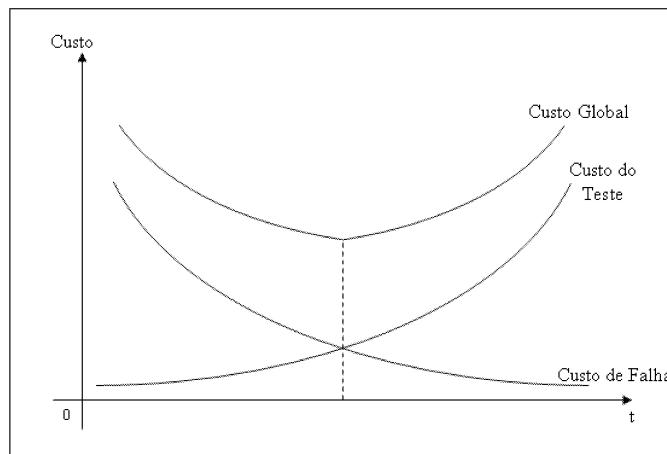


Figura 13.10 – Custos da confiabilidade de software.

O momento de liberação do software deve ser tal que minimize a soma desses dois componentes do custo.

## 13.6 Considerações finais

Neste capítulo apresentamos inicialmente a noção de confiabilidade de software, a confiabilidade no contexto da qualidade de software e sua importância no desenvolvimento de sistemas críticos.

A teoria de confiabilidade de software lida com métodos probabilísticos aplicados para analisar a ocorrência aleatória de falhas em um sistema de software. Os conceitos apresentados fundamentam e formalizam a teoria da confiabilidade de software: função Confiabilidade, função Taxa de Falhas, função de Falhas Acumuladas, função Intensidade de Falhas e Tempo Médio para Falhas. Discutimos brevemente a medição de confiabilidade de software e apresentamos as principais distribuições de probabilidade utilizadas na teoria de confiabilidade – distribuição exponencial, distribuição de Weibull e a distribuição Gamma.

Aspectos gerais sobre modelos de confiabilidade de software são discutidos, iniciando com a fundamentação básica da modelagem da confiabilidade de software. São apresentadas as principais classificações dos modelos de confiabilidade de software encontradas na literatura.

Alguns dos principais modelos de confiabilidade de software amplamente citados na literatura são discutidos, segundo as três principais abordagens clássicas – modelos de implante de defeitos, modelos baseados no domínio de dados e modelos baseados no domínio do tempo. São também discutidos os modelos baseados em cobertura de teste, uma nova abordagem de modelos de confiabilidade de software. As limitações dos modelos de confiabilidade também são discutidas.

Finalmente, apresentamos um procedimento geral para o cálculo da confiabilidade de software e algumas ferramentas para seleção de modelos de confiabilidade e estimação de seus parâmetros. Apresentamos resumidamente aplicações de medidas da confiabilidade de software.

Novas abordagens em confiabilidade de software e áreas importantes de aplicação de confiabilidade de software, temas de pesquisa e desenvolvimento incluem: confiabilidade de software baseada em arquitetura; confiabilidade de sistemas baseados em componentes; confiabilidade de servidores e aplicações *Web*; confiabilidade de aplicações móveis.

## Referências Bibliográficas

- [1] A. Abdurazik e J. Offutt. Using uml collaboration diagrams for static checking and test generation. In: *3rd International Conference on the Unified Modeling Language – UML'00 / LNCS*, volume 1939, p. 383-395, York, UK, out. 2000. Springer Berlin/Heidelberg.
- [2] A. T. Acree. *On Mutation*. Tese de doutoramento, Georgia Institute of Technology, Atlanta, GA, EUA, ago. 1980.
- [3] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, e F. G. Sayward. Mutation analysis. Technical Report GIT-ICS-79/08, Georgia Institute of Technology, Atlanta, GA, set. 1979.
- [4] E. Adams e S. S. Muchnick. Dbxtool: A window-based symbolic debugger for sun workstation. *Software Practice and Experience*, 16(7):653-669, jul. 1986.
- [5] H. Agrawal. *Towards Automatic Debugging of Computer Programs*. Tese de doutoramento, Purdue University, West Lafayette, IN, EUA, set. 1991.
- [6] H. Agrawal, J. Alberi, J. R. Horgan, J. Li, S. London, W. E. Wong, S. Ghosh, e N. Wilde. Mining system tests to aid software maintenance. *IEEE Computer*, 31(7):64-73, jul. 1998.
- [7] H. Agrawal, R. A. DeMillo, R. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, e E. H. Spafford. Design of mutant operators for the C programming language. Technical Report SERC-TR41-P, Software Engineering Research Center, Purdue University, West Lafayette, IN, EUA, mar. 1989.
- [8] H. Agrawal, R. A. DeMillo, e E. H. Spafford. An execution-backtracking approach to debugging. *IEEE Software*, 8(3):21-26, mai.1991.
- [9] H. Agrawal e J. R. Horgan. Dynamic program slicing. *ACM SIGPLAN Notices*, 25(6):246-256, jun. 1990.
- [10] H. Agrawal, J. R. Horgan, S. London, e W. E. Wong. Fault localization using execution slices and dataflow tests. In: *6th International Symposium on Software Reliability Engineering*, p. 143-151, Toulouse, França, out. 1995. IEEE Computer Society Press.
- [11] R. T. Alexander, J. M. Bieman, e A. A. Andrews. Towards the systematic testing of aspect-oriented programs. Technical report, Colorado State University, Fort Collins, CO, EUA, 2004. Disponível on-line: <http://www.cs.colostate.edu/~rta/publications/CS-04-105.pdf>.

- [12] R. T. Alexander, J. M. Bieman, S. Ghosh, e B. Ji. Mutation of Java objects. In: *13th International Symposium on Software Reliability Engineering – ISSRE’2002*, p. 341-351, Annapolis, MD, EUA, nov. 2002. IEEE Computer Society Press.
- [13] G. S. Almasi e A. Gottlieb. *Highly Parallel Computing*. The Benjamin Cummings Publishing Company, 2. ed., 1994.
- [14] G.R. Andrews e F.B. Schineider. Concepts and notations for concurrent programming. *ACM Computing Survey*, 15(1):3-43, mar. 1983.
- [15] L. Apfelbaum e J. Doyle. Model based testing. In: *10th International Software Quality Week*. Software Research Institute, mai.1997.
- [16] K. Araki, Z. Furukawa e J. Cheng. A general framework for debugging. *IEEE Software*, 8(3):14-20, mai.1991.
- [17] T. R. Arnold e W. A. Fuson. Testing “in a perfect world”. *Communications of the ACM*, 37(9):78-86, set. 1994.
- [18] M. Auguston. A program behavior model based on event grammar and its application for debugging automation. In: *2nd International Workshop on Automated and Algorithmic Debugging*, p. 277-291, Saint-Malo, França, mai.1995.
- [19] T. M. Austin, S. E. Breach e G. S. Sohi. Efficient detection of all pointer and array access errors. *ACM SIGPLAN Notes*, 29(6):290-301, jun. 1994.
- [20] D. Baldwin e F. Sayward. Heuristics for determining equivalence of program mutations. Research Report 276, Department of Computer Science, Yale University, New Haven, CT, EUA, 1979.
- [21] R. M. Balzer. Exdams: Extensible debugging and monitoring system. In: *Spring Joint Computer Conference*, p. 567-589, Reston, VA, EUA, 1969. AFIPS Press.
- [22] S. Barbey e A. Strohmeier. The problematics of testing object-oriented software. In: *2nd Conference on Software Quality Management – SQM’94*, volume 2, p. 411-426, jul. 1994.
- [23] F. Barbier, N. Belloir e J.-M. Bruel. Incorporation of test functionality into software components. In: *2nd International Conference on COTS-Based Software Systems*, volume 2580 de *Lecture Notes in Computer Science*, p. 25-35, Londres, UK, fev. 2003. Springer-Verlag.
- [24] E. F. Barbosa. Uma contribuição para a determinação de um conjunto essencial de operadores de mutação no teste de programas C. Dissertação de mestrado, ICMC-USP, São Carlos, SP, Brasil, nov. 1998. Disponível on-line: <http://www.teses.usp.br/teses/disponiveis/55/55134/tde-05022001-165802/>.
- [25] E. F. Barbosa, J. C. Maldonado e A. M. R. Vincenzi. Towards the determination of sufficient mutant operators for C. *Software Testing, Verification and Reliability*, 11(2):113-136, jun. 2001.
- [26] V. R. Basili e R. W. Selby. Comparing the effectiveness of software testing strategies. *IEEE Transactions on Software Engineering*, 13(12):1.278-1.296, dez. 1987.

- [27] S. L. Basin. Estimation of software error rates via capture-recapture sampling. Technical report, Science Applications, Inc., Palo Alto, CA, EUA, set. 1973.
- [28] K. Beck e E. Gamma. JUnit cookbook. Página WEB, 2006. Disponível em: <http://junit.sourceforge.net/>.
- [29] O. Beckman e B. Gupta. Developing test cases from use cases for web applications. In: *International Conference on Practical Software Testing Techniques – PSTT'2002*, Nova Orleans, 2002.
- [30] A. L. Beguelin. Xab: A tool for monitoring pvm programs. In: *26th Hawaii International Conference on System Sciences*, volume 2, p. 102-103. IEEE Press, jan. 1993.
- [31] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold Company, Nova York, NY, EUA, 2. ed., 1990.
- [32] S. Beydeda e V. Gruhn. An integrated testing technique for component-based software. In: *1st AICCSA ACS/IEEE International Conference on Computer Systems and Applications*, p. 328-334, Beirute, Líbano, jun. 2001. IEEE Computer Society Press.
- [33] S. Beydeda e V. Gruhn. State of the art in testing components. In: *Third International Conference on Quality Software – QSIC'03*, p. 146-153, Washington, DC, EUA, 2003. IEEE Computer Society.
- [34] J. M. Bieman, S. Ghosh e R. T. Alexander. A technique for mutation of Java objects. In: *16th IEEE International Conference on Automated Software Engineering*, p. 23-26, San Diego, CA, EUA, nov. 2001. IEEE Computer Society.
- [35] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*, volume 1. Addison Wesley Longman, Inc., 1999.
- [36] A.S. Binns e G. McGraw. Building a Java software engineering tool for testing applets. In: *IntraNet 96 NY Conference*, Nova York, NY, EUA, abr. 1996.
- [37] D. L. Bird e C. U. Munoz. Automatic generation of random self-checking test cases. *IBM System Journal*, 22(3):229-245, 1983.
- [38] P.V. Biron, K. Permanente e A. Malhotra. Xml schema part 2: Datatypes second edition – W3C recommendation. Página WEB, out. 2004. W3C – World Wide Web Consortium. Disponível em: <http://www.w3.org/TR/xmleschema-2/>.
- [39] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie e J. Siméon. Xquery 1.0: An xml query language – W3C recommendation. Página WEB, jan. 2007. W3C – World Wide Web Consortium. Disponível em: <http://www.w3.org/TR/xquery/>.
- [40] T. L. Booth. *Sequential Machines and Automata Theory*. Wiley, 1967.
- [41] B. Boothe. Efficient algorithms for bidirectional debugging. In: *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, p. 299-310, Nova York, NY, EUA, jun. 2000. ACM Press.

- [42] L. Bottaci. A genetic algorithm fitness function for mutation testing. In: *Seminal: Software Engineering Using Metaheuristic Innovative Algorithms – Meeting 7*. IEEE International Conference on Software Engineering, Toronto, Canadá, mai.2001. Disponível em: [http://www.dcs.kcl.ac.uk/projects/seminal/pastmeeting/\(007\)\(12,13\)-5-2001/bottaci.ps](http://www.dcs.kcl.ac.uk/projects/seminal/pastmeeting/(007)(12,13)-5-2001/bottaci.ps).
- [43] D. C. Bowen e J. M. Shukal. Software reliability. *RCA Engineer*, 25:15-18, 1979.
- [44] R. S. Boyer, B. Elspas e K. N. Levitt. Select – a formal system for testing and debugging programs by symbolic execution. In: *International Conference on Reliable software*, p. 234-245, Nova York, NY, EUA, 1975. ACM Press.
- [45] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler e F. Yergeau. XML – extensible markup language 1.0 (third edition) – W3C recommendation. Technical report, fev. 2004. Disponível em: <http://www.w3.org/TR/2004/REC-xml-20040204/>.
- [46] L. C. Briand, Y. Labiche e Y. Wang. An investigation of graph-based class integration test order strategies. *IEEE Transactions on Software Engineering*, 29(7):594-607, jul. 2003.
- [47] J. Brown e M. Lipow. Testing for software reliability. In: *Proceedings of the International Conference on Reliable Software*, 1975.
- [48] J. M. Bruel, J. Araújo, A. Moreira e A. Royer. Using aspects to develop built-in tests for components. In: *The 4th AOSD Modeling With UML Workshop*, São Francisco, CA, EUA, out. 2003.
- [49] T. A. Budd. *Mutation Analysis: Ideas, Example, Problems and Prospects*, chapter Computer Program Testing. North-Holland Publishing Company, 1981.
- [50] T. A. Budd e D. Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18(1):31-45, nov. 1982.
- [51] T. A. Budd, R. A. DeMillo, R. J. Lipton e F. G. Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In: *7th ACM Symposium on Principles of Programming Languages*, p. 220-233, Nova York, NY, EUA, jan. 1980.
- [52] S. Budkowski e P. Dembinski. An introduction to Estelle: a specification language for distributed systems. *Computer Network and ISDN Systems*, 14(1):3-23, 1987.
- [53] P. M. S. Bueno e M. Jino. Automated test data generation for program paths using genetic algorithms. In: *13th International Conference on Software Engineering & Knowledge Engineering – SEKE'2001*, p. 2-9, Buenos Aires, Argentina, jun. 2001.
- [54] G. A. Bundell, G. Lee, J. Morris, K. Parker e P. Lam. A software component verification tool. In: *1st International Conference on Software Methods and Tools (SMT'2000)*, p. 137-147, Wollongong, Australia, nov. 2000. IEEE Computer Society Press.
- [55] R.M. Butler e E.L.Lusk. Monitors, messages and clusters: The P4 parallel programming system. *Parallel Computing*, 20(4):547-564, abr. 1994.

- [56] M. Bybro. A mutation testing tool for Java programs. Dissertação de mestrado, Stockholm University, Estocolmo, Suécia, ago. 2003.
- [57] R. Calkin, R. Hempel, H.-C. Hoppe e P. Wypior. Portable programming with the parmacs message-passing library. *Parallel Computing*, 20(4):615-632, 1994.
- [58] L. F. Capretz. A brief history of the object-oriented approach. *SIGSOFT Softw. Eng. Notes*, 28(2):6, mar. 2003.
- [59] R. H. Carver e K.-C. Tai. Replay and testing for concurrent programs. *IEEE Software*, 8(2):66-74, mar. 1991.
- [60] M. Ceccato, P. Tonella e F. Ricca. Is aop code easier or harder to test than OOP code? In: *Fourth International Conference on Aspect-Oriented Software Development (AOSD'2005) – Workshop On Testing Aspect Oriented Programs*, Chicago, Illinois, EUA, mar. 2005.
- [61] M. J. Chaim. Poke-tool – uma ferramenta para suporte ao teste estrutural de programas baseados em análise de fluxo de dados. Dissertação de mestrado, DCA/FEE/UNICAMP – Campinas, SP, Brasil, abr. 1991.
- [62] M. L. Chaim, A. Carniello e M. Jino. Teste baseado em casos de uso. Boletim de Pesquisa e Desenvolvimento, dez. 2003. Disponível em: <http://www.cnptia.embrapa.br/modules/tinycontent3/content/2003/bp10.pdf>.
- [63] M. L. Chaim, J. C. Maldonado e M. Jino. On the use of dynamic data-flow testing information for fault localization. In: *Proceeding of the Workshop on Software Quality of the ACM/IEEE International Conference on Software Engineering*, mai.2002.
- [64] M. L. Chaim, J. C. Maldonado e M. Jino. A debugging strategy based on the requirements of testing. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(4-5):277-308, jul./out. 2004.
- [65] C. Chambers, I. Pechtchanski, V. Sarkar, M. J. Serrano e H. Srinivasan. Dependence analysis for Java. In: *LCPC '99: Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, volume 1863 of *Lecture Notes in Computer Science*, p. 35-52, Londres, UK, 2000. Springer-Verlag.
- [66] T. W. Chan. A framework for debugging. *Journal of Computer Information Systems*, 38(1):67-73, 1997.
- [67] H. Y. Chen, T. H. Tse, F. T. Chan e T. Y. Chen. In: black and white: an integrated approach to class-level testing of object-oriented programs. *ACM Transactions on Software Engineering Methodology*, 7(3):250-295, jul. 1998.
- [68] M.-H. Chen. *Tools and techniques for testing based software reliability estimation*. Tese de doutoramento, Purdue University, West Lafayette, IN, EUA, 1994.
- [69] M.-H. Chen e H. M. Kao. Testing object-oriented programs – an integrated approach. In: *10th International Symposium on Software Reliability Engineering*, p. 73-83, Washington, DC, USA, nov. 1999. IEEE Computer Society.

- [70] M.-H. Chen, M. R. Lyu e W. E. Wong. An empirical study of the correlation between code coverage and reliability estimation. In: *METRICS '96: Proceedings of the 3rd International Symposium on Software Metrics*, p. 133-141, Washington, DC, EUA, mar. 1996. IEEE Computer Society.
- [71] M.-H. Chen, M. R. Lyu e W. E. Wong. Effect of code coverage on software reliability measurement. *IEEE Transactions on Reliability*, 50(2):165-170, jun. 2001.
- [72] M.-H. Chen, A. P. Mathur e V. J. Rego. Effect of testing technique on software reliability estimates obtained using a time-domain model. *IEEE Transactions on Reliability*, 44(1):97-103, mar. 1995.
- [73] S.-K. Chen, W. K. Fuchs e J.-Y. Chung. Reversible debugging using program instrumentation. *IEEE Transactions on Software Engineering*, 27(8):715-727, ago. 2001.
- [74] T. Y. Chen e Y. Y. Cheung. On program dicing. *Journal of Software Maintenance*, 9(1):33-46, 1997.
- [75] P. Chevalley. Applying mutation analysis for object-oriented programs using a reflective approach. In: *8th Asia-Pacific Software Engineering Conference – APSEC'01*, p. 267-272, Macau, China, dez. 2001. IEEE Computer Press.
- [76] B. J. Choi, R. A. DeMillo, E. W. Krauser, R. J. Martin, A. P. Mathur, A. J. Offutt, H. Pan e E. H. Spafford. The mothra toolset. In: *Proceedings of the 22nd Annual Hawaii International Conference on Systems Sciences*, p. 275-284, Koa, Havaí, jan. 1989.
- [77] B. J. Choi, A. P. Mathur e A. P. Pattison. pmothra: Scheduling mutants for execution on a hypercube. In: *3rd Symposium on Software Testing, Analysis and Verification*, p. 58-65, Key West, FL, dez. 1989. ACM Press.
- [78] C. S. Chou e M. W. Du. Improved domain strategies for detecting path selection errors. In: *International Conference on Software Maintenance*, p. 165-173, Los Angeles, CA, EUA, set. 1987.
- [79] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178-187, mai. 1978.
- [80] I. S. Chung. Automatic testing generation for mutation testing using genetic operators. In: *International Conference on Software Engineering and Knowledge Engineering*. São Francisco, CA, EUA, jun. 1998.
- [81] R. D. Yang C. G. Chung. Path analysis testing of concurrent programs. *Information and Software Technology*, 34(1):43-56, jan. 1992.
- [82] T. Chusho. Test data selection and quality estimation based on the concept of essential branches for path testing. *IEEE Transactions on Software Engineering*, 13(5):509-517, mai. 1987.
- [83] L. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2(3):215-222, set. 1976.
- [84] L. A. Clarke, J. Hassell e D. J. Richardson. A close look at domain testing. *IEEE Transactions on Software Engineering*, 8(4):380-390, jul. 1982.

- [85] L. A. Clarke e D. J. Richardson. The application of error-sensitive testing strategies to debugging. *ACM SIGSOFT Software Engineering Notes*, 8(4):45-52, ago. 1983.
- [86] P. J. Clarke e B. A. Malloy. A taxonomy of classes for implementation-based testing. Technical report, Clemson University, mai.2001.
- [87] H. Cleve e A. Zeller. Locating causes of program failures. In: *ICSE '05: Proceedings of the 27th international conference on Software engineering*, p. 342-351, mai.2005.
- [88] T. E. Colanzi. Uma abordagem integrada de desenvolvimento e teste de software baseada na UML. Dissertação de mestrado, ICMC-USP, São Carlos, SP, Brasil, jun. 1999.
- [89] J. S. Collofello e L. Cousins. Toward automatic software fault localization through decision-to-decision path analysis. In: *Proceedings of the AFIP 1987 National Computer Conference*, p. 539-544, Chicago, IL, EUA, mai.1987.
- [90] J. Conallen. Modeling web application architectures with UML. *Communications of the ACM*, 42(10):63-70, out. 1999.
- [91] J. Conallen. *Building Web applications with UML*. Addison-Wesley, Boston, MA, EUA, 2. ed., out. 2002.
- [92] L. Copeland. *A Practitioner's Guide to Software Test Design*. Artech House Publishers, 2004.
- [93] W. J. Corcoran, H. Weingarten e P. W. Zehna. Estimating reliability after corrective action. *Management Science*, 10(4):786-795, jul. 1964.
- [94] B. Cornelius. Java versus C++. Documento on-line, abr. 1997. Disponível em: <http://www.dur.ac.uk/barry.cornelius/Java/java.versus.c++/>.
- [95] Parasoft Corporation. Using design by contract to automate Java software and componenttesting. Página WEB, 2002. Disponível em: [http://www.parasoft.com/jsp/printables/Using\\_Design\\_by\\_Contract.pdf](http://www.parasoft.com/jsp/printables/Using_Design_by_Contract.pdf).
- [96] W. M. Craft. Detecting equivalents mutants using compiler optimization. Dissertação de mestrado, Clemson University, Clemson, SC, EUA, 1989.
- [97] A. N. Crespo. Cobertura dos critérios potenciais-usos e a confiabilidade do software. Technical report, DCA/FEEC/UNICAMP, Campinas, SP, Brasil, 1996.
- [98] A. N. Crespo. *Modelos de Confiabilidade de Software Baseados em Cobertura de Critérios Estruturais de Teste*. Tese de doutoramento, DCA/FEEC/UNICAMP, Campinas, SP, Brasil, 1997.
- [99] A. N. Crespo, P. Matrella e A. Pasquini. Sensitivity of reliability growth models to operational profile errors. In: *ISSRE '96: Proceedings of the The Seventh International Symposium on Software Reliability Engineering (ISSRE '96)*, p. 35-44, Washington, DC, USA, out./nov. 1996. IEEE Computer Society.
- [100] A. N. Crespo, A. Pasquini, M. Jino e J. C. Maldonado. A binomial software reliability model based on coverage of structural testing criteria. In: *XIV Simpósio Brasileiro de Engenharia de Software – SBES'2000*, p. 211-226, out. 2000.

- [101] I. D.D. Curcio. ASAP – a simple assertion pre-processor. *ACM SIGPLAN Notices*, 33(12):44-51, dez. 1998.
- [102] A. R. C. da Rocha, J. C. Maldonado e K. C. Weber. *Qualidade de Software: Teoria e Prática*. Prentice Hall, São Paulo, SP, Brasil, 2001.
- [103] S. K. Damodaran-Kamal e J. M. Francioni. Nondeterminacy: Testing and debugging in message passing parallel programs. In: *III ACM/ONR Workshop on Parallel and Distributed Debugging*, p. 118-128. ACM Press, Nova York, NY, EUA, 1993.
- [104] C. Darwin. *On the Origin of Species by Means of Natural Selection or the Preservation of Favoured Races in the Struggle for Life*. 1859. Disponível em: <http://etext.virginia.edu/toc/modeng/public/DarOrig.html>.
- [105] A. M. Davis. A comparison of techniques for the specification of external system behavior. *Communications of the ACM*, 31(9):1.098-1.115, set. 1988.
- [106] J. de S. Coutinho. Software reliability growth. In: *IEEE Symposium on Computer Software Reliability*, 1973.
- [107] M. E. Delamaro. Proteum: Um ambiente de teste baseado na análise de mutantes. Dissertação de mestrado, ICMC/USP, São Carlos, SP, Brasil, out. 1993.
- [108] M. E. Delamaro. *Mutação de Interface: Um Critério de Adequação Interprocedimental para o Teste de Integração*. Tese de doutoramento, IFSC/USP, São Carlos, SP, Brasil, jun. 1997.
- [109] M. E. Delamaro, J. C. Maldonado, M. Jino e M. L. Chaim. Proteum: Uma ferramenta de teste baseada na análise de mutantes. In: *Caderno de Ferramentas do VII Simpósio Brasileiro de Engenharia de Software*, p. 31-33, Rio de Janeiro, RJ, Brasil, out. 1993.
- [110] M. E. Delamaro, J. C. Maldonado e A. P. Mathur. Interface mutation: An approach for integration testing. *IEEE Transactions on Software Engineering*, 27(3):228-247, mar. 2001.
- [111] M. E. Delamaro, M. Pezzè, A. M. R. Vincenzi e J. C. Maldonado. Mutant operators for testing concurrent Java programs. In: *XV Simpósio Brasileiro de Engenharia de Software – SBES'2001*, p. 272-285, Rio de Janeiro, RJ, Brasil, out. 2001.
- [112] M. E. Delamaro e A. M. R. Vincenzi. Structural testing of mobile agents. In: Egidio Astesiano Nicolas Guelfi and Gianna Reggio, editors, *III International Workshop on Scientific Engineering of Java Distributed Applications (FIDJI'2003)*, Lecture Notes on Computer Science, p. 73-85, Springer, nov. 2003.
- [113] M. E. Delamaro, A. M. R. Vincenzi e J. C. Maldonado. A strategy to perform coverage testing of mobile applications. In: *Workshop on Automation of Software Test – AST'2006*, p. 118-124, Xangai, China, mai.2006. ACM Press.
- [114] R. A. DeMillo, D. C. Gwind e K. N. King. An extended overview of the Mothra software testing environment. In: *II Workshop on Software Testing, Verification and Analysis*, p. 142-151. Computer Science Press, Banff, Canadá, jul. 1988.
- [115] R. A. DeMillo, R. J. Lipton e F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34-41, abr. 1978.

- [116] R. A. DeMillo e A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900-910, set. 1991.
- [117] R. A. DeMillo, H. Pan e E. H. Spafford. Critical slicing for software fault localization. In: *ISSTA '96: Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*, p. 121-134, Nova York, NY, EUA, 1996. ACM Press.
- [118] M. S. Deutsch. *Software Verification and Validation*. Prentice-Hall, Englewood Cliffs, NJ, EUA, 1982.
- [119] M. Doliner. Projeto cobertura. Página WEB, 2006. Disponível em: <http://cobertura.sourceforge.net/>.
- [120] A. L. S. Domingues. Avaliação de critérios e ferramentas de teste para programas OO. Dissertação de mestrado, ICMC/USP, São Carlos, SP, Brasil, jun. 2002. Disponível em: <http://www.teses.usp.br/teses/disponiveis/55/55134/tde-28112002-171043/>.
- [121] M. Ducassé. Coca: An automated debugger for C. In: *1999 International Conference on Software Engineering*, p. 504-513, Los Angeles, CA, EUA, mai.1999. IEEE Computer Society Press.
- [122] J. W. Duran e S. C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, 10(4), jul. 1984.
- [123] E. S. Dória. Replicação de estudos empíricos em engenharia de software. Dissertação de mestrado, ICMC/USP, São Carlos, SP, Brasil, mai.2001. Disponível em: <http://www.teses.usp.br/teses/disponiveis/55/55134/tde-10052002-103851/>.
- [124] Tester's Edge. Glass JAR toolkit. Página WEB, 2002. Disponível em: <http://www.testersedge.com/gjtk/>.
- [125] S. H. Edwards. Black-box testing using flowgraphs: An experimental assessment of effectiveness and automation potential. *Software Testing, Verification and Reliability*, 10(4):249-262, dez. 2000.
- [126] S. H. Edwards. A framework for practical, automated black-box testing of component-based software. *Software Testing, Verification and Reliability*, 11(2):97-111, jun. 2001.
- [127] S. H. Edwards. Toward reflective metadata wrappers for formally specified software components. In: *1st Workshop on Specification and Verification of Component-Based Systems – affiliated with OOPSLA'2001*, p. 14-21, Tampa, FL, EUA, out. 2001. ACM Press.
- [128] T. Elrad, M. Aksit, G. Kiczales, K. Lieberherr e H. Ossher. Discussing aspects of AOP. *Communications of the ACM*, 44(10):33-38, 2001.
- [129] M. C. F. P. Emer, S. R. Vergilio e M. Jino. A testing approach for xml schemas. In: *XXIX Annual International Computer Software and Applications Conference, COMP-SAC 2005 – QATWBA 2005*, volume 2, p. 57-62. IEEE Press, jul. 2005.

- [130] S. C. P. F. Fabbri. *A Análise de Mutantes no Contexto de Sistemas Reativos: Uma Contribuição para o Estabelecimento de Estratégias de Teste e Validação*. Tese de doutoramento, IFSC/USP, São Carlos, SP, Brasil, out. 1996.
- [131] S. C. P. F. Fabbri, J. C. Maldonado, M. E. Delamaro e P. C. Masiero. 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*, p. 96-104, Los Alamitos, CA, EUA, nov. 1999. IEEE Computer Society.
- [132] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero e M. E. Delamaro. Análise de mutantes baseada em máquinas de estado finito. In: *XI Simpósio Brasileiro de Redes de Computadores – SBRC'93*, p. 407-425, Campinas, SP, Brasil, mai.1993.
- [133] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero e M. E. Delamaro. Mutation analysis testing for finite state machines. In: *5th International Symposium on Software Reliability Engineering – ISSRE'94*, p. 220-229, Monterey – CA, nov. 1994. IEEE Computer Society Press.
- [134] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero e M. E. Delamaro. Mutation analysis applied to validate specifications based on petri nets. In: *FORTE'95 – 8th IFIP Conference on Formal Descriptions Techniques for Distribute Systems and Communication Protocols*, p. 329-337, Montreal, Canadá, out. 1995. Kluwer Academic Publishers.
- [135] S. C. P. F. Fabbri, J. C. Maldonado, T. Sugeta e P. C. Masiero. Mutation testing applied to validate specifications based on statecharts. In: *X International Symposium on Software Reliability Systems – ISSRE'99*, p. 210-219, Washington, DC, EUA, nov. 1999. IEEE Computer Society.
- [136] W. H. Farr e O. D. Smith. Statistical modeling and estimation of reliability functions for software (smerfs) user's guide. Relatório Técnico ADA275390, Naval Surface Warfare Center – NSWC, Dahlgren, VI, EUA, set. 1993. Disponível em: <http://handle.dtic.mil/100.2/ADA275390>.
- [137] L.P. Ferreira e S.R. Vergilio. Tdsgen: An environment based on hybrid genetic algorithms for generation of test data. In: *17th International Conference on Software Engineering and Knowledge Engineering*, volume 3103/2004, p. 1.431-1.432, Springer, 2005.
- [138] R. Filman e D. Friedman. Aspect-oriented programming is quantification and obliviousness. In: *Workshop on Advanced Separation of Concerns – OOPSLA'2000*, Minneapolis, MN, EUA, out. 2000.
- [139] J. Flower e A. Kolawa. Express is not just a message passing system. current and future directions in express. *Parallel Computing*, 20(4):597-614, abr. 1994.
- [140] R. P. Fonseca. Suporte ao teste estrutural de programas fortran no ambiente poke-tool. Dissertação de mestrado, DCA/FEEC/UNICAMP, Campinas, SP, Brasil, jan. 1993.
- [141] F. G. Frankl. *The Use of Data Flow Information for the Selection and Evaluation of Software Test Data*. Tese de doutoramento, Nova York University, Nova York, NY, EUA, out. 1987.

- [142] F. G. Frankl e E. J. Weyuker. A data flow testing tool. In: *II Conference on Software development tools, techniques, and alternatives*, p. 46-53, Los Alamitos, CA, EUA, dez. 1985. IEEE Computer Society Press.
- [143] F. G. Frankl e E. J. Weyuker. Data flow testing in the presence of unexecutable paths. In: *Workshop on Software Testing*, p. 4-13, Banff, Canadá, jul. 1986.
- [144] P. G. Frankl e E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1.483-1.498, out. 1988.
- [145] P. G. Frankl e E. J. Weyuker. A formal analysis of the fault-detecting ability of testing methods. *IEEE Transactions on Software Engineering*, 19(3):202-213, mar. 1993.
- [146] P. Fritzson, N. Shahmehri, M. Kamkar e T. Gyimothy. Generalized algorithmic debugging and testing. *ACM Letters on Programming Languages and Systems*, 1(4):303-322, 1992.
- [147] C. Gane e T. Sarson. *Análise estruturada de sistemas*. LTC Editora, Rio de Janeiro, RJ, Brasil, 1983.
- [148] P. Garg. Investigating coverage-reliability relationship and sensitivity of reliability to errors in the operational profile. In: *I International Conference on Software Testing, Reliability and Quality Assurance*, p. 21-35. IBM Press, dez. 1994.
- [149] A. Geist, A. Beguelin, J. J. Dongarra, W. Jiang, R. Manchek e V. Sunderam. Pvm 3 user's guide and reference manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, Oak Ridge, TN, EUA, mai.1993. Disponível em: <http://www.ugcs.caltech.edu/manuals/other/pvm-3.4.5/PVM-User-Guide.pdf>.
- [150] C. Ghezzi e M. Jazayeri. *Programming Languages Concepts*. John Wiley and Sons, Nova York, NY, EUA, 2. ed., 1987.
- [151] C. Giacometti, S. R. S. Souza e P. S. L. Souza. Teste de mutação para a validação de aplicações concorrentes usando PVM. *Revista Eletrônica de Iniciação Científica – Sociedade Brasileira de Computação*, 3(3), set. 2003. Disponível em: <http://www.sbc.org.br/reic/>.
- [152] A. Gill. *Introduction to the Theory of Finite-State Machine*. McGraw-Hill, Nova York, NY, EUA, 1962.
- [153] I. M. S. Gimenes, L. Barroca, E. H. M. Huzita e A. Carnielo. O processo de desenvolvimento de componentes através de exemplos. In: *VIII Escola Regional de Informática*, p. 1-32, Porto Alegre, RS, Brasil, mai.2000.
- [154] A. L. Goel e K. Okumoto. A time dependent error detection rate model for software reliability and other performance measures. *IEEE Transactions on Reliability*, 28(3):206-211, ago. 1979.
- [155] Al. L. Goel. Software reliability models: Assumptions, limitations and applicability. *IEEE Transactions on Software Engineering*, 11(12):1.411-1.423, dez. 1985.

- [156] M. Golan e D. Hanson. DUEL – A very high-level debugging language. In: *Winter USENIX Technical Conference*, San Diego, CA, EUA, jan. 1993. Disponível em: <http://www273.pair.com/drh/documents/duel.pdf>.
- [157] N. Gupta, H. He, X. Zhang e R. Gupta. Locating faulty code using failure-inducing chops. In: *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, p. 263-272, Nova York, NY, USA, 2005. ACM Press.
- [158] T. Gyimóthy, Á. Beszédes e I. Forgács. An efficient relevant slicing method for debugging. In: *VII European software engineering conference – ESEC/FSE-7*, p. 303-321, Londres, UK, 1999. Springer-Verlag.
- [159] G. Gönenc. A method for the design of fault-detection experiments. *IEEE Transactions on Computers*, 19(6):551-558, jun. 1970.
- [160] A. Haley e S. Zweben. Development and application of a white box approach to integration testing. *Journal of Systems and Software*, 4(4):309-315, nov. 1984.
- [161] D. Hamlet e R. Taylor. Partition testing does not inspire confidence (program testing). *IEEE Transactions on Software Engineering*, 16(12):1.402-1.411, 1990.
- [162] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231-274, jun. 1987.
- [163] D. Harel. Statecharts: On the formal semantics of statecharts. In: *II IEEE Symposium on Logic in Computer Science*, p. 54-64, Ithaca, NY, EUA, 1987. IEEE Press.
- [164] M. Harman, R. Hierons e S. Danicic. The relationship between program dependence and mutation analysis. In: *Mutation 2000 Symposium*, p. 5-12, San Jose, CA, out. 2000. Kluwer Academic Publishers.
- [165] E. R. Harold. Measure test coverage with Cobertura. IBM DeveloperWorks – Artigo On-Line, mai. 2005. Disponível em: <http://www-128.ibm.com/developerworks/java/library/j-cobertura/>.
- [166] M. J. Harrold. Testing: a roadmap. In: *ICSE'00: Proceedings of the Conference on The Future of Software Engineering*, p. 61-72, Nova York, NY, EUA, jun. 2000. ACM Press.
- [167] M. J. Harrold, D. Liang e S. Sinha. An approach to analyzing and testing component-based systems. In: *I Workshop on Testing Distributed Component-Based Systems at ICSE'1999*, Los Angeles, CA, EUA, mai.1999. IEEE Computer Society Press.
- [168] M. J. Harrold, J. D. McGregor e K. J. Fitzpatrick. Incremental testing of object-oriented class structures. In: *XIV International Conference on Software Engineering*, p. 68-80, Nova York, NY, EUA, mai.1992. ACM Press.
- [169] M. J. Harrold e G. Rothermel. Performing data flow testing on classes. In: *II ACM SIGSOFT Symposium on Foundations of Software Engineering*, p. 154-163, Nova York, NY, EUA, dez. 1994. ACM Press.

- [170] M. J. Harrold e M. L. Soffa. Interprocedural data flow testing. In: *III Symposium on Software testing, analysis, and verification*, p. 158-167, Key West, FL, EUA, dez. 1989. ACM Press.
- [171] M. J. Harrold e M. L. Soffa. Selecting and using data for integration testing. *IEEE Software*, 8(2):58-65, mar. 1991.
- [172] J. Hartmann e D. J. Robson. Techniques for selective revalidation. *IEEE Software*, 7(1):31-36, jan. 1990.
- [173] R. Hastings e B. Joyce. Purify: fast detection of memory leaks and access errors. In: *Proceedings of the Winter Usenix Conference*, p. 125-136, 1992.
- [174] M. T. Heath e J. A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29-39, set. 1991.
- [175] H. Hecht. Measurement, estimation and prediction of software reliability. Technical Report CR145135, National Aeronautics and Space Administration – NASA, jan. 1977.
- [176] M. S. Hecht. *Flow Analysis of Computer Programs*. The Computer Science Library: Programming Language Series. Elsevier Science Inc., Nova York, NY, EUA, 1977.
- [177] D. Hedley e M. A. Hennell. The causes and effects of infeasible paths in computer programs. In: *VIII International Conference on Software Engineering*, p. 259-266, Los Alamitos, CA, EUA, 1985. IEEE Computer Society Press.
- [178] P. M. Herman. A data flow analysis approach to program testing. *Australian Computer Journal*, 8(3):92-96, nov. 1976.
- [179] W. C. Hetzel. *An Experimental Analysis of Program Verification Methods*. Tese de doutoramento, University of North Carolina at Chapel Hill, 1976.
- [180] W. C. Hetzel e B. Hetzel. *The Complete Guide to Software Testing*. John Wiley & Sons, Inc., Nova York, NY, EUA, 2. ed., 1991.
- [181] J. Heumann. Is a use case a test case? In: *International Conference on Practical Software Testing Techniques – PSTT'2001*, St. Paul, MN, EUA, 2001.
- [182] R. M. Hierons, M. Harman e S. Danicic. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability*, 9(4):233-262, 1999.
- [183] E. Hilsdale e J. Hugunin. Advice weaving in AspectJ. In: *III International conference on Aspect-oriented software development – AOSD'04*, p. 26-35, Nova York, NY, EUA, 2004. ACM Press.
- [184] D. Hoffman e P. Strooper. A case study in class testing. In: *Conference of the Centre for Advanced Studies on Collaborative research – CASCON'93*, p. 472-482, Toronto, Ontario, Canadá, out. 1993. IBM Press.
- [185] D. Hoffman e P. Strooper. Classbench: a framework for automated class testing. *Software Practice and Experience*, 27(5):573-597, mai.1997.

- [186] W. E. Howden. Methodology for the generation of program test data. *IEEE Computer*, 24(5):554-559, mai.1975.
- [187] W. E. Howden. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, 2(3):208-215, set. 1976.
- [188] W. E. Howden. Symbolic testing and DISSECT symbolic evaluation system. *IEEE Transactions on Software Engineering*, 3(4):266-278, jul. 1977.
- [189] W. E. Howden. Theoretical and empirical studies of program testing. *IEEE Transactions on Software Engineering*, 4(4):293-298, jul. 1978.
- [190] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, 8(4):371-379, jul. 1982.
- [191] W. E. Howden. *Functional Program Testing and Analysis*. McGraw-Hill, Nova York, NY, EUA, 1987.
- [192] W. E. Howden. *Software Engineering and Technology: Functional Program Testing and Analysis*. Software Engineering and Technology. McGraw-Hill Book Co, Nova York, NY, EUA, 1987.
- [193] A. Hudson. Program errors as a birth and death process. Technical Report SP-3011, Systems Development Corporation, Santa Monica, CA, EUA, dez. 1967.
- [194] K. Hwang e F. A. Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill, Nova York, NY, EUA, 1990.
- [195] IBM. Rational PurifyPlus. Página WEB, 2005. Disponível em: <http://www-306.ibm.com/software/awdtools/purifyplus/>.
- [196] IEEE. IEEE standard glossary of software engineering terminology. Standard 610.12-1990 (R2002), IEEE Computer Society Press, 2002.
- [197] W. Isberg. Get test-inoculated! Artigo On-Line, abr. 2002. Disponível em: <http://www.ddj.com/dept/architect/184414846>.
- [198] D. Jackson e M. Woodward. Parallel firm mutation of Java programs. In: *Mutation Testing for the New Century*, p. 55-61, Norwell, MA, EUA, out. 2000. Kluwer Academic Publishers.
- [199] Z. Jelinski e P. B. Moranda. Software reliability research. In: *Statistical Computer Performance Evaluation*, p. 465-484, Nova York, NY, EUA, jun. 1972. Academic Press.
- [200] Z. Jin e A. J. Offut. Integration testing based on software couplings. In: *X Annual Conference on Computer Assurance (COMPASS 95)*, p. 13-23, Gaithersburg, MD, EUA, jan. 1995. IEEE Computer Society Press.
- [201] B. F. Jones, H. H. Sthamer e D. E. Eyres. Automatic structural testing using genetic algorithms. *The Software Engineering Journal*, 11(5):299-306, set. 1996.

- [202] J. A. Jones e M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In: *ASE '05: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, p. 273-282, Nova York, NY, USA, 2005. ACM Press.
- [203] R. F. Jorge. Teste de mutação: Subsídios para a redução do custo de aplicação. Dissertação de mestrado, ICMC/USP, São Carlos, SP, Brasil, mar. 2002.
- [204] P. C. Jorgensen e C. Erickson. Object oriented integration testing. *Communications of the ACM*, 37(9):30-38, set. 1994.
- [205] M. Kamkar. An overview and comparative classification of program slicing techniques. *Journal of Systems and Software*, 31(3):197-214, dez. 1995.
- [206] M. Kamkar. Application of program slicing in algorithmic debugging. *Information and Software Technology*, 40(11):637-645, dez. 1998.
- [207] Erik Kamsties e Christopher M. Lott. An empirical evaluation of three defect-detection techniques. In: *V European Software Engineering Conference*, p. 362-383, Londres, UK, set. 1995. Springer-Verlag.
- [208] H. Katz. *XQuery from the Experts:A Guide to the W3C XML Query Language*. Addison-Wesley, 1. ed., set. 2003.
- [209] G. Kiczales. Aspect-oriented programming. *ACM Computing Surveys*, 28(4es):154, dez. 1996.
- [210] J. Kienzle e R. Guerraoui. Aop: Does it make sense? the case of concurrency and failures. In: *XVI European Conference on Object-Oriented Programming - ECOOP'02*, volume 2374 de *Lecture Notes In Computer Science*, p. 37-61, Londres, UK, jun. 2002. Springer-Verlag.
- [211] J. Kienzle, Y. Yu e J. Xiong. On composition and reuse of aspects. In: *II Foundations of Aspect-Oriented Languages – FOAL'2003*, p. 17-24, mar. 2003.
- [212] S. Kim, J. A. Clark e J. A. McDermid. Assessing test set adequacy for object-oriented programs using class mutation. In: *Symposium on Software Technology – SoST'99*, p. 72-83, 1999.
- [213] S. Kim, J. A. Clark e J. A. Mcdermid. The rigorous generation of Java mutation operators using HAZOP. In: *12th International Conference on Software & Systems Engineering and their Applications (ICSSEA'99)*, dez. 1999.
- [214] S. Kim, J. A. Clark e J. A. McDermid. Class mutation: Mutation testing for object-oriented programs. In: *Object-Oriented Software Systems – OOSS*, 2000.
- [215] J. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385-394, jul. 1976.
- [216] E. Kit. *Software testing in the real world: improving the process*. Addison-Wesley, dez. 1995.
- [217] M. Kline. Software & hardware r&m: what are the differences? In: *The Annual Reliability & Maintainability Symposium*, p. 179-185, São Francisco, CA, EUA, jan. 1980.

- [218] A. Ko e B. Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In: *The 2004 Conference on Human factors in Computing Systems (SIGCHI)*, p. 151-158, Nova York, NY, EUA, abr. 2004. ACM Press.
- [219] B. Korel. PELAS – program error-locating assistant system. *IEEE Transactions on Software Engineering*, 14(9):1.253-1.260, set. 1988.
- [220] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870-879, ago. 1990.
- [221] B. Korel e J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155-163, 1988.
- [222] B. Korel e J. W. Laski. A tool for data flow oriented program testing. In: *II conference on Software development tools, techniques, and alternatives*, p. 34-37, Los Alamitos, CA, EUA, dez. 1985. IEEE Computer Society Press.
- [223] B. Korel e J. Rilling. Application of dynamic slicing in program debugging. In: *III Workshop on Automated and Algorithmic Debugging*, p. 43-58, Linköping, Suécia, mai.1997. Linköping Electronic Articles in Computer and Information Science.
- [224] B. Korel e J. Rilling. Program slicing in understanding of large programs. In: *VI International Workshop on Program Comprehension - IWPC'98*, p. 145-152, Washington, DC, EUA, jun. 1998. IEEE Computer Society.
- [225] B. Korel e S. Yalamanchili. Forward computation of dynamic program slices. In: *1994 ACM SIGSOFT international symposium on Software testing and analysis – ISSTA'94*, p. 66-79, Nova York, NY, EUA, ago. 1994. ACM Press.
- [226] E. W. Krauser, A. P. Mathur e V. J. Rego. High performance software testing on simd machines. *IEEE Transactions on Software Engineering*, 17(5):403-422, mai.1991.
- [227] H. Krawczyk e B. Wiszniewski. Classification of software defects in parallel programs. HPCTI Progress Report 1, Faculty of Electronics, Technical University of Gdańsk, Polônia, abr. 1995.
- [228] D. Kung, Y. Lu, N. Venugopalan, P. Hsia, Y. Toyoshima, C. Chen e J. Gao. Object state testing and fault analysis for reliable software systems. In: *VII International Symposium on Software Reliability Engineering – ISSRE'96*, p. 76-85, Washington, DC, EUA, out./nov. 1996. IEEE Computer Society.
- [229] D. C. Kung, J. Gao, P. Hsia, J. Lin e Y.Toyoshima. Class firewall, test order and regression testing of object-oriented programs. *Journal of Object-Oriented Program*, 8(2):51-65, mai.1995.
- [230] D. C. Kung, C.-H. Liu e P. Hsia. An object-oriented web test model for testing web applications. In: *XXIV International Computer Software and Applications Conference – COMPSAC'00*, p. 537-542, Washington, DC, EUA, out. 2000. IEEE Computer Society.
- [231] J. W. Laski e B. Korel. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, 9(3):347-354, mai.1983.

- [232] Y. Le Traon, T. Jéron, J.-M. Jézéquel e P. Morel. Efficient OO integration and regression testing. *IEEE Transactions on Reliability*, 49(1):12-25, mar. 2000.
- [233] S. C. L. Lee e J. Offutt. Generating test cases for xml-based web component interactions using mutation analysis. In: *XII International Symposium on Software Reliability Engineering – ISSRE’01*, p. 200, Washington, DC, EUA, nov. 2001. IEEE Computer Society.
- [234] P. S. J. Leitão. Suporte ao teste estrutural de programas cobol no ambiente poke-tool. Dissertação de mestrado, DCA/FEEC/UNICAMP, Campinas, SP, Brasil, ago. 1992.
- [235] O. A. L. Lemos. Teste de programas orientados a aspectos: uma abordagem estrutural para AspectJ. Dissertação de mestrado, ICMC/USP, São Carlos, SP, Brasil, fev. 2005. Disponível em: <http://www.teses.usp.br/teses/disponiveis/55/55134/tde-13042005-111234/>.
- [236] O. A. L. Lemos, F. C. Ferrari, P. C. Masiero e C. V. Lopes. Testing aspect-oriented programming pointcut descriptors. In: *WTAOP ’06: Proceedings of the 2<sup>nd</sup> workshop on Testing aspect-oriented programs*, p. 33-38, Nova York, NY, EUA, 2006. ACM Press.
- [237] O. A. L. Lemos, A. M. R. Vincenzi, J. C. Maldonado e P. C. Masiero. Teste de unidade de programas orientados a aspectos. In: *XVIII Simpósio Brasileiro de Engenharia de Software – SBES’04*, p. 55-70, Brasília, DF, Brasil, out. 2004.
- [238] R. Lencevicius, U. Hölzle e A. K. Singh. Query-based debugging of object-oriented programs. In: *XII ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications – OOPSLA’97*, p. 304-317, Nova York, NY, EUA, out. 1997. ACM Press.
- [239] L. Lian, S. Kusumoto, T. Kikuno, K. Matsumoto e K. Torii. A new fault localizing method for the program debugging process. *Information and Software Technology*, 39(4):271-284, abr. 1997.
- [240] H. Lieberman. The debugging scandal and what to do about it. *Communications of the ACM*, 40(4):26-29, abr. 1997.
- [241] H. Lieberman e C. Fry. *Software Visualization*, chapter ZStep 95: A Reversible, Animated Source Code Stepper, p. 277-292. MIT Press, 1998.
- [242] R. C. Linger, B. I. Witt e H. D. Mills. *Structured Programming: Theory and Practice the Systems Programming Series*. Addison-Wesley, mar. 1979.
- [243] S. Linkman, A. M. R. Vincenzi e J. Maldonado. An evaluation of systematic functional testing using mutation testing. In: *7th International Conference on Empirical Assessment in Software Engineering – EASE*, Keele, UK, abr. 2003. The IEE.
- [244] U. Linnenkugel e M. Müllerburg. Test data selection criteria for (software) integration testing. In: *I International Conference on Systems Integration – ICSI’90*, p. 709-717, Morristown, NJ, abr. 1990. IEEE Computer Society.
- [245] B. Littlewood e J. L. Verral. A bayesian reliability growth model for computer software. *Applied Statistics*, 22(3):332-346, 1973.

- [246] C. Liu e D.J. Richardson. Software components with retrospects. In: *International Workshop on the Role of Software Architecture in Testing and Analysis*, Marsala, Sicília, Itália, jul. 1998.
- [247] C. H. Liu, D. C. Kung e P. Hsia. Object-based data flow testing of web applications. In: *I Asia-Pacific Conference on Quality Software*, p. 7-16. IEEE Press, out. 2000.
- [248] C.H. Liu, D.C. Kung, P. Hsia e C.T. Hsu. Structural testing of web applications. In: *XI International Symposium on Software Reliability Engineering*, p. 84-96. IEEE Press, out. 2000.
- [249] B. Long, D. Hoffman e P. Strooper. Tool support for testing concurrent java components. *IEEE Transactions on Software Engineering*, 29(6):555-566, jun. 2003.
- [250] W. S. Lopes. Estelle: uma técnica para a descrição formal de serviços e protocolos de comunicação. *Revista Brasileira de Computação*, 5(1):33-44, set. 1989.
- [251] J. Lourenco, J. C. Cunha, H. Krawczyk, P. Kuzora, M. Neyman, and B. Wiszniewski. An integrated testing and debugging environment for parallel and distributed programs. In: *XXIII EUROMICRO Conference'97 New Frontiers of Information Technology*, p. 291-298. IEEE Press, set. 2001.
- [252] G. A. Di Lucca, A. R. Fasolino e P. Tramontana. Reverse engineering web applications: the ware approach. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(1-2):71-101, jan./abr. 2004.
- [253] G. Di Lucca, A. Fasolino e F. Faralli. Testing web applications. In: *ICSM '02: Proceedings of the International Conference on Software Maintenance (ICSM'02)*, p. 310-319, Washington, DC, EUA, out. 2002. IEEE Computer Society.
- [254] J. R. Lyle e M. Weiser. Automatic program bug location by program slicing. In: *II International Conference on Computers and Applications*, p. 877-883, Beijing, China, jun. 1987.
- [255] M. R. Lyu. *Handbook of Software Reliability Engineering*. McGraw-Hill, 1996.
- [256] Y. Ma, A. J. Offutt e Y. Kwon. MuJava: An automated class mutation system. *The Journal of Software Testing, Verification, and Reliability*, 15(2):97-133, 2005.
- [257] Y.-S. Ma, Y.-R. Kwon e J. Offutt. Inter-class mutation operators for Java. In: *XIII International Symposium on Software Reliability Engineering- ISSRE'2002*, p. 352-366, Washington, DC, EUA, nov. 2002. IEEE Computer Society.
- [258] N. Li Y. K. Malaiya. On input profile selection for software testing. In: *V International Symposium on Software Reliability Engineering – ISSRE'94*, p. 196-205, nov. 1994.
- [259] Y. K. Malaiya, N. Li, J. Bieman, R. Karcick e B. Skibe. The relationship between test coverage and reliability. In: *V International Symposium on Software Reliability Engineering – ISSRE'94*, p. 186-195, Monterey, CA, nov. 1994.
- [260] Y. K. Malaiya, M. L. Naixin, J. M. Bieman e R. Karcich. Software reliability growth with test coverage. *IEEE Transactions on Reliability*, 51(4):420-426, dez. 2002.

- [261] J. C. Maldonado. *Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software*. Tese de doutoramento, DCA/FEEC/UNICAMP, Campinas, SP, Brasil, jul. 1991.
- [262] J. C. Maldonado, E. F. Barbosa, A. M. R. Vincenzi e M. E. Delamaro. Evaluation N-selective mutation for C programs: Unit and integration testing. In: *Mutation testing for the new century*, p. 22-33, Norwell, MA, EUA, out. 2000. Kluwer Academic Publishers.
- [263] J. C. Maldonado, M. L. Chaim e M. Jino. Arquitetura de uma ferramenta de teste de apoio aos critérios potenciais usos. In: *XXII Congresso Nacional de Informática*, São Paulo, SP, Brasil, set. 1989.
- [264] J. C. Maldonado, S. R. Vergílio, M. L. Chaim e M. Jino. Critérios potenciais usos: Análise da aplicação de um benchmark. In: *VI Simpósio Brasileiro de Engenharia de Software – SBES'92*, p. 357-374, Gramado, RS, Brasil, nov. 1992.
- [265] J. C. Maldonado, A. M. Vincenzi, E. F. Barbosa, S. R. S. Souza e M. E. Delamaro. Aspectos teóricos e empíricos de teste de cobertura de software. In: *VI Escola de Informática da Sociedade Brasileira de Computação (SBC) – Regional Sul*, mai.1998.
- [266] N. Malevris, D. F. Yates e A. Veevers. Predictive metric for likely feasibility of program paths. *Journal of Electronic Materials*, 19(6):115-118, jun. 1990.
- [267] Man Machine Systems. Java code coverage analyzer – JCover. Página WEB, 2002. Disponível em: <http://www.mmsindia.com/JCover.html>.
- [268] A. C. Marshall, D. Hedley, I. J. Riddell e M. A. Hennell. Static dataflow-aided weak mutation analysis (sdawm). *Information and Software Technology*, 32(1):99-104, jan./fev. 1990.
- [269] E. Martins e C. M. Toyota. Construção de classes autotestáveis. In: *VIII Simpósio de Computação Tolerante a Falhas – SCTF'99*, p. 196-209, Campinas, SP, Brasil, jul. 1999.
- [270] V. Massol. Projeto PatternTesting. Página WEB, 2002. Disponível em: <http://patterntesting.sourceforge.net/>.
- [271] V. Matena e B. Stearns. *Applying Enterprise JavaBeans: Component-Based Development for the J2EE Platform*. Addison-Wesley, 2. ed., 2001.
- [272] A. P. Mathur. Performance, effectiveness and reliability issues in software testing. In: *15th Annual International Computer Software and Applications Conference*, p. 604-605, Tóquio, Japão, set. 1991. IEEE Computer Society Press.
- [273] A. P. Mathur e E. W. Krauser. Modeling mutation on a vector processor. In: *X International Conference on Software Engineering – ICSE'88*, p. 154-161, Los Alamitos, CA, EUA, abr. 1988. IEEE Computer Society Press.
- [274] A. P. Mathur e W. E. Wong. Evaluation of the cost of alternative mutation strategies. In: *VII Simpósio Brasileiro de Engenharia de Software – SBES'93*, p. 320-335, Rio de Janeiro, RJ, Brasil, out. 1993.

- [275] A. P. Mathur e W. E. Wong. An empirical comparison of data flow and mutation based test adequacy criteria. *Software Testing, Verification and Reliability*, 4(1):9-31, mar. 1994.
- [276] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(6):308-320, dez. 1976.
- [277] R. McDaniel e J. D. McGregor. Testing polymorphic interactions between classes. Technical Report TR-94-103, Clemson University, mar. 1994.
- [278] J. D. McGregor. Functional testing of classes. In: *Proc. 7th International Quality Week*, São Francisco, CA, mai. 1994. Software Research Institute.
- [279] J. D. McGregor e D. A. Sykes. *A Practical Guide to Testing Object-Oriented Software*. Addison-Wesley, 2001.
- [280] P. Mellor. Software reliability modelling: the state of the art. *Information and Software Technology*, 29(2):81-98, mar./abr 1987.
- [281] B. Meyer. Applying design by contract. *Computer*, 25(10):40-51, out. 1992.
- [282] C. C. Michael, G. McGraw e M. A. Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27(12):1.085-1.110, dez. 2001.
- [283] Microsoft Corporation. COM: Delivering on the promises of component technology. Página WEB, 2002. Disponível em: [http://www.computer-society.com/Redirect.php?id\\_url=532](http://www.computer-society.com/Redirect.php?id_url=532).
- [284] H. D. Mills. On the statistical validation of computer programs. Relatório Técnico FSC-72-6015, IBM Federal Systems Division, Gaithersburg, MD, 1972.
- [285] S. Monk e S. Hall. Virtual mock objects using AspectJ with JUNIT. Artigo On-Line, out. 2002. XProgramming.com. Disponível em: <http://xprogramming.com/xpmag/virtualMockObjects.htm>.
- [286] A. Mood, F. Graybill e D. Boes. *Introduction to the Theory of Statistics*. Probability and Statistics. McGraw-Hill, 3. ed., abr. 1974.
- [287] S. Moore, D. Cronk, K. S. London e J. Dongarra. Review of performance analysis tools for mpi parallel programs. In: *VIII European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 2131 of *Lecture Notes In Computer Science*, p. 241-248, Londres, UK, set. 2001. Springer-Verlag.
- [288] P. B. Moranda. Predictions of software reliability during debugging. In: *Annual Reliability and Maintainability Symposium*, p. 327-332, Washington, DC, EUA, 1975.
- [289] P. L. Moranda e Z. Jelinski. Final report on software reliability study. Technical Report 63921, McDonnell Douglas Astronautics Company, 1972.
- [290] L. J. Morell. A theory of fault-based testing. *IEEE Transactions on Software Engineering*, 16(8):844-857, ago. 1990.
- [291] G. C. Murphy, P. Townsend e P. S. Wong. Experiences with cluster and class testing. *Communications of the ACM*, 37(9):39-47, set. 1994.

- [292] J. D. Musa, A. Ianino e K. Okumoto. *Software Reliability – Measurement, Prediction, Application*. Software Engineering. McGraw-Hill, dez. 1987.
- [293] J. D. Musa e K. Okumoto. A logarithmic poisson execution time model for software reliability measurement. In: *VII International conference on Software engineering - ICSE'84*, p. 230-238, Piscataway, NJ, EUA, mar. 1984. IEEE Press.
- [294] G. J. Myers, C. Sandler, T. Badgett e T. M. Thomas. *The Art of Software Testing*. John Wiley & Sons, Nova York, NY, EUA, 2. ed., 2004.
- [295] G. J. Myers. A controlled experiment in program testing and code walkthroughs/inspections. *Communications of the ACM*, 21(9):760-768, set. 1978.
- [296] K. K. Nakazato, J. C. Maldonado, S. C. P. F. Fabbri e P. C. Masiero. Aspectos teóricos e de implementação de critérios de geração de sequências de teste baseados em máquinas de estado finito. Relatório Técnico 25, ICMC/USP, São Carlos, SP, Brasil, out. 1994.
- [297] P. Nardi, E. S. Spoto, M. E. Delamaro e A. M. R. Vincenzi. JaBUTi/BD: Utilização de critérios estruturais em aplicações de bancos de dados Java. In: *XIX Simpósio Brasileiro de Engenharia de Software – SBES'05*, p. 45-50, Uberlândia, MG, Brasil, out. 2005.
- [298] P. O. A. Navaux. Introdução ao processamento paralelo. *RBC – Revista Brasileira de Computação*, 5(2):31-43, out. 1989.
- [299] F. Nelson. Estimating software reliability from test data. *Microelectronics and Reliability*, 17:67-73, 1978.
- [300] A. Nishimatsu, M. Jihira, S. Kusumoto e K. Inoue. Call-mark slicing: an efficient and economical way of reducing slice. In: *XXI International Conference on Software Engineering – ICSE'99*, p. 422-431, Los Alamitos, CA, EUA, mai. 1999. IEEE Computer Society Press.
- [301] S. C. Ntafos. On required element testing. *IEEE Transactions on Software Engineering*, 10(6):795-803, nov. 1984.
- [302] S. C. Ntafos. A comparison of some structural testing strategies. *IEEE Transactions on Software Engineering*, 14(6):868-874, jul. 1988.
- [303] Object Management Group. Unified Modeling Language (UML), versão 2.0. Página WEB, mai.2004. Disponível em: <http://www.omg.org/technology/documents/formal/uml.htm>.
- [304] A. J. Offutt. Quality attributes of web software applications. *IEEE Software*, 19(2):25-32, mar. 2002.
- [305] A. J. Offutt e A. Abdurazik. Generating tests from UML specifications. In: *II International Conference on The Unified Modeling Language – UML'99*, volume 1723 of *Lecture Notes in Computer Science*, p. 416-429, Fort Collins, CO, EUA, out. 1999. Springer Berlin/Heidelberg.
- [306] A. J. Offutt e W. M. Craft. Using compiler optimization techniques to detect equivalent mutants. *Software Testing, Verification and Reliability*, 4(3):131-154, 1994.

- [307] A. J. Offutt e J. H. Hayes. A semantic model of program faults. In: *1996 ACM SIGSOFT International Symposium on Software Testing and Analysis – ISSTA’96*, p. 195-200, Nova York, NY, EUA, jan. 1996. ACM Press.
- [308] A. J. Offutt e A. Irvine. Testing object-oriented software using the category-partition method. In: *XVII International Conference on Technology of Object-Oriented Languages and Systems*, p. 293-304, Santa Barbara, CA, EUA, ago. 1995. Prentice-Hall.
- [309] A. J. Offutt e K. N. King. A Fortran 77 interpreter for mutation analysis. In: *Papers of the Symposium on Interpreters and interpretive techniques – SIGPLAN’87*, p. 177-188, Nova York, NY, EUA, jun. 1987. ACM Press.
- [310] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch e C. Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering Methodology*, 5(2):99-118, abr. 1996.
- [311] A. J. Offutt e A. J. Pan. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification and Reliability*, 7(3):165-192, 1997.
- [312] A. J. Offutt, J. Pan, K. Tewary e T. Zhang. An experimental evaluation of data flow and mutation testing. *Software Practice and Experience*, 26(2):165-176, fev. 1996.
- [313] A. J. Offutt, G. Rothermel e C. Zapf. An experimental evaluation of selective mutation. In: *XV International conference on Software Engineering - ICSE’93*, p. 100-107, Los Alamitos, CA, EUA, mai.1993. IEEE Computer Society Press.
- [314] A. J. Offutt, J. M. Voas e J. Payne. Mutation operators for ADA. Technical Report ISSE-TR-96-09, George Mason University, Fairfax, VA, EUA, mar. 1996.
- [315] A. J. Offutt e W. Xu. Generating test cases for web services using data perturbation. *ACM SIGSOFT Software Engineering Notes*, 29(5):1-10, set. 2004.
- [316] A. J. Offutt, R. Alexander, Y. Wu, Q. Xiao e C. Hutchinson. A fault model for subtype inheritance and polymorphism. In: *12th International Symposium on Software Reliability Engineering – ISSRE’01*, p. 84-93, Hong Kong, China, nov. 2001. IEEE Computer Society Press.
- [317] A. Orso, M. J. Harrold, D. Rosenblum, G. Rothermel, H. Do e M. L. Soffa. Using component metacontent to support the regression testing of component-based software. In: *IEEE International Conference on Software Maintenance - ICSM’01*, p. 716, Washington, DC, EUA, nov. 2001. IEEE Computer Society.
- [318] A. Orso, M. J. Harrold e D. S. Rosenblum. Component metadata for software engineering tasks. In: *Revised Papers from the II International Workshop on Engineering Distributed Objects – EDO’00*, volume 1999 of *Lecture Notes in Computer Science*, p. 129-144, Londres, UK, 2001. Springer-Verlag.
- [319] T. J. Ostrand e M. J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676-686, jun. 1988.
- [320] T. J. Ostrand e E. J. Weyuker. Using data flow analysis for regression testing. In: *VI Annual Pacific Northwest Software Quality Conference*, p. 233-247, Portland, OR, EUA, set. 1988.

- [321] H. Pan. *Software Debugging with Dynamic Instrumentation and Test-Based Knowledge*. Tese de doutoramento, Purdue University, West Lafayette, IN, EUA, ago. 1993.
- [322] H. Pan e E. H. Spafford. Toward automatic localization of software faults. In: *X Pacific Northwest Software Quality Conference*, p. 192-209, Portland, OR, EUA, out. 1992.
- [323] PARASOFT Corporation. C++ Test. Página WEB, 2000. Disponível em: <http://www.parasoft.com/>.
- [324] PARASOFT Corporation. Insure++. Página WEB, 2000. Disponível em: <http://www.parasoft.com/>.
- [325] R. P. Pargas, M. J. Harrold e R. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification and Reliability*, 9(4):263-282, 1999.
- [326] A. Pasquini, A. Crespo e P. Matrella. Sensitivity of reliability-growth models to operational profiles errors vs testing accuracy. *IEEE Transactions on Reliability*, 45(4):531-540, dez. 1996.
- [327] D. E. Perry e G. E. Kaiser. Adequate testing and object-oriented programming. *Journal on Object-Oriented Programming*, 2(5):13-19, jan./fev. 1990.
- [328] H. Pham e X. Zhang. Nhpp software reliability and cost models with testing coverage. *European Journal of Operational Research*, 145:443-454, 2003.
- [329] R. S. Pressman. *Engenharia de Software*. McGraw-Hill, 6. ed., 2006.
- [330] A. M. Price e A. Zorzo. Visualizando o fluxo de controle de programas. In: *IV Simpósio Brasileiro de Engenharia de Software – SBES'90*, Águas de São Pedro, SP, Brasil, out. 1990.
- [331] R. L. Probert e F. Guo. Mutation testing of protocols: Principles and preliminary experimental results. In: *IFIP TC6 – Third International Workshop on Protocol Test Systems*, p. 57-76. North-Holland, 1991.
- [332] Productivity through Sofware plc. ProLint Advanced Graphical Lint for C e C++. Página WEB, 2000. Disponível em: <http://www.pts.com/flxlint.cfm>.
- [333] Quest Software. JProbe Suite. Página WEB, 2003. Disponível em: <http://www.quest.com/jprobe/>.
- [334] C. V. Ramamoorthy e F. B. Bastani. Software reliability – status and perspectives. *IEEE Transactions on Software Engineering*, 8(4):354-371, jul. 1982.
- [335] C. V. Ramamoorthy, S. F. Ho e W. T. Chen. On automated generation of program test data. *IEEE Transactions on Software Engineering*, 2(4):293-300, dez. 1976.
- [336] S. Rapps e E. J. Weyuker. Data flow analysis techniques for test data selection. In: *VI International Conference on Software Engineering*, p. 272-278, Tóquio, Japão, set. 1982, IEEE Computer Society Press.
- [337] S. Rapps e E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367-375, abr. 1985.

- [338] A. Rashid e R. Chitchyan. Persistence as an aspect. In: *II International Conference on Aspect-Oriented Software Development – AOSD’03*, p. 120-129, Nova York, NY, EUA, mar. 2003. ACM Press.
- [339] RATIONAL Software Corporation. PureCoverage. Página WEB, 2000. Disponível em: <http://www-306.ibm.com/software/awdtools/purifyplus/>.
- [340] RATIONAL Software Corporation. Purify. Página WEB, 2000. Disponível em: <http://www-306.ibm.com/software/awdtools/purifyplus/>.
- [341] S. Reisman. Management and integrated tools. *IEEE Software*, 7(3):71-77, mai.1990.
- [342] Software Research. User’s guide – TCAT for Java/Windows – version 3.2. Página WEB, 2006. Disponível em: <http://www.soft.com/>.
- [343] F. Ricca e P. Tonella. Analysis and testing of web applications. In: *XXIII International Conference on Software Engineering - ICSE’01*, p. 25-34, Washington, DC, EUA, mai.2001. IEEE Computer Society.
- [344] M. Roper. *Software Testing*. McGraw-Hill Book Company Europe, 1994.
- [345] M. Roper, I. Maclean, A. Brooks, J. Miller e M. Wood. Genetic algorithms and the automatic generation of test data. Relatório Técnico RR/95/195, University of Strathclyde, Glasgow, UK, 1995.
- [346] A. C. A. Rosa e E. Martins. Using a reflexive architecture to validate object-oriented applications by fault injection. In: *1st Workshop on Reflexive Programming in C++ and Java*, p. 76-80, Vancouver, Canadá, 1998. Disponível em: <http://www.ic.unicamp.br/~eliane/>.
- [347] D. S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19-31, jan. 1995.
- [348] D. S. Rosenblum. Adequate testing of component-based software. Technical Report UCI-ICS-97-34, University of California, Irvine, CA, EUA, ago. 1997.
- [349] G. Rothermel, M. J. Harrold e J. Dedhia. Regression test selection for C++ software. *Software Testing, Verification and Reliability*, 10(2):77-109, jun. 2000.
- [350] B. Rudner. Seeding/tagging estimations of errors: Models and estimates. Relatório Técnico RADC-TR-77-15, Rome Air Development Center, jan. 1977.
- [351] T. D. Sant’ana. Ambiente de simulação e teste de programas paralelos. Dissertação de mestrado, ICMC/USP, São Carlos, SP, Brasil, 2001.
- [352] G. J. Schick e R. W. Wolverton. An analysis of competing software reliability models. *IEEE Transactions on Software Engineering*, 4(2):104-120, mar. 1978.
- [353] E. Y. Shapiro. *Algorithmic Program DeBugging*. MIT Press, Cambridge, MA, EUA, 1983.
- [354] T. Shimomura. Critical slice-based fault localization for any type of error. *IEICE Transactions on Information and Systems*, E76-D(6):656-667, jun. 1993.

- [355] T. Shimomura, Y. Oki, T. Chikaraishi e T. Ohta. An algorithmic fault-locating method for procedural languages and its implementation FIND. In: *II Workshop on Automated and Algorithmic Debugging – AADEBUG’95*, p. 191-203, Saint-Malo, França, mai.1995.
- [356] M. L. Shooman. Probabilistic models for software reliability prediction. In: *Statistical Computer Performance Evaluation*, p. 485-502, Nova York, NY, EUA, 1972. Academic Press.
- [357] D. P. Sidhu e T.-K. Leung. Formal methods for protocol testing: A detailed study. *IEEE Transactions on Software Engineering*, 15(4):413-426, abr. 1989.
- [358] E. J. Silva e S. R. Vergilo. Predtool: Uma ferramenta para apoiar o teste baseado em predicados. In: *XXX Conferência Latinoamericana de Informática – CLEI’2004*, p. 117-128, Arica, Peru, set. 2004. Sociedad Peruana de Computación. Disponível em: <http://clei2004.spc.org.pe/es/html/pdfs/40.pdf>.
- [359] S. Silva-Barradas. *Mutation Analysis of Concurrent Software*. Tese de doutoramento, Dottorato di Ricerca in Ingegneria Informatica e Automatica, Politecnico di Milano, 1998.
- [360] A. S. Simão. Proteum-RS/PN: Uma ferramenta para a validação de redes de petri baseada na análise de mutantes. Dissertação de mestrado, ICMC/USP, São Carlos, SP, Brasil, fev. 2000. Disponível em: <http://www.teses.usp.br/teses/disponiveis/55/55134/tde-19022001-095632/>.
- [361] A. S. Simão, J. C. Maldonado e S. C. P. F. Fabbri. Proteum-RS/PN: A tool to support edition, simulation and validation of Petri nets based on mutation testing. In: *XIV Simpósio Brasileiro de Engenharia de Software – SBES’2000*, p. 227-242, João Pessoa, PB, Brasil, out. 2000.
- [362] A. S. Simão, A. M. R. Vincenzi, J. C. Maldonado e A. C. L. Santana. A language for the description of program instrumentation and the automatic generation of instruments. *CLEI Electronic Journal*, 6(1):23, 2003.
- [363] S. Sinha e M. J. Harrold. Analysis of programs with exception-handling constructs. In: *International Conference on Software Maintenance – ICSM’98*, p. 348-357, Washington, DC, EUA, nov. 1998. IEEE Computer Society.
- [364] S. Sinha e M. J. Harrold. Criteria for testing exception-handling constructs in java programs. In: *IEEE International Conference on Software Maintenance – ICSM’99*, p. 265-274, Washington, DC, EUA, ago./set. 1999. IEEE Computer Society.
- [365] M. D. Smith e D. J. Robson. Object-oriented programming – the problems of validation. In: *VI International Conference on Software Maintenance – ICSM’90*, p. 272-281, Washington, DC, EUA, nov. 1990. IEEE Computer Society.
- [366] M. Snir, S. Otto, H. Steven, D. Walker e J. J. Dongara. *MPI: The Complete Reference*, volume 1. The MIT Press, 2. ed., set. 1998.
- [367] C. Sommerhauser. SUN launches new suite of Java testing tools. Página WEB, 1997. Disponível em: <http://www.sun.com/smi/Press/sunflash/1997-08/sunflash.970811.1112.xml>.

- [368] N. Soundarajan e B. Tyler. Testing components. In: *Workshop on Specification and Verification of Component-Based Systems – OOPSLA’01*, p. 4-9, Tampa, FL, EUA, out. 2001. ACM Press.
- [369] A. L. Souter e L. L. Pollock. Omen: A strategy for testing object-oriented software. In: *IX International Symposium on Software Testing and Analysis - ISSTA’00*, p. 49-59, Nova York, NY, EUA, 2000. ACM Press.
- [370] A. L. Souter e L. L. Pollock. The construction of contextual def-use associations for object-oriented systems. *IEEE Transactions on Software Engineering*, 29(11):1.005-1.018, 2003.
- [371] A. L. Souter, L. L. Pollock e D. Hisley. Inter-class def-use analysis with partial class representations. In: *PASTE’99: Proceedings of the 1999 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, p. 47-56, Nova York, NY, EUA, 1999. ACM Press.
- [372] S. R. S. Souza. Avaliação do custo e eficácia do critério análise de mutantes na atividade de teste de software. Dissertação de mestrado, ICMC/USP, São Carlos, SP, Brasil, jun. 1996.
- [373] S. R. S. Souza e J. C. Maldonado. Avaliação do impacto da minimização de conjuntos de casos de teste no custo e eficácia do critério análise de mutantes. In: *XI Simpósio Brasileiro de Engenharia de Software – SBES’97*, Fortaleza, CE, Brasil, 1997.
- [374] S. R. S. Souza, J. C. Maldonado, S. C. P. F. Fabbri e W. Lopes de Souza. Mutation testing applied to estelle specifications. *Software Quality Journal*, 8(4):285-302, abr. 2000. Kluwer Academic Publishers.
- [375] S. R. S. Souza, J. C. Maldonado e S. R. Vergilio. Análise de mutantes e potenciais-usos: Uma avaliação empírica. In: *XIII Conferência Internacional de Tecnologia de Software – CITS’97*, p. 225-236, Curitiba, PR, Brasil, jun. 1997.
- [376] S. R. S. Souza, S. R. Vergilio, P. S. L. Souza, A. S. Simão, T. G. Bliscosque, A. M. Lima e A. C. Hausen. Valipar: A testing tool for message-passing parallel programs. In: *XVII International Conference on Software knowledge and Software Engineering – SEKE’05*, Taipei, Taiwan, China, jul. 2005.
- [377] A. V. Staa. *Programação Modular*. Editora Campus/Elsevier, Rio de Janeiro, RJ, Brasil, 2000.
- [378] A. V. Staa. *Qualidade de Software: Teoria e Prática*, capítulo Instrumentação, p. 226-237. Prentice-Hall, São Paulo, SP, Brasil, 2001.
- [379] R. M. Stallman, R. H. Pesch e S. Shebs. *Debugging with GDB: The GNU Source-Level Debugger*. Free Software Foundation, Cambridge, MA, EUA, 9. ed., fev. 2002.
- [380] T. Sugita, J. C. Maldonado e W. E. Wong. Mutation testing applied to validate SDL specifications. In: *16th IFIP International Conference on Testing of Communicating Systems – TestCom2004*, p. 193-208, Oxford, United Kingdom, mar. 2004. Springer.
- [381] C. Szyperski. *Component Software Beyond Object-Oriented Programming*. Addison-Wesley Professional, 2. ed., nov. 2002.

- [382] K.-C. Tai. Predicate-based test generation for computer programs. In: *XV International Conference on Software Engineering - ICSE'93*, p. 267-276, Los Alamitos, CA, EUA, mai.1993. IEEE Computer Society Press.
- [383] K.-C. Tai, R. H. Carver e E. E. Obaid. Debugging concurrent Ada programs by deterministic execution. *IEEE Transactions on Software Engineering*, 17(1):45-63, jan. 1991.
- [384] K.-C. Tai e F. J. Daniels. Interclass test order for object-oriented software. *Journal of Object-Oriented Programming*, 12(4):18-25, 1999.
- [385] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2. ed., 2001.
- [386] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146-160, 1972.
- [387] M. Tatsubori, S. Chiba, M.-O. Killijian e K. Itano. OpenJava: A Class-Based Macro System for Java. In: *Reflection and Software Engineering*, volume 1826 de *Lecture Notes in Computer Science*, p. 117-133, Heidelberg, Alemanha, jun. 2000. Springer-Verlag.
- [388] R. N. Taylor, D. L. Levine e C. Kelly. Structural testing of concurrent programs. *IEEE Transactions on Software Engineering*, 18(3):206-215, mar. 1992.
- [389] Telcordia Technologies. xSuds Toolsuite. Página WEB, 1998. Disponível em: <http://xsuds.argreenhouse.com/>.
- [390] K. Templer e C. Jeffery. A configurable automatic instrumentation tool for ansi c. In: *XIII IEEE international conference on Automated software engineering – ASE'98*, p. 249-258, Washington, DC, EUA, out. 1998. IEEE Computer Society.
- [391] The AspectJ Team. The AspectJ programming guide, fev. 2003. Xerox Corporation. Disponível em: <http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/aspectj-home/doc/progguide/index.html>.
- [392] H. S. Thompson, D. Beech, M. Maloney e N. Mendelsohn. XML Schema part 1: Structures second edition. Página WEB, out. 2004. W3C – World Wide Web Consortium. Disponível em: <http://www.w3.org/TR/xmlschema-1/>.
- [393] J. Tian, P. Lu e J. Palma. Test-execution-based reliability measurement and modeling for large commercial software. *IEEE Transactions on Software Engineering*, 21(5):405-414, mai.1995.
- [394] Jeff Tian. Integrating time domain and input domain analyses of software reliability using tree-based models. *IEEE Transactions on Software Engineering*, 21(12):945-958, dez. 1995.
- [395] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121-189, 1995. Disponível em: <http://www.research.ibm.com/people/t/tip/papers/jpl1995.pdf>.
- [396] A. Tolmach e A. W. Appel. A debugger for standard ML. *Journal of Functional Programming*, 5(2):155-200, abr. 1995.

- [397] N. Tracey, J. Clark, K. Mander e J. McDermid. An automated framework for structural test-data generation. In: *XIII IEEE International Conference on Automated Software Engineering – ASE'98*, p. 285-288, Washington, DC, EUA, out. 1998. IEEE Computer Society.
- [398] Jan Tretmans. Testing concurrent systems: A formal approach. In: Jos C. M. Baeten and Sjouke Mauw, eds., *CONCUR '99: Concurrency Theory, 10th International Conference, Eindhoven, The Netherlands, ago. 24-27, 1999, Proceedings*, volume 1664 de *Lecture Notes in Computer Science*, p. 46-65. Springer, 1999.
- [399] C. D. Turner e D. J. Robson. The state-based testing of object-oriented programs. In: *Conference on Software Maintenance – CSM'93*, p. 302-310, Washington, DC, EUA, set. 1993. IEEE Computer Society.
- [400] H. Ural e B. Yang. A structural test selection criterion. *Information Processing Letters*, 28(3):157-163, jul. 1988.
- [401] B. Vanhaute, B. De Win e B. De Decker. Building frameworks in AspectJ. In: *Workshop on Advanced Separation of Concerns*, p. 1-6, jun. 2001.
- [402] G. S. Varadan. Trends in reliability and test strategies. *ieees*, 12(3):10, mai.1995.
- [403] A. Veevers e A. Marshall. A relationship between software coverage metrics and reliability. *Software Testing, Verification and Reliability*, 4(1):3-8, 1994.
- [404] S. R. Vergilio. *Critérios Restritos: Uma Contribuição para Aprimorar a Eficácia da Atividade de Teste de Software*. Tese de doutoramento, DCA/FEEC/UNICAMP, Campinas, SP, Brasil, jul. 1997.
- [405] S. R. Vergilio, J. C. Maldonado e M. Jino. Caminhos não-executáveis na automação das atividades de teste. In: *VI Simpósio Brasileiro de Engenharia de Software – SBES'92*, p. 343-356, Gramado, RS, Brasil, nov. 1992.
- [406] S. R. Vergilio, J. C. Maldonado e M. Jino. Constraint based criteria: An approach for test case selection in the structural testing. *Journal of Electronic Testing: Theory and Applications*, 17(2):175-183, abr. 2001.
- [407] S. R. Vergilio, J. C. Maldonado e M. Jino. Experimental results from application of fault-sensitive testing strategies. *Revista de Informática Teórica e Aplicada – RITA*, 12(1):61-82, jun. 2005.
- [408] S. R. Vergilio, J. C. Maldonado, M. Jino e I. W. Soares. Constraint based structural testing criteria. *Journal of Systems and Software*, 79(6):756-771, jun. 2006.
- [409] S. R. Vergilio, S. R. S. Souza e P. S. L. Souza. Coverage testing criteria for message-passing parallel programs. In: *Latin-American Test Workshop – LATW'05*, volume 1, p. 161-166, Salvador, BA, Brasil, mar./abr. 2005.
- [410] I. Vessey. Expertise in debugging computer programs: A process analysis. *International Journal on Man-Machine Studies*, 23(5):459-494, 1985.
- [411] P. R. S. Vilela. *Critérios Potenciais Usos de Integração: Definição e Análise*. Tese de doutoramento, DCA/FEEC/Unicamp, Campinas, SP, Brasil, abr. 1998.

- [412] P. R. S. Vilela, J. C. Maldonado e M. Jino. Program graph visualization. *Software Practice and Experience*, 27(11):1.245-1.262, nov. 1997.
- [413] A. M. R. Vincenzi. *Orientação a Objeto: Definição, Implementação e Análise de Recursos de Teste e Validação*. Tese de doutoramento, ICMC/USP, São Carlos, SP, Brasil, mai.2004. Disponível em: <http://www.teses.usp.br/teses/disponiveis/55/55134/tde-17082004-122037>.
- [414] A. M. R. Vincenzi, M. E. Delamaro, J. C. Maldonado e W. E. Wong. Java bytecode static analysis: Deriving structural testing requirements. In: *II UK Software Testing Workshop – UK-Softest'2003*, p. 21, Department of Computer Science, University of York, York, UK, set. 2003. University of York Press.
- [415] A. M. R. Vincenzi, M. E. Delamaro, J. C. Maldonado e W. E. Wong. Establishing structural testing criteria for Java bytecode. *Software Practice and Experience*, 36(14):1.513-1.541, nov. 2006.
- [416] A. M. R. Vincenzi, J. C. Maldonado, E. F. Barbosa e M. E. Delamaro. Interface sufficient operators: A case study. In: *XIII Simpósio Brasileiro de Engenharia de Software – SBES'99*, p. 373-391, Florianópolis, SC, Brasil, out. 1999.
- [417] A. M. R. Vincenzi, J. C. Maldonado, E. F. Barbosa e M. E. Delamaro. Operadores essenciais de interface: Um estudo de caso. In: *XIII Simpósio Brasileiro de Engenharia de Software – SBES'99*, p. 373-391, Florianópolis, SC, out. 1999.
- [418] A. M. R. Vincenzi, J. C. Maldonado, E. F. Barbosa e M. E. Delamaro. Unit and integration testing strategies for C programs using mutation-based criteria. In: *Symposium on Mutation Testing*, p. 45, San Jose, CA, out. 2000. Kluwer Academic Publishers. (Edição especial do *Software Testing Verification and Reliability Journal* 11(4), 2001).
- [419] A. M. R. Vincenzi, J. C. Maldonado, M. E. Delamaro, E. S. Spoto e W. E. Wong. *Component-Based Software Quality: Methods and Techniques*, volume 2.693 de *Lecture Notes in Computer Science*, chapter Component-Based Software: An Overview of Testing, p. 99-127. Springer-Verlag, Nova York, NY, EUA, jun. 2003. (A. Cechich, M. Piattini e A. Vallecillo eds.).
- [420] A. M. R. Vincenzi, J. C. Maldonado, W. E. Wong e M. E. Delamaro. Coverage testing of Java programs and components. *Science of Computer Programming*, 56(1-2):211-230, 2005.
- [421] A. M. R. Vincenzi, E. Y. Nakagawa, J. C. Maldonado, M. E. Delamaro e R. A. F. Romero. Bayesian-learning based guidelines to determine equivalent mutants. *International Journal of Software Engineering and Knowledge Engineering – IJSEKE*, 12(6):675-689, dez. 2002.
- [422] A. M. R. Vincenzi, W. E. Wong, M. E. Delamaro e J. C. Maldonado. JaBUTi: A coverage analysis tool for Java programs. In: *XVII Simpósio Brasileiro de Engenharia de Software – SBES'2003*, p. 79-84, Manaus, AM, Brasil, out. 2003.
- [423] C. Viravan. *Enhancing Debugging Technology*. Tese de doutoramento, Purdue University, West Lafayette, IN, EUA, mar. 1994.

- [424] Y. Wang, G. King e H. Wickburg. A method for built-in tests in component-based software maintenance. In: *III European Conference on Software Maintenance and Reengineering – CSMR'99*, p. 186-189, Washington, DC, EUA, mar. 1999. IEEE Computer Society.
- [425] J. Wegener, A. Baresel e H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841-854, dez. 2001.
- [426] R. Weichselbaum. Software test automation by means of genetic algorithms. In: *VI International Conference on Software Testing, Analysis and Review*, Munique, Alemanha, 1998.
- [427] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352-357, jul. 1984.
- [428] C. M. L. Werner e R. M. M. Braga. Desenvolvimento baseado em componentes. Minicurso apresentado no XIV Simpósio Brasileiro de Engenharia de Software – SBES'2000, out. 2000.
- [429] E. J. Weyuker. The complexity of data flow criteria for test data selection. *Information Processing Letters*, 19(2):103-109, ago. 1984.
- [430] E. J. Weyuker. The cost of data flow testing: an empirical study. *IEEE Transactions on Software Engineering*, 16(2):121-128, fev. 1990.
- [431] E. J. Weyuker e B. Jeng. Analyzing partition testing strategies. *IEEE Transactions on Software Engineering*, 17(7):703-711, jul. 1991.
- [432] J. Whaley e M. Rinard. Compositional pointer and escape analysis for Java programs. *ACM SIGPLAN Notices*, 34(10):187-206, 1999.
- [433] L. J. White e E. I. Cohen. A domain strategy for computer program testing. *IEEE Transactions on Software Engineering*, 6(3):247-257, mai.1980.
- [434] L. J. White e P. N. Sahay. A computer system for generating test data using the domain strategy. In: *II Conference on Software development tools, techniques, and alternatives*, p. 38-45, Los Alamitos, CA, EUA, 1985. IEEE Computer Society Press.
- [435] W. E. Wong. *On Mutation and Data Flow*. Tese de doutoramento, Department of Computer Science, Purdue University, West Lafayette, IN, EUA, dez. 1993.
- [436] W. E. Wong, J. R. Horgan, S. S. Gokhale e K. S. Trivedi. Locating program features using execution slices. In: *1999 IEEE Symposium on Application-Specific Systems and Software Engineering and Technology – ASSET'99*, p. 194-203, Washington, DC, EUA, 1999. IEEE Computer Society.
- [437] W. E. Wong, J. C. Maldonado, M. E. Delamaro e A. P. Mathur. Constrained mutation in C programs. In: *VIII Simpósio Brasileiro de Engenharia de Software – SBES'94*, p. 439-452, Curitiba, PR, Brasil, out. 1994.
- [438] W. E. Wong, J. C. Maldonado, M. E. Delamaro e S. R. S. Souza. A comparison of selective mutation in C and Fortran. In: *Workshop do Projeto Validação e Teste de Sistemas de Operação*, p. 71-80, Águas de Lindóia, SP, Brasil, jan. 1997.

- [439] W. E. Wong e A. P. Mathur. Reducing the cost of mutation testing: an empirical study. *Journal of Systems and Software*, 31(3):185-196, dez. 1995.
- [440] W. E. Wong, A. P. Mathur e J. C. Maldonado. Mutation versus all-uses: An empirical evaluation of cost, strength and effectiveness. In: *Software Quality and Productivity: Theory, practice and training*, p. 258-265, Londres, UK, UK, dez. 1995. Chapman & Hall, Ltd.
- [441] M. Wood, M. Roper, A. Brooks e J. Miller. Comparing and combining software defect detection techniques: a replicated empirical study. In: *VI European conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering – ESEC’97/FSE-5*, p. 262-277, Nova York, NY, EUA, set. 1997. Springer-Verlag Nova York, Inc.
- [442] M. R. Woodward. Mutation testing – its origin and evolution. *Information and Software Technology*, 35(3):163-169, mar. 1993.
- [443] M. R. Woodward e K. Halewood. From weak to strong, dead or alive? an analysis of some mutation testing issues. In: *II Workshop on Software Testing, Verification and Analysis*, p. 152-158, Banff, Canadá, jul. 1988.
- [444] M. R. Woodward, D. Heddley e M. A. Hennel. Experience with path analysis and testing of programs. *IEEE Transactions on Software Engineering*, 6(3):278-286, mai.1980.
- [445] Y. Wu e A. J. Offutt. Modeling and testing web-based applications. Relatório Técnico ISE-TR-02-08, George Mason University, Fairfax, VA, EUA, nov. 2002. Disponível em: [http://ise.gmu.edu/techrep/2002/02\\_08.pdf](http://ise.gmu.edu/techrep/2002/02_08.pdf).
- [446] S. Xanthakis, C. Ellis, C. Skourlas, A. LeGall e S. Katsikas. Application of genetic algorithms to software testing. In: *V IEEE International Conference on Software Engineering*, p. 625-636, Tolouse, França, dez. 1992.
- [447] M. Xie. Software reliability models – a selected annotated bibliography. *Software Testing, Verification and Reliability*, 3(1):3-28, 1993.
- [448] D. Xu e W. Xu. State-based incremental testing of aspect-oriented programs. In: *AOSD ’06: Proceedings of the 5th international conference on Aspect-oriented software development*, p. 180-189, Nova York, NY, EUA, 2006. ACM Press.
- [449] D. Xu, W. Xu e K. Nygard. A state-based approach to testing aspect-oriented programs. In: *XVII International Conference on Software Engineering and Knowledge Engineering – SEKE’2005*, p. 6, Taiwan, China, jul. 2005.
- [450] W. Xu e D. Xu. State-based testing of integration aspects. In: *WTAOP ’06: Proceedings of the 2nd workshop on Testing aspect-oriented programs*, p. 7-14, Nova York, NY, EUA, 2006. ACM Press.
- [451] C.-S. Yang, A. L. Souter e L. L. Pollock. All-du-path coverage for parallel programs. In: *1998 ACM SIGSOFT International Symposium on Software Testing and Analysis – ISSTA’98*, p. 153-162, Nova York, NY, EUA, jan. 1998. ACM Press.
- [452] C-S. D. Yang. *Program-Based, Structural Testing of Shared Memory Parallel Programs*. Tese de doutoramento, University of Delaware, 1999.

- [453] S. J. Zeil, F. H. Afifi e L. J. White. Detection of linear errors via domain testing. *ACM Transactions on Software Engineering Methodology*, 1(4):422-451, out. 1992.
- [454] A. Zeller. Isolating cause-effect chains from computer programs. *ACM SIGSOFT Software Engineering Notes*, 27(6):1-10, nov. 2002.
- [455] X. Zhang e R. Gupta. Cost effective dynamic program slicing. In: *ACM SIGPLAN 2004 conference on Programming language design and implementation – PLDI'04*, p. 94-106. ACM Press, jun. 2004.
- [456] Xiangyu Zhang, Rajiv Gupta e Youtao Zhang. Precise dynamic slicing algorithms. In: *XXV International Conference on Software Engineering – ICSE'03*, p. 319-329, Washington, DC, EUA, mai.2003. IEEE Computer Society.
- [457] J. Zhao. Dependence analysis of Java bytecode. In: *XXIV IEEE Annual International Computer Software and Applications Conference – COMPSAC'2000*, p. 486-491, Taipei, Taiwan, out. 2000. IEEE Computer Society Press.
- [458] Y. Zhou, D. Richardson e H. Ziv. Towards a practical approach to test aspect-oriented software. In: *Testing Component-based Systems – TECOS'2004*, Erfurt, Alemanha, set. 2004.
- [459] H. Zhu. A formal analysis of the subsume relation between software test adequacy criteria. *IEEE Transactions on Software Engineering*, 22(4):248-255, abr. 1996.

# Índice Remissivo

- Aceitador, 82, 83
- Adendo, 176
  - anterior, 180
  - de contorno, 180
  - posterior, 180
- Alcançabilidade, 284
- Algoritmo genético, 278, 279
- Análise
  - de fluxo de dados, 274, 276, 291
  - de Mutantes, 77, 84
  - dinâmica, 274
  - do valor limite, 14
- Aplicação *Web*, 209
- Aplicabilidade, 252
- Arco, 55
- AspectJ, 177
- Aspectos, 175, 176
  - de desenvolvimento, 177
  - de produção, 177
- Associação, 50, 57
- Borda, 280
- Caminho, 51, 55
  - ausente, 270, 280
  - completo, 51, 54
  - linearmente independente, 55
  - livre de definição, 57
  - livre de laço, 51, 54
  - não executável, 54, 271
  - simples, 51, 54, 57
- Caso de teste, 3
- CMS, 96
- Cobertura, 278
- Codificação, 293, 295
- Complexidade, 251, 253
  - ciclomática, 48, 55, 61
- Computação do caminho, 272
- Condição do caminho, 272, 273, 279
- Confiabilidade, 7, 315
  - de hardware, 317
  - de software, 316, 317, 353
- Conjunto
  - de caracterização, 36
  - de casos de teste, 3
  - de junção, 180
  - de teste, 3
- Correção coincidente, 269, 280
- Critério
  - de McCabe, 55, 62
  - de teste, 4
  - estrutural, 133
  - fluxo de comunicação, 242
  - fluxo de controle, 56, 192, 242, 278
  - fluxo de dados, 57, 192, 242, 249, 252, 253, 278
  - passagem de mensagem, 242
- Potenciais-Usos, 253, 254
- Todas-Arestas, 48, 55, 56, 62, 196
- Todas-Arestas-Transversais, 197
- Todas-Definições, 50, 55
- Todos-c-Usos, 55
- Todos-Caminhos, 48, 55, 56
- Todos-Du-Caminhos, 50
- Todos-Nós, 48, 55, 56, 62
- Todos-Nós-Transversais, 196
- Todos-p-Usos, 55
- Todos-Potenciais-Du-Caminhos, 50
- Todos-Potenciais-Usos, 50
- Todos-Potenciais-Usos/Du, 50
- Todos-Usos, 50, 62, 197
- Todos-Usos-Transversais, 197
- Dado de teste, 3
- Declaração intertipos, 180
- Defeito, 2
- Definição de variável, 49, 55
- DeMillo, 78

- Depuração, 6, 293  
algorítmica interprocedimental, 306  
algoritmica, 305  
algoritmica intraprocedimental, 307  
baseada em rastreamento e inspeção,  
298  
baseada em requisitos de teste, 305  
com asserções, 299  
delta, 310
- Depurador simbólico, 296, 298, 299
- Dice, 304
- Dicing, 304
- Distribuição  
de Weibull, 330  
exponencial, 327  
Gamma, 332
- Domínio  
de entrada, 2  
de Saída, 2
- DS, 32, 301
- DTD, 210
- Dynamic  
dice, 302  
slice, 302
- Efeito de acoplamento, 80, 84
- Efeitos da programação OO, 123
- Elemento requerido, 269, 271, 278
- Engano, 2
- Erro, 2  
de observabilidade, 236  
de travamento, 236
- Error Guessing, 24
- Escore de mutação, 89
- Especificação, 27
- Estratégia incremental, 141  
hierárquica, 146, 153
- Estudos  
experimentais, 251, 256  
teóricos, 251, 252
- Execução  
controlada, 245, 246  
dinâmica, 271, 274  
em reverso, 298, 299, 310  
simbólica, 272, 274
- Execution dice, 302
- EXPER, 96
- Falha, 2
- Fases  
de teste, 3  
de teste OO, 128
- Fatia  
crítica, 303  
de programa, 294  
dinâmica, 301, 303  
estática, 301
- Fatiamento  
de programas, 301  
dinâmico, 301, 302, 307, 312  
estático, 301, 312  
usando informação de teste, 303
- Ferramenta  
de teste, 169  
de teste OO, 174
- Fluxo  
de controle, 56  
de dados, 57
- FMS, 96
- Fragmento de recorte, 304  
de execução, 304  
dinâmico, 304  
estático, 304
- Função  
Confiabilidade, 318, 319  
de Falhas Acumuladas, 318, 323  
Intensidade de Falhas, 318, 325  
Taxa de Falhas, 318, 319
- Funções de confiabilidade, 327
- Geração  
aleatória, 257, 271, 274  
de dados de teste, 7  
de seqüências de teste, 37
- Gerador, 82, 83
- GFC, 54
- Grafo  
AODU, 192, 194  
Causa-Efeito, 18  
de Chamadas de Classe, 133  
de Fluxo de Controle, 54, 66, 192,  
237  
de Fluxo de Controle de Classe, 133  
de Fluxo de Controle Paralelo, 237,  
238  
de Instruções, 137, 138
- Def, 65
- Def-Uso, 63, 139, 192

- Def-uso, 57
- HAZOP, 85
- Heurística
- baseada em recorte, 305
  - baseadas em fatias, 303
- Hierarquia de critérios, 253
- Hipótese de programador competente, 82, 84
- HTML, 209, 210
- HTTP, 209
- Imitadores virtuais de objetos, 182
- Indefinição de variável, 52
- Instrumentação de código, 189
- Invariante, 187
- JaBUTi, 137
- JaBUTi/AJ, 198
- Leitura de código, 260, 262, 263, 265
- Lipton, 78
- Máquina de Estados Finitos, 28, 29, 78, 97, 114
- Método
- DS, 38
  - TT, 38
  - UIO, 38, 42
  - W, 38, 42
- Manutenção, 295, 296
- Medição de confiabilidade, 326, 357
- MEF, 28
- Message Passing Interface, 234
- Metamodelo, 216
- Modelo
- aspectual de estados, 200
  - baseado em cobertura de teste, 348
  - baseado no domínio de dados, 338
  - baseado no domínio do tempo, 340
  - de comportamento, 214
  - de confiabilidade, 333, 335, 336
  - de depuração depois do teste, 295
  - de implante de defeitos, 337
  - de Jelinski-Moranda, 345
  - de objetos, 211
  - de Weibull, 343
  - geométrico, 346
- Hipótese-Validação, 294, 295
- tipo Binomial, 349
- Mothra, 97
- MPI, 234
- Mutação
- aleatória, 91, 92, 258
  - de Interface, 108, 255, 259
  - restrita, 91, 92, 258
  - seletiva, 92
- Mutante
- equivalente, 89, 90, 271, 285
  - morto, 88, 105
  - vivo, 88, 105–107
- Nó, 53, 55
- Não-determinismo, 231, 244, 245
- Necessidade, 284
- Object Relation Diagram, 203
- Operadores de mutação, 84, 85, 155, 156, 158, 159
- essenciais, 259
- Oráculo, 3
- ORD, 203
- Ordenação
- de aspectos, 203
  - de classes, 203
- Página Web, 216
- Pós-condições, 187
- Padrões de código, 187
- Parallel Virtual Machine, 234
- Particionamento em classes de equivalência, 11
- POA, 175, 176
- POKE-TOOL, 68
- Ponto
- off*, 281
  - on*, 281
- Potencial-associação, 50
- Potencial-uso, 50
- Precondições, 187
- Programa concorrente, 231, 232
- Programação orientada a aspectos, 175
- Proteum, 98, 258, 259
- Proteum/IM, 113
- PVM, 234
- Qualidade de software, 315
- Recorte, 304

- Rede de Petri, 78, 97, 115  
Relação de inclusão, 251, 252, 254–256  
Revisão técnica, 2
- Sayward, 78  
Schema XML, 210, 225  
Script, 210  
SDAWM, 92  
Seqüência  
    Única de Entrada e Saída, 34  
    de distinção, 32, 38  
    de sincronização, 31  
    UIO, 34  
Serviço Web, 210, 222  
Sintomas internos, 297, 298  
Slice, 301  
Slicing, 294  
SS, 31, 301  
Static slice, 302  
Strength, 252, 258  
Suficiência, 285
- Técnica  
     $N \times 1$ , 281  
     $N \times N$ , 282  
     $V \times V$ , 282  
Técnica baseada em caminhos, 272  
Tempo Médio entre Falhas, 318  
Tempo Médio para Falhas, 318, 326  
Teste, 1  
    aleatório, 5  
    baseado em defeitos, 221  
    baseado em especificação, 27, 132  
    baseado em estados, 200  
    baseado em modelo de especificação, 216  
    baseado em predicados, 287  
    baseado em programs, 133  
    de caminho básico, 55  
    de componentes, 119, 165  
    de domínios, 279  
    de especificação, 260  
    de integração, 4, 128, 129, 169, 191, 203, 213, 220, 259  
    de mutação, 77–79, 83, 222, 243, 255  
    de mutação OO, 154  
    de partição, 5  
    de regressão, 4  
    de sistema, 4, 128  
    de subdomínio, 5, 77  
    de unidade, 4, 128, 191, 219  
    dinâmico, 218  
    embutido em componentes, 185  
    estático, 217  
    estrutural, 200, 211, 261–263  
    Evolucionário, 277  
    exaustivo, 5  
    funcional, 9–11  
    funcional sistemático, 15  
    interclasse, 162  
    intermétodo, 133, 160  
    intraclasse, 133  
    intramétodo, 133, 155  
    misto, 260  
    orientado a objetos, 119
- URL, 210  
Uso  
    computacional, 50, 54, 55, 239  
    comunicacional, 239  
    de variável, 49  
    predicativo, 50, 54, 55, 239
- Validação, 1  
ValiPar, 247  
Verificação, 1  
Vizinhança, 81, 84  
VV&T, 1  
    dinâmica, 2  
    estática, 2
- Walkthrough, 2, 261
- XML, 210