Teste Estrutural e de Mutação no Contexto de Programas OO

Ellen Francine Barbosa
José Carlos Maldonado
Departamento de Ciências de Computação
Instituto de Ciências Matemáticas e de Computação (ICMC)
Universidade de São Paulo – Campus de São Carlos
Caixa Postal 668
13560-970 São Carlos, SP
{francine, jcmaldon}@icmc.usp.br

Auri Marcelo Rizzo Vincenzi
Instituto de Informática
Universidade Federal de Goiás (UFG)
Caixa Postal 131
74001-970 Goiânia, GO
auri@inf.ufg.br

Márcio Eduardo Delamaro
Faculdade de Informática de Marília
Fundação de Ensino Eurípides Soares da Rocha (UNIVEM)
Caixa Postal 2041
17525-901 Marília, SP
delamaro@fundanet.br

Resumo

As exigências por software com maior qualidade têm motivado a definição de métodos e técnicas para o desenvolvimento de software que atinjam os padrões de qualidade impostos. Com isso, o interesse pela atividade de teste de software vem aumentando nos últimos anos. Vários pesquisadores têm investigado os diferentes critérios de teste, buscando obter uma estratégia de teste com baixo custo de aplicação e, ao mesmo tempo, com grande capacidade em revelar erros. O objetivo deste minicurso é apresentar os aspectos teóricos e práticos relacionados à atividade de teste de software, tanto no contexto do paradigma procedimental quanto orientado a objeto. Em particular, ênfase é dada aos critérios de teste baseados em Fluxo de Dados e em Mutação, com o apoio das ferramentas PokeTool e Proteum (teste de programas C), e JaBUTi (teste de programas Java em nível de bytecode), desenvolvidas no contexto do grupo de Engenharia de Software do ICMC/USP. Perspectivas e trabalhos de pesquisa sendo realizados na área de teste também são brevemente discutidos.

1 Introdução: Terminologia e Conceitos Básicos

A crescente utilização de sistemas baseados em computação em praticamente todas as áreas da atividade humana tem provocado grande demanda por qualidade e produtividade, tanto do ponto de vista do processo de produção como do ponto de vista dos produtos de software gerados. Nesse contexto, a Engenharia de Software – disciplina que aplica os princípios de engenharia com o objetivo de produzir software de alta qualidade a baixo custo [73] – evoluiu significativamente nas últimas décadas. Por meio de um conjunto de etapas que envolvem o desenvolvimento e aplicação de métodos, técnicas, critérios e ferramentas, a Engenharia de Software busca oferecer meios para que tais objetivos possam ser alcançados.

No entanto, apesar das técnicas, critérios, métodos e ferramentas empregados, erros[1] no produto ainda podem ocorrer. Atividades agregadas sob o nome de Garantia de Qualidade de Software têm sido introduzidas ao longo de todo o processo de desenvolvimento, entre elas atividades de VV&T (Verificação, Validação e Teste), com o objetivo de minimizar a ocorrência de erros e riscos associados. A verificação visa a assegurar que o software, ou uma determinada função do mesmo, esteja sendo implementado corretamente. Verifica-se, inclusive, se os métodos e processos de desenvolvimento foram adequadamente aplicados. A validação, por sua vez, procura assegurar que o software sendo desenvolvido é o software correto, de acordo com os requisitos do usuário.

Dentre as técnicas de verificação e validação, a atividade de teste é uma das mais utilizadas, constituindo-se em um dos elementos para fornecer evidências da confiabilidade do software em complemento a outras atividades como, por exemplo, o uso de revisões e de técnicas formais e rigorosas de especificação e de verificação [57]. De fato, os testes são considerados elementos críticos para a garantia da qualidade do software [14].

A atividade de teste consiste em uma análise dinâmica do produto, sendo relevante para a identificação e eliminação dos erros que persistem, representando a última revisão da especificação, projeto e codificação [38, 57, 73, 100]. Segundo Myers [65], o objetivo principal do teste de software é revelar a presença de erros ou defeitos no produto. Nesse sentido, o teste bem sucedido é aquele que consegue determinar casos de teste para os quais o programa sendo testado falhe. Salienta-se, entretanto, que a atividade de teste tem sido apontada entre as mais onerosas no desenvolvimento de software, podendo, em alguns casos, consumir grande parte dos custos de desenvolvimento [73].

Apesar de não ser possível, por meio de testes, provar que um programa está correto, estes contribuem para aumentar a confiança de que o software desempenha as funções especificadas. Além disso, apesar das limitações próprias da atividade de teste, sua aplicação de maneira sistemática e bem planejada pode garantir ao software algumas características mínimas, importantes tanto para o estabelecimento da qualidade do produto como para o seu processo de evolução.

O teste de produtos de software envolve basicamente quatro etapas: planejamento de testes, projeto de casos de teste, execução e avaliação dos resultados dos testes [4, 6, 14, 65, 73]. Tais atividades devem ser desenvolvidas ao longo do próprio processo de desenvolvimento de software e, em geral, concretizam-se em três fases [73]: teste de unidade, teste de integração e teste de sistema. O teste de unidade concentra esforços na menor unidade do projeto de software, ou seja, procura

[1] A IEEE tem realizado vários esforços de padronização, entre eles a padronização da terminologia utilizada no contexto de Engenharia de Software. O padrão IEEE 610.12-1990 [49] diferencia os termos: defeito (*fault*) – passo, processo ou definição de dados incorreto, como uma instrução ou comando incorreto; engano (*mistake*) – ação humana que produz um resultado incorreto, como uma ação incorreta tomada pelo programador; erro (*error*) – diferença entre o valor obtido e o valor esperado, ou seja, qualquer estado intermediário incorreto ou resultado inesperado na execução do programa constitui um erro; e falha (*failure*) – produção de uma saída incorreta com relação à especificação. Neste texto, os termos engano, defeito e erro serão referenciados como erro (causa) e o termo falha (conseqüência) a um comportamento incorreto do programa.

identificar erros de lógica e de implementação em cada módulo do software, separadamente. O teste de integração é uma atividade sistemática aplicada durante a integração da estrutura do programa visando a descobrir erros associados às interfaces entre os módulos; o objetivo é, a partir dos módulos testados no nível de unidade, construir a estrutura de programa que foi determinada pelo projeto. O teste de sistema, realizado após a integração do sistema, visa a identificar erros de funções e características de desempenho que não estejam de acordo com a especificação.

Um ponto crucial da atividade de teste é o projeto e/ou avaliação dos casos de teste a serem utilizados. O programa, em princípio, deveria ser exercitado com todos os valores do domínio de entrada. Sabe-se, entretanto, que o teste exaustivo é impraticável por razões de custo e tempo. Dessa forma, por questões de produtividade, o objetivo é utilizarem-se casos de teste que tenham alta probabilidade de revelar a presença da maioria dos erros existentes com o mínimo de tempo e esforço, por questões de produtividade.

Dentro dessa perspectiva, para que a atividade de teste possa ser conduzida de forma sistemática e teoricamente fundamentada, faz-se necessária a aplicação de técnicas e critérios que indiquem como testar o software, quando parar os testes e que, se possível, forneçam uma medida objetiva do nível de confiança e de qualidade alcançados com os testes realizados [20]. Em geral, os critérios de teste de software são estabelecidos, basicamente, a partir das técnicas: funcional, estrutural, baseada em erros e baseada em estados. Na técnica funcional, os requisitos de teste são estabelecidos a partir da especificação do software. Na técnica estrutural, os requisitos são derivados a partir dos aspectos de implementação do software. Na técnica baseada em erros, os requisitos de teste são obtidos a partir do conhecimento sobre erros típicos cometidos durante o processo de desenvolvimento de software. Na técnica baseada em estados, os requisitos de teste são derivados a partir da especificação representada por um modelo de estados como, por exemplo, uma máquina de estado finito [12] ou um statechart [37].

É importante ressaltar que as técnicas de teste devem ser vistas como complementares e a questão que se coloca é como utilizá-las de forma que as vantagens de cada uma sejam melhor exploradas em uma estratégia de teste que leve a uma atividade de teste de boa qualidade, eficaz e de baixo custo [4, 14, 73, 93]. De fato, dada a diversidade de critérios existentes para cada uma das técnicas e reconhecido seu caráter complementar, o estabelecimento de estratégias de teste com tais características é fundamental. Estudos teóricos e empíricos, que proporcionem uma síntese do conhecimento sobre vantagens, desvantagens e limitações de cada um dos critérios de teste, têm sido conduzidos nessa direção.

Outro aspecto relevante associado à atividade de teste refere-se ao desenvolvimento de ferramentas que automatizem a aplicação das técnicas e critérios associados. Sem a utilização de ferramentas automatizadas como mecanismos de apoio, a atividade de teste tende a ser extremamente trabalhosa, propensa a erros e limitada a programas muito simples [46]. Além de contribuir para a qualidade e a produtividade dos testes, a existência de ferramentas automatizadas: (1) viabiliza a realização de estudos empíricos; (2) auxilia a condução dos testes de regressão; e (3) apóia o processo de ensino e aprendizado envolvendo a aplicação prática dos conceitos de teste.

Em linhas gerais, observa-se que os critérios baseados em Fluxo de Dados [44, 53, 66, 75, 76, 88] e o critério Análise de Mutantes [10, 21, 22] têm sido fortemente investigados por diversos pesquisadores, sob diferentes aspectos. Resultados desses estudos fornecem evidências de que esses critérios, hoje investigados fundamentalmente no meio acadêmico, às vezes em cooperação com a indústria, podem, em médio prazo, constituir o estado da prática em ambientes de produção de software.

O paradigma de desenvolvimento orientado a objetos (OO), o qual tem sido cada vez mais utilizado, em grande parte devido ao seu potencial para reutilização, constitui outro fator relevante a ser considerado no contexto da atividade de teste. O teste de programas OO e o teste de

componentes devem lidar com novos problemas introduzidos pelas características das linguagens OO. Encapsulamento, herança, polimorfismo e acoplamento dinâmico, embora tragam benefícios para o projeto e a codificação, oferecem novos desafios para as áreas de teste e manutenção [63]. Atualmente, a maioria das organizações desenvolvedoras de software ainda estão no processo de transição para o paradigma OO e, à medida que mais e mais organizações adotarem tal paradigma, maior será a demanda por técnicas, critérios e ferramentas que apóiem o teste de sistemas desse tipo [38, 52, 92]. Apesar de diversos trabalhos nessa direção, ainda são poucas as evidências sobre a eficácia das técnicas e critérios propostos. Como destacado por Offutt e Irvine [68], os critérios de teste utilizados no paradigma procedimental precisam ser reavaliados e, quando necessário, passar por adaptações que permitam sua utilização no teste de programas OO.

O grupo de Engenharia de Software do ICMC/USP, em colaboração com outros grupos de pesquisa na área de teste, vem desenvolvendo atividades de pesquisa concentradas no estudo de princípios, estratégias, métodos e critérios de teste e validação de software, bem como na especificação e implementação de ferramentas de teste que apóiem a realização das atividades de teste e viabilizem a avaliação do aspecto complementar dos critérios de teste por meio de estudos empíricos. Dentre os critérios de teste investigados pelo grupo destacam-se os critérios baseados em Fluxo de Dados e os critérios baseados em Mutação, explorados tanto no contexto do paradigma procedimental como OO. No que se refere às ferramentas de teste desenvolvidas destacam-se: *PokeTool* [11] (apoio à aplicação de critérios estruturais no teste de programas C); *Proteum* [15] e *Proteum/IM* [17] (apoio à aplicação dos critérios Análise de Mutantes e Mutação de Interface, respectivamente, no teste de programas C); e *JaBUTi* [92, 94] (teste de bytecode Java). Além disso, no contexto do teste de especificações, destaca-se o desenvolvimento de critérios baseados em Mutação para o teste de Máquinas de Estado Finito (MEFs) [24, 27], Statecharts [25, 29] e Redes de Petri [28, 81], apoiados pelas ferramentas *Proteum/FSM* [24], *Proteum/ST* [29] e *Proteum/PN* [28, 81, 82], respectivamente. Mais recentemente, também tem sido explorado o desenvolvimento de critérios de teste baseados em Mutação para Estelle [84] e SDL [86].

O presente texto visa a abordar os aspectos teóricos e práticos relacionados à atividade de teste de software, tanto no contexto do paradigma de desenvolvimento procedimental quanto orientado a objeto. Com esse objetivo em mente, o texto está organizado da seguinte forma[2]: nesta seção foram introduzidos os conceitos básicos e a terminologia pertinentes ao teste de software. Na Seção 2 são apresentados os critérios de teste mais difundidos das técnicas funcional, estrutural, baseada em erros e baseada em estados. Em particular, ênfase é dada às técnicas baseadas em Fluxo de Dados e em Mutação, com o apoio das ferramentas de teste *PokeTool* e *Proteum*, respectivamente. Na Seção 3 são discutidas algumas das principais questões relacionadas ao teste OO. Inicialmente, as fases de teste para programas OO são caracterizadas, contrapondo-se com as fases de teste para programas procedimentais. Em seguida, é discutido o impacto das características de encapsulamento, herança, polimorfismo e acoplamento dinâmico na definição e aplicação de critérios de teste. Uma visão geral sobre os principais critérios de teste investigados no contexto de desenvolvimento OO também é apresentada. Além disso, aspectos operacionais da ferramenta *JaBUTi* são brevemente discutidos. Na Seção 4 são discutidos alguns aspectos referentes a estudos teóricos e empíricos, conduzidos a fim de avaliar e comparar os diversos critérios de teste existentes. Finalmente, na Seção 5 são apresentadas as conclusões e perspectivas de trabalhos futuros na área de teste de software.

[2]Os aspectos discutidos neste texto foram extraídos essencialmente de [4, 58, 61, 92, 93].

2 Técnicas e Critérios de Teste

Conforme mencionado na Seção 1, para se conduzir e avaliar a qualidade da atividade de teste utilizam-se as técnicas de teste funcional, estrutural, baseada em erros e baseada em estados. Tais técnicas diferenciam-se pela origem da informação utilizada na avaliação e construção dos conjuntos de casos de teste [57].

Além disso, segundo Howden [48], o teste pode ser classificado de duas maneiras: teste baseado em especificação (*specification-based testing*) e teste baseado em programa (*program-based testing*). De acordo com tal classificação, têm-se que os critérios das técnicas funcional e baseada em estado são baseados em especificação, e os critérios das técnicas estrutural e baseada em erros são considerados critérios baseados em programa.

No teste baseado em especificação, o objetivo é determinar se o programa satisfaz aos requisitos funcionais e não-funcionais especificados. O problema é que, em geral, a especificação existente é informal e, desse modo, a determinação da cobertura total da especificação que foi obtida por um dado conjunto de casos de teste também é informal [72]. Por outro lado, os critérios de teste baseados em especificação podem ser utilizados em qualquer contexto (procedimental ou OO) e em qualquer fase de teste (unidade, integração, sistema) sem a necessidade de modificação.

O teste baseado em programa requer a inspeção do código-fonte e a seleção de casos de teste que exercitem partes do código e não de sua especificação [72]. O objetivo é identificar erros na estrutura interna do programa. A desvantagem dessa abordagem é que a mesma pode ser dependente da linguagem e requer o acesso ao código-fonte para ser aplicada.

Nesta seção apresentam-se com mais detalhes as técnicas estrutural e baseada em erros, mais especificamente os critérios baseados em Fluxo de Dados [57, 76] e em Mutação [16, 22]. Por meio desses critérios, ilustram-se os principais aspectos pertinentes à atividade de teste de software. O programa `identifier` (Figura 1) será utilizado para facilitar a ilustração dos conceitos desenvolvidos no texto. Para propiciar uma visão mais abrangente apresenta-se, primeiramente, uma visão geral das técnicas baseada em estados e funcional, bem como alguns de seus critérios mais conhecidos.

2.1 Técnica Baseada em Estados

O teste baseado em estados utiliza uma representação baseada em estados para modelar o comportamento do sistema ou unidade que será testada. Com base nesse modelo, critérios de geração de seqüências de teste podem ser utilizados de modo a garantir o seu correto funcionamento.

Um dos critérios de geração de seqüências de teste, baseado em Máquinas de Estado Finito (MEFs), é o critério W [12]. Além do critério W, podem ser encontrados na literatura outros critérios, tais como DS [36], UIO [79] e Wp [33]. Critérios baseados em mutação também têm sido investigados para a geração de conjuntos de casos de teste para MEFs [26, 27].

Dada a própria natureza dos objetos de englobarem estado e comportamento, os critérios baseados em MEFs também são bastante utilizados no contexto de OO, a fim de representar o aspecto comportamental dos objetos [7,8,45,63,64,87]. Tal assunto será brevemente retomado na Seção 3.

2.2 Técnica Funcional

O teste funcional, também conhecido como teste caixa preta, trata o software como uma caixa cujo conteúdo é desconhecido e da qual só é possível visualizar o lado externo, ou seja, os dados de entrada fornecidos e as respostas produzidas como saída [6, 65]. O testador utiliza, essencialmente, a especificação de requisitos do programa para derivar os requisitos de testes, ou mesmo os casos de teste que serão empregados, sem se importar com os detalhes de implementação [6].

```
/*****************************************************************************
Identifier.c
ESPECIFICACAO: O programa deve determinar se um identificador eh ou nao valido em
'Silly Pascal' (uma estranha variante do Pascal).  Um identificador valido deve
comecar com uma letra e conter apenas letras ou digitos.  Alem disso, deve ter no
minimo 1 caractere e no maximo 6 caracteres de comprimento
*****************************************************************************/
```

```
            #include <stdio.h>
            main ()
 /*  1 */   {
 /*  1 */       char  achar;
 /*  1 */       int length, valid_id;
 /*  1 */       length = 0;
 /*  1 */       valid_id = 1;
 /*  1 */       printf ("Identificador: ");
 /*  1 */       achar = fgetc (stdin);
 /*  1 */       valid_id = valid_s(achar);
 /*  1 */       if(valid_id)
 /*  2 */       {
 /*  2 */           length = 1;
 /*  2 */       }
 /*  3 */       achar = fgetc (stdin);
 /*  4 */       while(achar != '\n')
 /*  5 */       {
 /*  5 */           if(!(valid_f(achar)))
 /*  6 */           {
 /*  6 */               valid_id = 0;
 /*  6 */           }
 /*  7 */           length++;
 /*  7 */           achar = fgetc (stdin);
 /*  7 */       }
 /*  8 */       if(valid_id &&
                  (length >= 1)&&(length < 6))
 /*  9 */       {
 /*  9 */           printf ("Valido\n");
 /*  9 */       }
 /* 10 */       else
 /* 10 */       {
 /* 10 */           printf ("Invalid\n");
 /* 10 */       }
 /* 11 */   }
```

```
            int valid_s(char ch)
 /* 1 */    {
 /* 1 */       if(((ch >= 'A') &&
                   (ch <= 'Z')) ||
                  ((ch >= 'a') &&
                   (ch <= 'z')))
 /* 2 */       {
 /* 2 */           return (1);
 /* 2 */       }
 /* 3 */       else
 /* 3 */       {
 /* 3 */           return (0);
 /* 3 */       }
 /* 4 */    }

            int valid_f(char ch)
 /* 1 */    {
 /* 1 */        if(((ch >= 'A') &&
                    (ch <= 'Z')) ||
                   ((ch >= 'a') &&
                    (ch <= 'z')) ||
                   ((ch >= '0') &&
                    (ch <= '9')))
 /* 2 */        {
 /* 2 */            return (1);
 /* 2 */        }
 /* 3 */        else
 /* 3 */        {
 /* 3 */            return (0);
 /* 3 */        }
 /* 4 */    }
```

Figura 1: Programa exemplo: `identifier` (contém ao menos um erro).

Assim, uma especificação de qualidade e de acordo com os requisitos do usuário é de fundamental importância para apoiar a aplicação dos critérios relacionados a essa técnica. Alguns exemplos de critérios de teste funcional são [73]:

- Particionamento em Classes de Equivalência: A partir das condições de entrada de dados identificadas na especificação, o domínio de entrada de um programa é dividido em classes de equivalência válidas e inválidas. Em seguida, seleciona-se o menor número possível de casos de teste, baseando-se na hipótese de que um elemento de uma dada classe seria representativo da classe toda, sendo que para cada uma das classes inválidas deve ser gerado um caso de teste distinto. O uso de particionamento permite examinar os requisitos de forma mais sistemática e restringir o número de casos de teste existentes. Alguns autores também consideram o domínio de saída do programa para estabelecer as classes de equivalência.

- Análise do Valor Limite: É um complemento ao critério Particionamento em Classes de Equivalência, sendo que os limites associados às condições de entrada são exercitados de forma mais rigorosa. Ao invés de selecionar-se qualquer elemento de uma classe, os casos de teste são escolhidos nas fronteiras das classes, visto que são nesses pontos que se concentra um grande número de erros. O espaço de saída do programa também é particionado e são exigidos casos de teste que produzam resultados nos limites dessas classes de saída.

- Grafo de Causa-Efeito: Os critérios anteriores não exploram combinações das condições de entrada. Esse critério estabelece requisitos de teste baseados nas possíveis combinações das condições de entrada. Primeiramente, são levantadas as possíveis condições de entrada (causas) e as possíveis ações (efeitos) do programa. A seguir, constrói-se um grafo relacionando as causas e efeitos levantados. Esse grafo é convertido em uma tabela de decisão a partir da qual são derivados os casos de teste.

Conforme discutido anteriormente, um dos problemas relacionados aos critérios funcionais é que muitas vezes a especificação do programa é feita de modo descritivo e não formal. Dessa maneira, os requisitos de teste derivados de tais especificações são também, de certa forma, imprecisos e informais. Como conseqüência, tem-se dificuldade em automatizar a aplicação de tais critérios, que ficam, em geral, restritos à aplicação manual. Coloca-se, ainda, a dificuldade de quantificar a atividade de teste, uma vez que não se pode garantir que partes essenciais ou críticas do código do programa foram executadas. Por outro lado, visto que os critérios funcionais baseiam-se exclusivamente na especificação do software para derivar os requisitos de teste, tais critérios podem ser aplicados praticamente em todas as fases de teste e em programas construídos sob diferentes paradigmas de desenvolvimento [45, 63, 68, 93].

A título de ilustração, considere o programa `identifier` e o critério Particionamento em Classes de Equivalência. Na Tabela 1 são identificadas as condições de entrada e as classes de equivalência válidas e inválidas. A partir dessas classes, poderia ser elaborado o seguinte conjunto de casos de teste : T_0 = {(a1, Válido), (2B3, Inválido), (Z-12, Inválido), (A1b2C3d, Inválido)}. De posse do conjunto T_0, seria natural indagar se esse conjunto exercita todos os comandos ou todos os desvios de fluxo de controle de uma dada implementação. Usualmente, utilizam-se critérios estruturais de teste, apresentados a seguir, como critérios de adequação ou critérios de cobertura para se analisar questões como esta, propiciando a quantificação e a qualificação da atividade de teste de acordo com o critério escolhido. Assim, quanto mais rigoroso o critério utilizado e se erros não forem revelados, maior a confiança no produto em desenvolvimento.

Tabela 1: Classes de equivalência para o programa `identifier`.

Restrições de Entrada	**Classes Válidas**	**Classes Inválidas**
Tamanho (t) do identificador	$1 \leq t \leq 6$ (1)	$t > 6$ (2)
Primeiro caracter (c) é uma letra	Sim (3)	Não (4)
Contém somente caracteres válidos	Sim (5)	Não (6)

2.3 Técnica Estrutural

Na técnica de teste estrutural, também conhecida como teste caixa branca (em oposição ao nome caixa preta), os aspectos de implementação são fundamentais na escolha dos casos de teste. O teste estrutural baseia-se no conhecimento da estrutura interna da implementação. Em geral, a maioria dos critérios dessa técnica utiliza uma representação de programa conhecida como grafo de fluxo de controle (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.

A representação de um programa P como um grafo de fluxo de controle consiste em estabelecer uma correspondência entre nós e blocos e em indicar possíveis fluxos de controle entre blocos através dos arcos. Um grafo de fluxo de controle é portanto um grafo orientado, com um único nó de entrada e um único nó de saída, no qual cada vértice representa um bloco indivisível de comandos e cada aresta representa um possível desvio de um bloco para outro. Cada bloco tem as seguintes características: (1) uma vez que o primeiro comando do bloco é executado, todos os demais são executados seqüencialmente; e (2) não existe desvio de execução para nenhum comando dentro do bloco. A partir do grafo de programa podem ser escolhidos os elementos que devem ser executados, caracterizando assim o teste estrutural.

Considere o programa `identifier`. Na Figura 1 identifica-se a caracterização dos blocos de comandos por meio dos números à esquerda dos comandos. A Figura 2 ilustra o grafo de fluxo de controle do programa `identifier` (função `main`) gerado pela ferramenta *ViewGraph* [91].

Seja um grafo de fluxo de controle $G = (N, E, s)$ onde 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 $(n_1, n_2, \ldots, n_k)$, $k \geq 2$, tal que existe um arco de n_i para n_i+1 para $i = 1, 2, \ldots, k-1$. 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 um caminho no qual o primeiro nó é o nó de entrada e o último nó é o 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. Se $IN(x) = 0$ x é uma nó de entrada, e se $OUT(x) = 0$, x é um nó de saída.

Em relação ao programa `identifier`, (2,3,4,5,6,7) é um caminho simples e livre de laços e o caminho (1,2,3,4,5,7,4,8,9,11) é um caminho completo. Observe que o caminho (6,7,4,5,7,4,8,9) é não executável e qualquer caminho completo que o inclua é também não executável, ou seja, não existe um dado de entrada que leve à execução desse caminho.

Os critérios de teste estrutural baseiam-se em diferentes tipos de conceitos e elementos de programa para determinar os requisitos de teste. Na Tabela 2 ilustram-se alguns desses elementos e critérios associados.

Os critérios de teste estrutural são, em geral, classificados em:

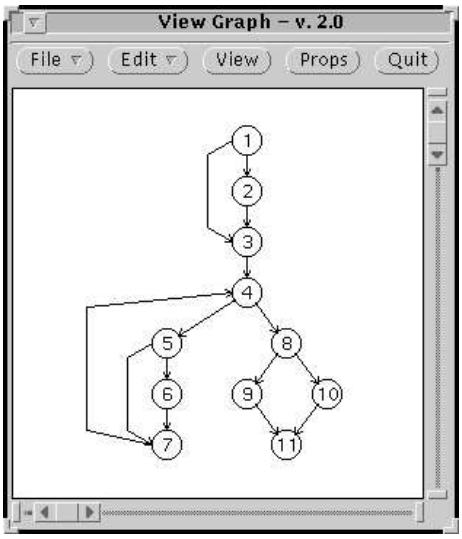


Figura 2: Grafo de fluxo de controle do programa `identifier` gerado pela *ViewGraph*.

Tabela 2: Elementos e critérios associados em relação ao programa `identifier`.

Elemento	**Exemplo (`identifier`)**	**Critério**
Nó	6	Todos-Nós
Arco	(7,4)	Todos-Arcos
Laço	(4,5,6,7,4)	*Boundary-Interior*
Caminho	(1,2,3,4,8,9,11)	Todos-Caminhos
Definição de variáveis	`length=0`	Todas-Defs
Uso predicativo de variáveis	`achar != '\n'`	Todos-P-Usos
Uso computacional de variáveis	`length++`	Todos-C-Usos

- 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 [73]: Todos-Nós – exige que a execução do programa passe, ao menos uma vez, em cada vértice do grafo de fluxo, ou seja, que cada comando do programa seja executado pelo menos uma vez; Todos-Arcos – requer que cada aresta do grafo, ou seja, cada desvio de fluxo de controle do programa, seja exercitada pelo menos uma vez; e Todos-Caminhos – requer que todos os caminhos possíveis do programa sejam executados. Outros critérios dessa categoria são: Cobertura de Decisão; Cobertura de Condição; Cobertura de Condições Múltiplas; LCSAJ (*Linear Code Sequence and Jump*) [99]; o critério *Boundary-Interior* [47]; e a família de critérios *K*-tuplas requeridas, de Ntafos [66].

- Critérios Baseados em Fluxo de Dados: Utilizam informações do fluxo de dados do programa para determinar os requisitos de teste. Esses critérios exploram as interações que envolvem definições de variáveis e referências a tais definições para estabelecerem os requisitos de teste [76]. Exemplos dessa classe de critérios são os critérios de Rapps e Weyuker [75, 76] e os critérios Potenciais-Usos [57]. Tais critérios serão descritos mais detalhadamente nas próximas seções.

- 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, que utiliza a complexidade ciclomática do grafo de programa para derivar os requisitos de teste. Essencialmente, esse critério requer que um conjunto de caminhos linearmente independentes do grafo de programa seja executado [73].

A técnica estrutural apresenta uma série de limitações e desvantagens. Um dos principais problemas refere-se à impossibilidade, em geral, de se determinar automaticamente se um caminho é executável ou não. Ou seja, não existe um algoritmo que dado um caminho completo qualquer, decida se o caminho é executável e forneça o conjunto de valores que causam a execução desse caminho [89]. Assim, é preciso a intervenção do testador para determinar quais são os caminhos não executáveis para o programa sendo testado.

Independentemente dessas desvantagens, essa técnica é vista como complementar à técnica funcional [73]. De fato, é importante observar que os casos de teste obtidos durante a aplicação dos critérios funcionais podem corresponder ao conjunto inicial dos testes estruturais. Como, em geral, o conjunto de casos de teste funcional não é suficiente para satisfazer totalmente um critério de teste estrutural, novos casos de teste são gerados e adicionados ao conjunto até que se atinja o grau de satisfação desejado, explorando-se, desse modo, os aspectos complementares das duas técnicas [83]. Ainda, informações obtidas pela aplicação desses critérios têm sido consideradas relevantes para as atividades de manutenção, depuração e confiabilidade de software [6, 73].

2.3.1 Critérios Baseados em Fluxo de Dados

Os critérios baseados em Fluxo de Dados [44], propostos em meados da década de 70, utilizam informações do fluxo de dados para derivar os requisitos de teste. Uma característica comum de tais critérios é que eles requerem que sejam testadas as interações que envolvam definições de variáveis e subseqüentes referências a essas definições [44, 53, 66, 76, 88].

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 até mesmo de erros simples e triviais. A introdução dessa classe de critérios procurou fornecer uma hierarquia entre os critérios Todos-Arcos e Todos-Caminhos, visando a tornar o teste mais rigoroso, já que o teste de Todos-Caminhos é, em geral, impraticável.

Dentre os critérios de Fluxo de Dados, destacam-se os critérios de Rapps e Weyuker [76], introduzidos nos anos 80. Rapps e Weyuker propuseram o Grafo Def-Uso (*Def-Use Graph*) que consiste em uma extensão do grafo de programa [75, 76]. Nele 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 (chamado de definição da variável) e pontos em que esse valor é utilizado (chamado de referência ou uso da variável). Os requisitos de teste são determinados com base em tais associações.

A Figura 3 ilustra o Grafo-Def-Uso do programa `identifier`. Conforme o modelo de fluxo de dados definido em [57], 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 [34]. 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. A ocorrência de uma variável é um uso quando a referência a essa variável não a estiver definindo. Dois tipos de usos são distinguidos: *c-uso* e *p-uso*. O primeiro tipo afeta diretamente uma computação sendo realizada ou permite que o

resultado de uma definição anterior possa ser observado; o segundo tipo afeta diretamente o fluxo de controle do programa.

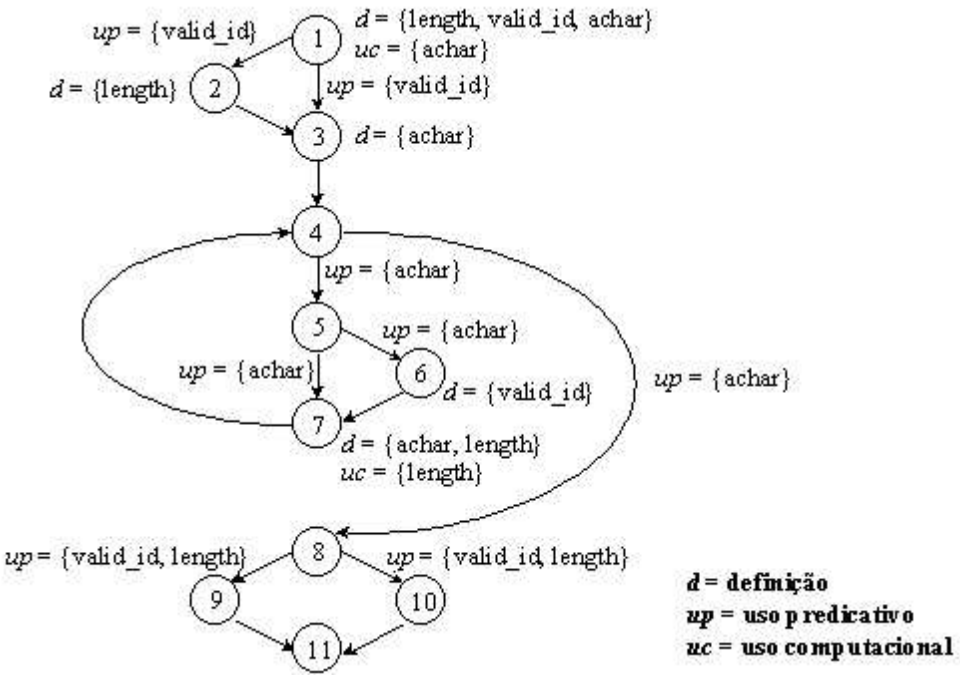


Figura 3: Grafo def-uso do programa `identifier`.

O critério mais básico da família de critérios definidos por Rapps e Weyuker [76] é o critério Todas-Definições. Entre os critérios dessa família, o critério Todos-Usos tem sido um dos mais utilizados e investigados.

- 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, através de pelo menos um caminho livre de definição, ou seja, um caminho onde a variável não é redefinida.

Como exemplo, para exercitar a definição da variável `length` definida no nó 1, de acordo com o critério Todas-Definições, poderiam ser executados 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). O subcaminho (1,3,4,8,9) é não executável, assim como qualquer caminho completo que o inclua. Se qualquer um dos demais caminhos for exercitado, o requisito de teste é satisfeito. Para satisfazer o critério Todas-Definições, esta análise deve ser feita para toda definição que ocorre no programa.

Em relação ao critério Todos-Usos, com respeito à mesma definição, são requeridas as seguintes associações: (1,7, `length`); (1,(8,9),`length`) e (1,(8,10),`length`). As notações (i,j,var) e (i,(j, k),var) indicam que a variável var é definida no nó i e existe um uso computacional de var no nó j ou um uso predicativo 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). Observe que a associação (1,(8,9), `length`) é não executável pois o único caminho que livre de definição possível de exercitá-la seria um caminho que incluísse o subcaminho (1,3,4,8,9). Já para a associação (1,7,`length`),

qualquer caminho completo executável incluindo um dos subcaminhos (1,3,4,5,6,7), (1,3,4,5,7) é suficiente para exercitá-la. Esta mesma análise deve ser feita para todas as demais variáveis e associações pertinentes, a fim de satisfazer o critério Todos-Usos.

A maior parte dos critérios baseados em Fluxo de Dados, para requerer um determinado elemento (caminho, associação, etc.), exige a ocorrência explícita de um uso de variável e não garante, necessariamente, a inclusão dos critérios Todos-Arcos na presença de caminhos não executáveis, presentes na maioria dos programas [57]. Com a introdução do conceito de potencial-uso, nos início dos anos 90, Maldonado [57] definiu a família de critérios Potenciais-Usos e a correspondente família de critérios executáveis, obtida pela eliminação dos caminhos e associações não 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, requerem que caminhos livres de definição a partir da definição de uma determinada variável sejam executados, independentemente de ocorrer um uso dessa variável nesse caminho.

Os critérios básicos que fazem parte dessa família de critérios são [57]:

- 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[3] 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.

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 grafo de programa associando a cada nó do grafo informações a respeito das definições que ocorrem nesses nós. Tal grafo é denominado de Grafo Def [57].

Como exemplo, tem-se que as potenciais-associações (1,6,`length`) e (7,6,`length`) são requeridas pelo critério Todos-Potenciais-Usos [57], não sendo requeridas pelos demais critérios de Fluxo de Dados que não fazem uso do conceito potencial-uso. Observe que, por definição, toda associação é uma potencial-associação. Dessa forma, as associações requeridas pelo critério Todos-Usos são um subconjunto das potenciais-associações requeridas pelo critério Todos-Potenciais-Usos.

A relação de inclusão é uma importante propriedade dos critérios, sendo utilizada para avaliá-los, do ponto de vista teórico. O critério Todos-Arcos, por exemplo, inclui o critério Todos-Nós, ou seja, qualquer conjunto de casos de teste que satisfaz o critério Todos-Arcos também satisfaz o critério Todos-Nós, necessariamente. Quando não é possível estabelecer essa ordem de inclusão para dois critérios, como é o caso de Todas-Definições e Todos-Arcos, diz-se que tais critérios são incomparáveis [76]. Deve-se observar que os critérios Potenciais-Usos são os únicos critérios baseados em Fluxo de Dados que satisfazem, na presença de caminhos não executáveis, as propriedades mínimas esperadas de um critério de teste, e que nenhum outro critério baseado em Fluxo de Dados os inclui. Um aspecto relevante é que alguns dos critérios Potenciais-Usos "*bridge*

[3]Um potencial-du-caminho em relação à variável x é um caminho livre de definição $(n_1, ..., n_j, n_k)$ com relação a x do nó n_1 para o nó n_k e para o arco (n_j, n_k), onde o caminho $(n_1, ..., n_j)$ é um caminho livre de laço e no nó n_1 ocorre uma definição de x.

the gap" entre os critérios Todos-Arcos e Todos-Caminhos mesmo na presença de caminhos não executáveis, o que não ocorre para os demais critérios baseados em Fluxo de Dados.

Como já citado, uma das desvantagens do teste estrutural é a existência de caminhos 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 conseqüência, nenhum caso de teste será requerido para exercitá-la. Mesmo assim, esses 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 P. Esse rigor na definição dos requisitos favorece a automatização desses critérios.

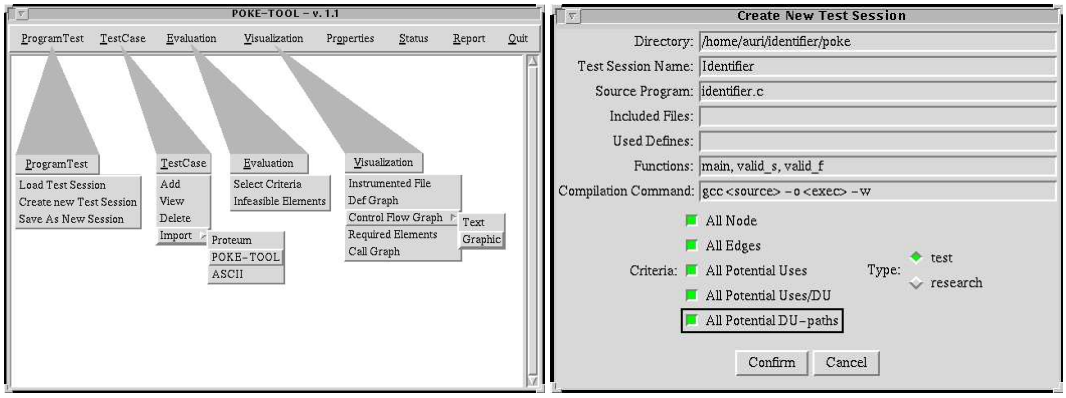
Os critérios estruturais têm sido utilizados principalmente no teste de unidade, visto que os requisitos de teste por eles exigidos limitam-se ao escopo da unidade. Na tentativa de estender o uso de tais critérios para diferentes contextos, alguns esforços podem ser identificados. Observam-se na literatura extensões dos critérios baseados em Fluxo de Dados tanto para o teste de integração em programas procedimentais [42, 43, 90], quanto para o teste de unidade e integração em programas OO [41, 92]. Vincenzi et al. [92, 93], por exemplo, têm investigado o uso de critérios de Fluxo de Controle e de Dados no teste de programas OO e de componentes. Visando a desenvolver uma solução aplicável tanto a programas OO quanto componentes de software (os quais, em geral, são testados pelos clientes utilizando somente técnicas funcionais), investigou-se como realizar análise estática de programas Java diretamente a partir do código objeto (bytecode Java). Com isso, independentemente da existência do código-fonte da aplicação sendo testada, é possível derivar requisitos de teste estruturais os quais podem ser utilizados tanto para avaliar a qualidade de conjuntos de teste quanto para a própria geração de casos de teste. Tais aspectos são retomados e discutidos mais detalhadamente na Seção 3.

2.3.2 A Ferramenta de Teste *PokeTool*

Para ilustrar os conceitos abordados será utilizada a ferramenta *PokeTool* (*Potential Uses Criteria Tool for Program Testing*) [11, 59], desenvolvida na FEEC/UNICAMP, em colaboração com o ICMC/USP. Essa ferramenta apóia a aplicação dos critérios Potenciais-Usos e também de outros critérios estruturais, como Todos-Nós e Todos-Arcos, no teste de programas C. A Figura 4(a) mostra a tela principal da ferramenta e as principais funções fornecidas.

A *PokeTool* é orientada à sessão de trabalho. O termo sessão trabalho (ou sessão de teste) é utilizado para designar as atividades envolvendo um teste. O teste pode ser realizado em etapas nas quais são armazenados os estados intermediários da aplicação de teste a fim de que possam ser recuperados posteriormente. Desse modo, é possível ao usuário iniciar e encerrar o teste de um programa, bem como retomá-lo a partir de onde este foi interrompido. Basicamente, o usuário entra com o programa a ser testado, com o conjunto de casos de teste e seleciona todos ou alguns dos critérios disponíveis (Todos-Potenciais-Usos, Todos-Potenciais-Usos/Du, Todos-Potenciais-Du-Caminhos, Todos-Nós e Todos-Arcos). Como saída, a ferramenta fornece ao usuário o conjunto de arcos primitivos[4] [13], o Grafo Def obtido do programa em teste, o programa instrumentado para teste, o conjunto de associações necessárias para satisfazer o critério selecionado e o conjunto de associações ainda não exercitadas. A Figura 4(b) mostra a criação de uma sessão de teste para o programa `identifier` utilizando todos os critérios apoiados pela ferramenta.

[4] O conjunto de arcos primitivos consiste de arcos que uma vez executados garantem a execução de todos os demais arcos do grafo de programa.



(a) Opções disponíveis na ferramenta *PokeTool*. (b) Tela para criar uma sessão de teste.

Figura 4: Ferramenta *PokeTool*.

Atualmente, a ferramenta encontra-se disponível para os ambientes DOS e UNIX. A versão para DOS possui interface simples, baseada em menus. A versão para UNIX possui módulos funcionais cuja utilização se dá por meio de interface gráfica ou linha de comando (*shell scripts*).

A título de ilustração, considere o programa `identifier` e os critérios Todos-Arcos e Todos-Potenciais-Usos. As tabelas 3 e 4 trazem os elementos requeridos por esses critérios, respectivamente. Utilizando o conjunto de casos de teste T_0 = {(a1, Válido), (2B3, Inválido), (Z-12, Inválido), (A1b2C3d, Inválido)}, gerado anteriormente a fim de satisfazer o critério Particionamento em Classes de Equivalência, é possível observar qual a cobertura obtida em relação aos critérios Todos-Arcos e Todos-Potenciais-Usos (Figura 5(a) e Figura 5(b), respectivamente). Ainda na Figura 5(b), são ilustrados para o critério Todos-Potenciais-Usos os elementos requeridos e não executados quando a cobertura é inferior a 100%.

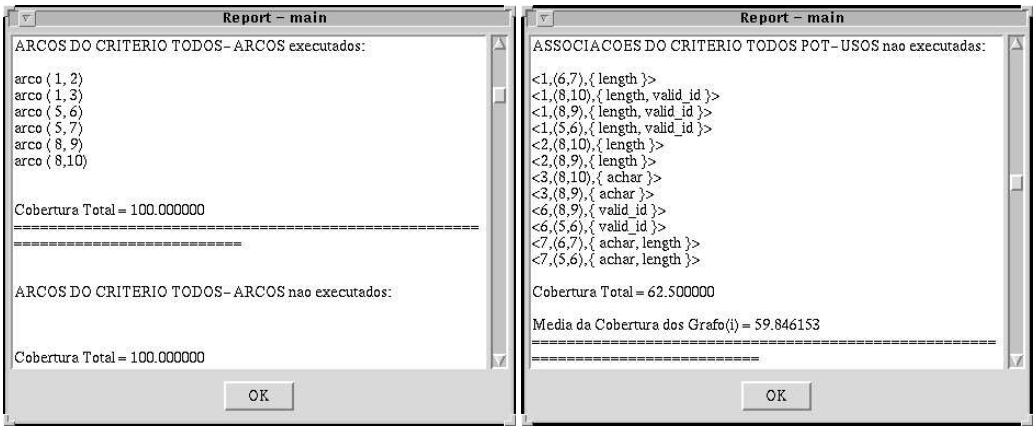
Tabela 3: Elementos requeridos pelo critério Todos-Arcos.

Arcos Primitivos					
Arco (1,2)	Arco (1,3)	Arco (5,6)	Arco (5,7)	Arco (8,9)	Arco (8,10)

Tabela 4: Elementos requeridos pelo critério Todos-Potenciais-Usos.

Associações Requeridas	
1) $\langle 1, (6,7), \{\texttt{length}\}\rangle$	17) $\langle 2, (6,7), \{\texttt{length}\}\rangle$
2) $\langle 1, (1,3), \{\texttt{achar}, \texttt{length}, \texttt{valid\_id}\}\rangle$	18) $\langle 2, (5,6), \{\texttt{length}\}\rangle$
3) $\langle 1, (8,10), \{\texttt{length}, \texttt{valid\_id}\}\rangle$	19) $\langle 3, (8,10), \{\texttt{achar}\}\rangle$
4) $\langle 1, (8,10), \{\texttt{valid\_id}\}\rangle$	20) $\langle 3, (8,9), \{\texttt{achar}\}\rangle$
5) $\langle 1, (8,9), \{\texttt{length}, \texttt{valid\_id}\}\rangle$	21) $\langle 3, (5,7), \{\texttt{achar}\}\rangle$
6) $\langle 1, (8,9), \{\texttt{valid\_id}\}\rangle$	22) $\langle 3, (6,7), \{\texttt{achar}\}\rangle$
7) $\langle 1, (7,4), \{\texttt{valid\_id}\}\rangle$	23) $\langle 3, (5,6), \{\texttt{achar}\}\rangle$
8) $\langle 1, (5,7), \{\texttt{length}, \texttt{valid\_id}\}\rangle$	24) $\langle 6, (8,10), \{\texttt{valid\_id}\}\rangle$
9) $\langle 1, (5,7), \{\texttt{valid\_id}\}\rangle$	25) $\langle 6, (8,9), \{\texttt{valid\_id}\}\rangle$
10) $\langle 1, (5,6), \{\texttt{length}, \texttt{valid\_id}\}\rangle$	26) $\langle 6, (5,7), \{\texttt{valid\_id}\}\rangle$
11) $\langle 1, (5,6), \{\texttt{valid\_id}\}\rangle$	27) $\langle 6, (5,6), \{\texttt{valid\_id}\}\rangle$
12) $\langle 1, (2,3), \{\texttt{achar}, \texttt{valid\_id}\}\rangle$	28) $\langle 7, (8,10), \{\texttt{achar}, \texttt{length}\}\rangle$
13) $\langle 1, (1,2), \{\texttt{achar}, \texttt{length}, \texttt{valid\_id}\}\rangle$	29) $\langle 7, (8,9), \{\texttt{achar}, \texttt{length}\}\rangle$
14) $\langle 2, (8,10), \{\texttt{length}\}\rangle$	30) $\langle 7, (5,7), \{\texttt{achar}, \texttt{length}\}\rangle$
15) $\langle 2, (8,9), \{\texttt{length}\}\rangle$	31) $\langle 7, (6,7), \{\texttt{achar}, \texttt{length}\}\rangle$
16) $\langle 2, (5,7), \{\texttt{length}\}\rangle$	32) $\langle 7, (5,6), \{\texttt{achar}, \texttt{length}\}\rangle$

Observe que somente com os casos de teste funcionais foi possível cobrir o critério Todos-Arcos ao passo que para 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 Todos-



(a) Todos-Arcos. (b) Todos-Potenciais-Usos.

Figura 5: Relatórios gerados pela ferramenta *PokeTool* em relação ao programa `identifier`.

Arcos-adequado, ou seja, o critério Todos-Arcos foi satisfeito e o erro presente no programa `identifier` não foi revelado. Certamente, um conjunto adequado ao critério Todos-Arcos que revelasse o erro poderia ter sido gerado; o que se ilustra aqui é que não necessariamente a presença do erro é revelada.

Desejando-se melhorar a cobertura em relação ao critério Todos-Potenciais-Usos, novos casos de teste devem ser inseridos visando a cobrir as associações que ainda não foram executadas. Primeiramente, deve-se verificar, entre as associações não executadas, se existem associações não executáveis. No caso, as associações $\langle 1, (8, 9), \{\texttt{length}, \texttt{valid\_id}\}\rangle$, $\langle 2, (8, 10), \{\texttt{length}\}\rangle$ e $\langle 6, (8, 9), \{\texttt{valid\_id}\}\rangle$ são não executáveis. Na Tabela 5 esse processo é ilustrado até que se atinja a cobertura de 100% para o critério Todos-Potenciais-Usos. O símbolo ✓ indica quais associações foram cobertas por quais conjuntos de casos de teste e o símbolo $\times$ mostra quais são as associações não-executáveis.

Tabela 5: Ilustração da evolução da sessão de teste para cobrir o critério Todos-Potenciais-Usos.

Associações Requeridas	T_0	T_1	T_2	**Associações Requeridas**	T_0	T_1	T_2
1) $\langle 1, (6, 7), \{\texttt{length}\}\rangle$		✓		17) $\langle 2, (6, 7), \{\texttt{length}\}\rangle$	✓		
2) $\langle 1, (1, 3), \{\texttt{achar}, \texttt{length}, \texttt{valid\_id}\}\rangle$	✓			18) $\langle 2, (5, 6), \{\texttt{length}\}\rangle$	✓		
3) $\langle 1, (8, 10), \{\texttt{length}, \texttt{valid\_id}\}\rangle$		✓		19) $\langle 3, (8, 10), \{\texttt{achar}\}\rangle$		✓	
4) $\langle 1, (8, 10), \{\texttt{valid\_id}\}\rangle$	✓			20) $\langle 3, (8, 9), \{\texttt{achar}\}\rangle$		✓	
5) $\langle 1, (8, 9), \{\texttt{length}, \texttt{valid\_id}\}\rangle$	×	×	×	21) $\langle 3, (5, 7), \{\texttt{achar}\}\rangle$	✓		
6) $\langle 1, (8, 9), \{\texttt{valid\_id}\}\rangle$	✓			22) $\langle 3, (6, 7), \{\texttt{achar}\}\rangle$	✓		
7) $\langle 1, (7, 4), \{\texttt{valid\_id}\}\rangle$	✓			23) $\langle 3, (5, 6), \{\texttt{achar}\}\rangle$	✓		
8) $\langle 1, (5, 7), \{\texttt{length}, \texttt{valid\_id}\}\rangle$	✓			24) $\langle 6, (8, 10), \{\texttt{valid\_id}\}\rangle$	✓		
9) $\langle 1, (5, 7), \{\texttt{valid\_id}\}\rangle$	✓			25) $\langle 6, (8, 9), \{\texttt{valid\_id}\}\rangle$	×	×	×
10) $\langle 1, (5, 6), \{\texttt{length}, \texttt{valid\_id}\}\rangle$		✓		26) $\langle 6, (5, 7), \{\texttt{valid\_id}\}\rangle$	✓		
11) $\langle 1, (5, 6), \{\texttt{valid\_id}\}\rangle$	✓			27) $\langle 6, (5, 6), \{\texttt{valid\_id}\}\rangle$			✓
12) $\langle 1, (2, 3), \{\texttt{achar}, \texttt{valid\_id}\}\rangle$	✓			28) $\langle 7, (8, 10), \{\texttt{achar}, \texttt{length}\}\rangle$	✓		
13) $\langle 1, (1, 2), \{\texttt{achar}, \texttt{length}, \texttt{valid\_id}\}\rangle$	✓			29) $\langle 7, (8, 9), \{\texttt{achar}, \texttt{length}\}\rangle$	✓		
14) $\langle 2, (8, 10), \{\texttt{length}\}\rangle$	×	×	×	30) $\langle 7, (5, 7), \{\texttt{achar}, \texttt{length}\}\rangle$	✓		
15) $\langle 2, (8, 9), \{\texttt{length}\}\rangle$		✓		31) $\langle 7, (6, 7), \{\texttt{achar}, \texttt{length}\}\rangle$			✓
16) $\langle 2, (5, 7), \{\texttt{length}\}\rangle$	✓			32) $\langle 7, (5, 6), \{\texttt{achar}, \texttt{length}\}\rangle$			✓

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)}

Observe que mesmo tendo satisfeito um critério mais rigoroso como o critério Todos-Potenciais-Usos, a presença do erro ainda não foi revelada. Assim, motiva-se a pesquisa de critérios de teste que exercitem os elementos requeridos com maior probabilidade de revelar erros. Outra perspectiva que se coloca é utilizar uma estratégia de teste incremental, que informalmente procura-se ilustrar neste texto. Em primeiro lugar foram exercitados os requisitos de teste requeridos pelo critério Todos-Arcos, em seguida os requeridos pelo critério Todos-Potenciais-Usos e, posteriormente, poder-se-ia considerar o critério Análise de Mutantes (descrito a seguir).

2.4 Técnica Baseada em Erros

A técnica de teste baseada em erros utiliza informações sobre os tipos de erros mais freqüentes no processo de desenvolvimento de software para derivar os requisitos de teste. A ênfase da técnica está nos erros que o programador ou projetista pode cometer durante o desenvolvimento e nas abordagens que podem ser usadas para detectar a sua ocorrência. Semeadura de Erros [10] e Análise de Mutantes [22] são critérios típicos que se concentram em erros.

No critério Semeadura de Erros, introduzido nos anos 80, uma quantidade conhecida de defeitos é semeada artificialmente no programa. Após o teste, do total de defeitos encontrados, verificam-se quais são naturais e quais são artificiais. Usando estimativas de probabilidade, o número de defeitos naturais ainda existentes no programa pode ser estimado. Entre os problemas associados à aplicação do critério destacam-se: (1) os defeitos artificiais podem interagir com os naturais fazendo com que os defeitos naturais sejam "mascarados" pelos defeitos semeados; (2) para obter um resultado estatístico não questionável é necessário o uso de programas capazes de conter 10.000 defeitos ou mais; (3) é preciso assumir que os defeitos estão uniformemente distribuídos pelo programa, o que, em geral, não é verdade. Programas reais apresentam longos trechos de código simples e com poucos defeitos, e pequenos trechos de grande complexidade e alta concentração de defeitos [10].

O critério Análise de Mutantes surgiu na década de 70 na *Yale University* e *Georgia Institute of Technology*, possuindo um forte relacionamento com um método clássico para detecção de erros lógicos em circuitos digitais – o modelo de teste de falha única [32]. Basicamente, o critério utiliza um conjunto de programas ligeiramente modificados (mutantes) obtidos a partir de determinado programa P para avaliar o quanto um conjunto de casos de teste T é adequado para o teste de P. O objetivo é determinar um conjunto de casos de teste que consiga revelar, por meio da execução de P, as diferenças de comportamento existentes entre P e seus mutantes [21].

A seguir é apresentada uma visão geral do critério Análise de Mutantes. Também são discutidos aspectos referentes à ferramenta de apoio *Proteum*, desenvolvida no ICMC/USP [15]. Informações detalhadas sobre o critério e sobre a ferramenta podem ser obtidas em [15, 60].

2.5 O Critério Análise de Mutantes

Um dos primeiros artigos que descrevem a idéia de teste de mutantes foi publicado em 1978 [22]. A idéia básica da técnica apresentada por DeMillo, conhecida como "hipótese do programador competente" (*competent programmer hypothesis*), assume que programadores experientes escrevem programas corretos ou muito próximos do correto. Assumindo a validade desta hipótese, pode-se afirmar que erros são introduzidos nos programas por meio de pequenos desvios sintáticos que, embora não causem erros sintáticos, alteram a semântica do programa e, conseqüentemente, conduzem-no a um comportamento incorreto. Para revelar tais erros, a Análise de Mutantes identifica os desvios sintáticos mais comuns e, através da aplicação de pequenas transformações sobre o programa em teste, encoraja o testador a construir casos de testes que mostrem que tais transformações levam a um programa incorreto [2].

Outra hipótese explorada na aplicação do critério Análise de Mutantes é o "efeito de acoplamento" (*coupling effect*) [22], a qual assume que erros complexos estão relacionados a erros simples. Assim sendo, espera-se, e alguns estudos empíricos já confirmaram esta hipótese [1, 9], que conjuntos de casos de teste capazes de revelar erros simples são também capazes de revelar erros complexos. Nesse sentido, aplica-se uma mutação de cada vez no programa P em teste, ou seja, cada mutante contém apenas uma transformação sintática. Um mutante com k transformações sintáticas é referenciado por k-mutante; neste texto são utilizados apenas 1-mutantes.

Partindo-se da hipótese do programador competente e do efeito de acoplamento, a princípio, o testador deve fornecer um programa P a ser testado e um conjunto de casos de teste T cuja adequação deseja-se avaliar. O programa é executado com T e se apresentar resultados incorretos então um erro foi encontrado e o teste termina. Caso contrário, o programa ainda pode conter erros que o conjunto T não conseguiu revelar. O programa P sofre então pequenas alterações, dando origem aos programas $P_1, P_2, \ldots, P_n$ denominados mutantes de P, diferindo de P apenas pela ocorrência de erros simples.

Com o objetivo de modelar os desvios sintáticos mais comuns, operadores de mutação (*mutant operators*) são aplicados a um programa P, transformando-o em programas similares: mutantes de P. Entende-se por operador de mutação as regras que definem as alterações que devem ser aplicadas no programa original P. Os operadores de mutação são construídos para satisfazer a um entre dois propósitos: (1) induzir mudanças sintáticas simples com base nos erros típicos cometidos pelos programadores (como trocar o nome de uma variável); ou (2) forçar determinados objetivos de teste (como executar cada arco do programa) [69].

Em seguida, os mutantes são executados com o mesmo conjunto de casos de teste T. O objetivo é obter casos de teste que resultem apenas em mutantes mortos (para algum caso de teste o resultado do mutante e o do programa original diferem entre si) e equivalentes (o mutante e o programa original apresentam sempre o mesmo resultado, para qualquer $d \in D$). Neste caso, tem-se um conjunto de casos de teste T adequado ao programa P em teste, no sentido de que, ou P está correto, ou possui erros pouco prováveis de ocorrerem [22].

É preciso ressaltar que, em geral, a equivalência entre programas é uma questão indecidível e requer a intervenção do testador. Essa limitação teórica, no entanto, não significa que o problema deva ser abandonado por não apresentar solução. Na verdade, alguns métodos e heurísticas têm sido propostos para determinar a equivalência de programas em uma grande porcentagem dos casos de interesse [10].

Um ponto importante destacado por DeMillo [20] é que a Análise de Mutantes fornece uma medida objetiva do nível de confiança da adequação dos casos de teste analisados por meio da definição de um escore de mutação (*mutation score*), que relaciona o número de mutantes mortos com o número de mutantes gerados. O escore de mutação é calculado da seguinte forma:

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

sendo:

$DM(P,T)$: número de mutantes mortos pelos casos de teste em T.

$M(P)$: número total de mutantes gerados.

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

O escore de mutação varia no intervalo entre 0 e 1 sendo que, quanto maior o escore mais adequado é o conjunto de casos de teste para o programa sendo testado. Percebe-se com essa fórmula

que apenas $DM(P,T)$ depende do conjunto de casos de teste utilizado e que, $EM(P)$ é obtido à medida que o testador, manualmente ou com o apoio de heurísticas, decide que determinado mutante vivo é equivalente [83].

Um dos maiores problemas para a aplicação do critério está relacionado ao seu alto custo, uma vez que o número de mutantes gerados, mesmo para pequenos programas, pode ser muito grande, exigindo um tempo de execução muito alto. Várias estratégias têm sido propostas para fazer com que a Análise de Mutantes possa ser utilizada de modo mais eficiente, dentro de limites economicamente viáveis. Uma solução bastante explorada pela comunidade de teste procura diminuir o custo de aplicação da Análise de Mutantes por meio da redução do número de mutantes a serem executados e analisados. Nessa perspectiva, algumas abordagens derivadas da Análise de Mutantes foram propostas: Mutação Aleatória (*Randomly Selected Mutation*) [1], Mutação Restrita (*Constrained Mutation*) [62] e Mutação Seletiva (*Selective Mutation*) [70]. Tais abordagens procuram selecionar apenas um subconjunto do total de mutantes gerados, reduzindo o custo associado, mas com a expectativa de não reduzir a eficácia do critério.

Assim como os critérios baseados em Fluxo de Dados, o critério Análise de Mutantes também tem sido essencialmente utilizado no teste de unidade. Na tentativa de estender sua aplicação para o teste de integração, Delamaro et al. [15, 16] propuseram o critério Mutação de Interface (*Interface Mutation*) – um critério para o teste de integração baseado no conceito de mutação, neste caso, mutação de interface entre os módulos componentes do software. A idéia básica é viabilizar o teste da interface entre as unidades que compõem o software, ao contrário da Análise de Mutantes, que explora somente as características das unidades separadamente [16, 17]. As abordagens de Mutação Aleatória, Mutação Restrita e Mutação Seletiva também foram estendidas de modo a permitir sua aplicação no teste de integração.

Outra linha de pesquisa investigada refere-se ao teste de especificações, com ênfase no teste e validação de aspectos comportamentais de sistemas reativos e validação de protocolos. Nesse sentido, extensões ao critério Análise de Mutantes têm sido propostas para o teste de especificações em Redes de Petri [28,81], Statecharts [23,85], Máquinas de Estado Finito [26,27], Estelle [74,84], SDL [86] e Especificações Algébricas [98].

Além disso, começam a aparecer na literatura extensões do teste de mutação para o teste de programas OO [18, 35, 50, 51, 56, 92, 93]. Tais extensões são descritas mais detalhadamente na Seção 3.

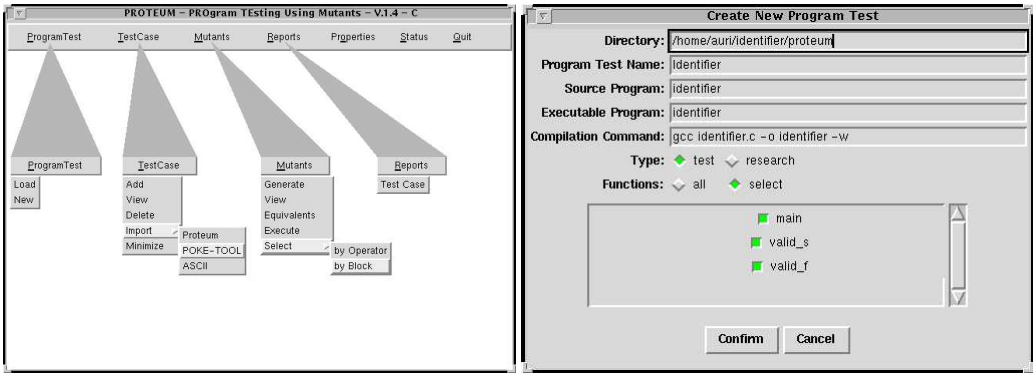
2.5.1 A Ferramenta de Teste *Proteum*

Para ilustrar os conceitos referentes ao teste de mutação será utilizada a ferramenta *Proteum* (*PROgram TEsting Using Mutants*) [15], desenvolvida no ICMC/USP. A ferramenta apóia a aplicação do critério Análise de Mutantes no teste de programas C, estando disponível para os sistemas operacionais SunOS, Solaris e Linux.

A Figura 6(a) apresenta a tela principal da ferramenta bem como as funções disponíveis. Basicamente, a *Proteum* oferece ao testador recursos para, através da aplicação do critério Análise de Mutantes, avaliar a adequação ou gerar um conjunto de casos de teste T para determinado programa P. Com base nas informações fornecidas pela *Proteum*, o testador pode melhorar a qualidade de T até obter um conjunto adequado ao critério. Desse modo, a ferramenta pode ser utilizada como instrumento de avaliação bem como de seleção de casos de teste.

A *Proteum* também trabalha com sessão de teste, ou seja, conjunto de atividades envolvendo um teste que podem ser realizadas em etapas, sendo possível ao usuário iniciar e encerrar o teste de um programa, bem como retomá-lo a partir de onde este foi interrompido. Uma sessão de teste com o apoio da ferramenta *Proteum* pode ser conduzida por meio de uma interface gráfica ou por

meio de *scripts*. O processo de criação de uma sessão de teste utilizando a interface gráfica é ilustrado na Figura 6(b).



(a) Opções disponíveis na ferramenta *Proteum*.

(b) Tela para criar uma sessão de teste.

Figura 6: Ferramenta *Proteum*.

Os recursos oferecidos pela ferramenta permitem a execução das seguintes operações: definição de casos de teste, execução do programa em teste, seleção dos operadores de mutação que serão utilizados para gerar os mutantes, geração dos mutantes, execução dos mutantes com os casos de teste definidos, análise dos mutantes vivos e cálculo do escore de mutação. As funções implementadas na *Proteum* possibilitam que alguns desses recursos sejam executados automaticamente (como a execução dos mutantes), enquanto que para outros são fornecidas facilidades para que o testador possa realizá-los (como a análise de mutantes equivalentes). Além disso, diversas características adicionais foram incorporadas de modo a facilitar a atividade de teste e/ou a condução de experimentos. É o caso, por exemplo, da possibilidade de executar um mutante com todos os casos de teste disponíveis, mesmo que algum deles já o tenha matado. Com esse tipo de teste, chamado *research*, conseguem-se dados a respeito da eficiência dos operadores de mutação ou mesmo para a determinação de estratégias de minimização dos conjuntos de casos de teste [15].

Um dos pontos essenciais para a aplicação do critério Análise de Mutantes é a definição do conjunto de operadores de mutação. A *Proteum* conta com 71 operadores de mutação divididos em quatro classes: mutação de comandos, mutação de operadores, mutação de variáveis e mutação de constantes. É possível escolher os operadores de acordo com a classe de erros que se deseja enfatizar, permitindo que a geração de mutantes seja feita em etapas ou até mesmo dividida entre vários testadores trabalhando independentemente. Na Tabela 6 são ilustrados alguns operadores de mutação para cada uma das classes de operadores.

A seguir, é avaliada a adequação do teste do programa `identifier`, realizado até este ponto com o uso da ferramenta *PokeTool*, em relação ao critério Análise de Mutantes, tendo como apoio a ferramenta *Proteum*. Em outras palavras, é avaliada a adequação dos conjuntos Todos-Usos-adequado e Todos-Potenciais-Usos-adequado em relação ao critério Análise de Mutantes.

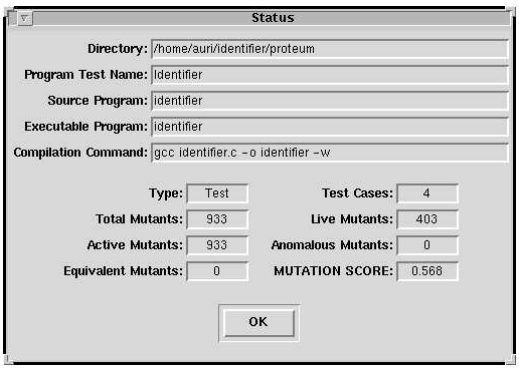
Inicialmente, somente os casos de teste do conjunto T_0 foram importados. A Figura 7(a) mostra o estado da sessão de teste após a execução dos mutantes. Em seguida, como o escore de mutação ainda não é satisfatório, foram adicionados os casos de teste do conjunto T_1 e T_2 (Figura 7(b)). Observe que mesmo após a adição de todos os casos de teste do conjunto Todos-Potenciais-Usos-adequado, 371 mutantes ainda permaneceram vivos.

Em uma primeira análise dos mutantes vivos, 78 foram marcados como equivalentes e mais 13 casos de teste foram criados visando a matar os mutantes vivos não-equivalentes: $T_3 = T_2 \cup$ {(zzz, Válido), (aA, Válido), (A1234, Válido), (ZZZ, Válido), (AAA, Válido), (aa09, Válido), ([, Invá-

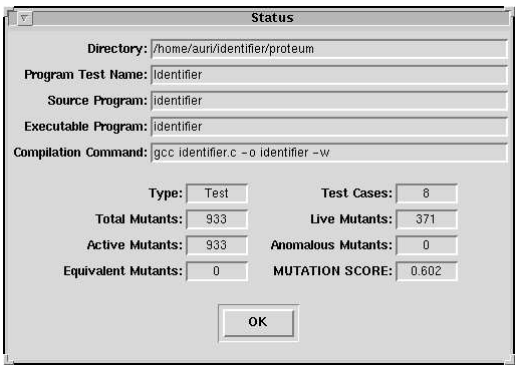
Tabela 6: Exemplos de operadores de mutação para programas C.

Operador	**Descrição**
SSDL	Retira um comando de cada vez do programa.
ORRN	Substitui um operador relacional por outro operador relacional.
VTWD	Substitui a referência escalar pelo seu valor sucessor e predecessor.
Ccsr	Substitui referências escalares por constantes.
SWDD	Substitui o comando *while* por *do-while*.
SMTC	Interrompe a execução do laço após duas execuções.
OLBN	Substitui operador lógico por operador *bitwise*.
Cccr	Substitui uma constante por outra constante.
VDTR	Força cada referência escalar a possuir cada um dos valores: negativo, positivo e zero.

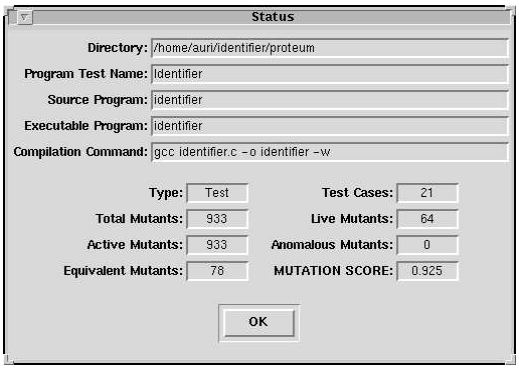
lido), ({, Inválido), (x/, Inválido), (x:, Inválido), (x18, Válido), (x[[, Inválido), (x{{, Inválido)}. A Figura 8 ilustra dois dos mutantes vivos que foram analisados. O mutante da Figura 8 (a) é um mutante equivalente e o mutante da Figura 8 (b) é um mutante que morre com o caso de teste ([, Inválido), presente em T_3. Os pontos nos quais as mutações foram aplicadas estão destacados em negrito. A Figura 7(c) ilustra o resultado obtido após T_3 ter sido executado com todos os mutantes vivos. Como pode ser observado, 64 mutantes ainda permaneceram vivos. Isto significa que qualquer um desses 64 mutantes poderiam ser considerados "corretos" em relação à atividade de teste atual, uma vez que não existe um caso de teste selecionado que seja capaz de distinguir entre o comportamento dos mutantes e do programa original (Figura 7(c)).



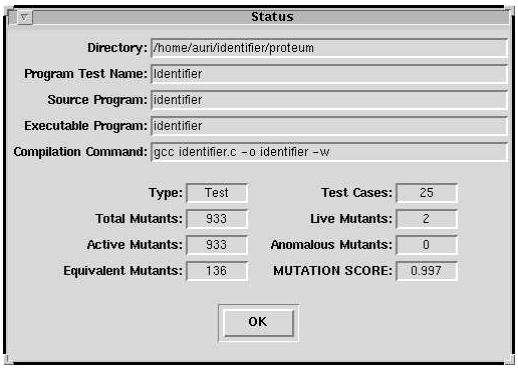
(a) Conjunto T_0.



(b) Conjuntos T_1 e T_2.

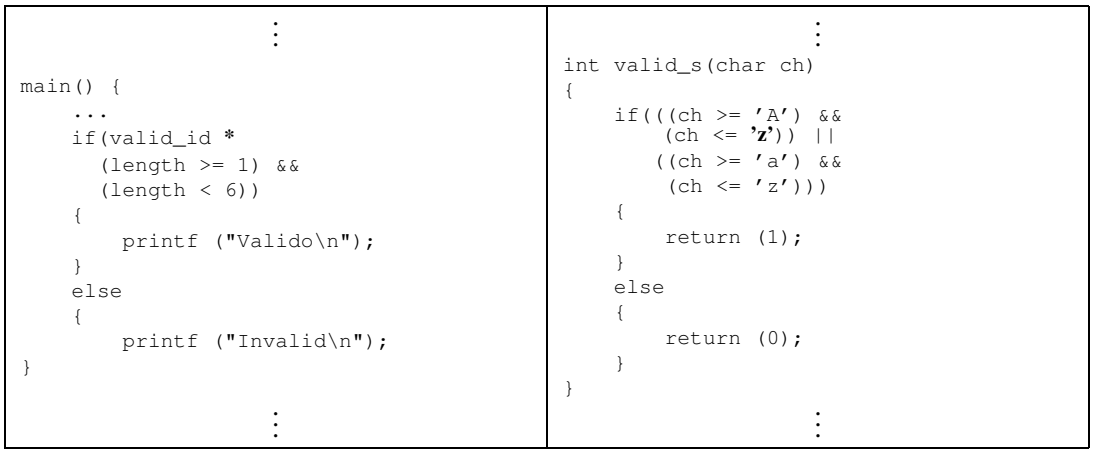


(c) Conjunto T_3.



(d) Conjuntos T_4.

Figura 7: Telas de *status* da sessão de teste da ferramenta *Proteum*.



```
                    ⋮

main() {
    ...
    if(valid_id *
      (length >= 1) &&
      (length < 6))
    {
        printf ("Valido\n");
    }
    else
    {
        printf ("Invalid\n");
}

                    ⋮
```

(a) Mutante equivalente.

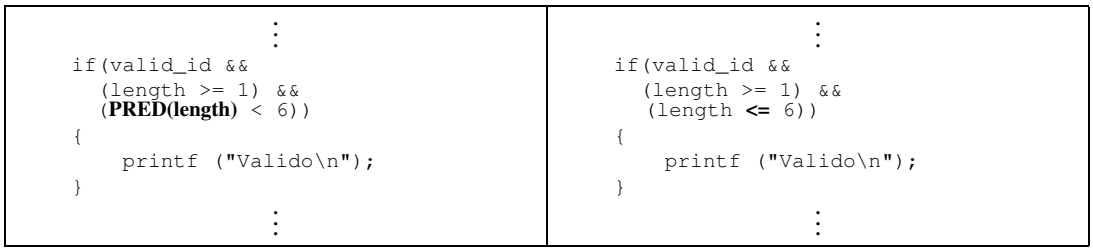
```
                    ⋮

int valid_s(char ch)
{
    if(((ch >= 'A') &&
        (ch <= 'z')) ||
       ((ch >= 'a') &&
        (ch <= 'z')))
    {
        return (1);
    }
    else
    {
        return (0);
    }
}

                    ⋮
```

(b) Mutante não-equivalente.

Figura 8: Exemplos de mutantes do programa `identifier`.



```
                    ⋮

    if(valid_id &&
      (length >= 1) &&
      (PRED(length) < 6))
    {
        printf ("Valido\n");
    }

                    ⋮
```

(a) Mutante *error-revealing*.

```
                    ⋮

    if(valid_id &&
      (length >= 1) &&
      (length <= 6))
    {
        printf ("Valido\n");
    }

                    ⋮
```

(b) Mutante correto.

Figura 9: Mutantes vivos do programa `identifier`.

A fim de obter uma melhor cobertura do critério Análise de Mutantes, o processo de análise dos mutantes vivos continuou até que todos os equivalentes fossem marcados. Ao término desse processo, mais quatro casos de teste foram construídos ($T_4 = T_3 \cup$ {(@, Inválido), (‘, Inválido), (x@, Inválido), (x‘, Inválido)}). A Figura 7(d) mostra o resultado final obtido. Observe que ainda restaram dois mutantes vivos (Figura 9 (a) e (b)). Esses mutantes são chamados *error-revealing* e um deles representa o programa correto: Figura 9 (b). Um mutante é dito ser *error-revealing* se para qualquer caso de teste t tal que $P^*(t) \neq M^*(t)$ pudermos concluir que $P^*(t)$ não está de acordo com o resultado esperado, ou seja, revela a presença de um erro.

Observe que os mutantes *error-revealing*, figuras 9(a) e 9(b), foram gerados pelos operadores de mutação ORRN e VTWD e que necessariamente o erro presente na versão do programa `identifier` é revelado ao se elaborar qualquer caso de teste que seja capaz de distinguir o comportamento entre esses mutantes e a versão do programa `identifier` em teste. Os mutantes da Figura 9 morrem, por exemplo, com o caso de teste (ABCDEF, Válido).

O erro encontrado no programa original foi corrigido e, após a sua correção, o conjunto completo de casos de teste T_5 foi reavaliado ($T_5 = T_4 \cup$ {(ABCDEF, Válido)}), resultando em um conjunto 100% adequado ao critério Análise de Mutantes para a versão corrigida do programa `identifier` (Figura 10). A parte corrigida está destacada em negrito.

Para o programa `identifier`, utilizando-se todos os operadores de mutação, foram gerados 933 mutantes. Aplicando-se somente os operadores da Tabela 6 teriam sido gerados somente 340 mutantes, representando uma economia de aproximadamente 63%. De fato, os operadores de mutação ilustrados nessa tabela constituem um conjunto de operadores essenciais para a linguagem C [5]. Ou seja, um conjunto de casos de teste capaz de distinguir os mutantes gerados por esses

```
/*****************************************************************************
Identifier.c
ESPECIFICACAO: O programa deve determinar se um identificador eh ou nao valido em
'Silly Pascal' (uma estranha variante do Pascal).  Um identificador valido deve
comecar com uma letra e conter apenas letras ou digitos.  Alem disso, deve ter no
minimo 1 caractere e no maximo 6 caracteres de comprimento
*****************************************************************************/

            #include <stdio.h>
            main ()
 /*  1 */   {
 /*  1 */       char  achar;
 /*  1 */       int length, valid_id;
 /*  1 */       length = 0;
 /*  1 */       valid_id = 1;
 /*  1 */       printf ("Identificador: ");
 /*  1 */       achar = fgetc (stdin);
 /*  1 */       valid_id = valid_s(achar);
 /*  1 */       if(valid_id)
 /*  2 */       {
 /*  2 */           length = 1;
 /*  2 */       }
 /*  3 */       achar = fgetc (stdin);
 /*  4 */       while(achar != '\n')
 /*  5 */       {
 /*  5 */           if(!(valid_f(achar)))
 /*  6 */           {
 /*  6 */               valid_id = 0;
 /*  6 */           }
 /*  7 */           length++;
 /*  7 */           achar = fgetc (stdin);
 /*  7 */       }
 /*  8 */       if(valid_id &&
 /*  9 */       {   (length >= 1) && (length <= 6))
 /*  9 */           printf ("Valido\n");
 /*  9 */       }
 /* 10 */       else
 /* 10 */       {
 /* 10 */           printf ("Invalid\n");
 /* 10 */       }
 /* 11 */   }


            int valid_s(char ch)
/* 1 */     {
/* 1 */        if(((ch >= 'A') &&
                   (ch <= 'Z')) ||
                  ((ch >= 'a') &&
                   (ch <= 'z')))
/* 2 */        {
/* 2 */            return (1);
/* 2 */        }
/* 3 */        else
/* 3 */        {
/* 3 */            return (0);
/* 3 */        }
/* 4 */     }

            int valid_f(char ch)
/* 1 */     {
/* 1 */         if(((ch >= 'A') &&
                    (ch <= 'Z')) ||
                   ((ch >= 'a') &&
                    (ch <= 'z')) ||
                   ((ch >= '0') &&
                    (ch <= '9')))
/* 2 */         {
/* 2 */             return (1);
/* 2 */         }
/* 3 */         else
/* 3 */         {
/* 3 */             return (0);
/* 3 */         }
/* 4 */     }
```

Figura 10: Versão do programa `identifier` corrigida.

operadores, em geral, seria capaz de distinguir os mutantes não equivalentes gerados pelos demais operadores de mutação, determinando um escore de mutação bem próximo de 1. Observe que os operadores de mutação ORRN e VTWD, que geraram os mutantes *error-revealing*, estão entre os operadores essenciais, o que neste caso, não comprometeria a eficácia da atividade de teste.

3 Aplicação de Critérios de Teste no Contexto de Programas OO

Nas seções anteriores foram discutidos conceitos e aspectos básicos relacionados ao teste de programas procedimentais. A seguir, são abordados alguns dos principais aspectos e direções de pesquisa na área de teste de programas OO, procurando-se mostrar que os conceitos e mecanismos desenvolvidos originalmente para o teste de programas procedimentais também podem ser utilizados nesse contexto de desenvolvimento, com as devidas adaptações. Inicialmente são identificadas as fases de teste para programas OO, as quais são comparadas às fases de teste para programas procedimentais. O impacto das características de encapsulamento, herança, polimorfismo e acoplamento dinâmico na definição e aplicação de critérios de teste também é discutido. Além disso,

é dada uma visão geral dos principais trabalhos relacionados ao teste baseado em Fluxo de Dados e ao teste de Mutação para programas OO. Algumas iniciativas de automatização no contexto de teste OO, em especial a ferramenta *JaBUTi*, também são brevemente discutidas.

3.1 Fases de Teste OO

Assim como os métodos de desenvolvimento de software são divididos em várias fases de modo a permitir que o engenheiro de sistemas implemente a solução do problema passo a passo, a atividade de teste também é dividida em fases. Com isso, o testador pode se concentrar em diferentes aspectos do software e utilizar diferentes critérios de teste em cada uma delas [55]. Conforme discutido na Seção 1, em nível procedimental, a atividade de teste pode ser considerada como uma atividade incremental realizada em três fases [73]: teste de unidade, teste de integração e teste de sistema. No contexto de programas OO, entretanto, algumas variações são identificadas, conforme apresentado a seguir.

Na Figura 11, adaptada de [8], são ilustradas as três fases de teste mencionadas acima, bem como os elementos utilizados em cada uma das fases tanto para programas procedimentais como para programas OO. Segundo o padrão IEEE 610.12-1990 [49], uma unidade é um componente de software que não pode ser subdividido. Considerando que teste é uma atividade dinâmica, em programas procedimentais uma unidade F refere-se a uma sub-rotina ou um procedimento que é a menor parte funcional de um programa que pode ser executada. Observa-se, ainda, 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, no momento 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.

Com base em tais definições, pode-se considerar que em programas OO a menor unidade a ser testada é um método. A classe à qual o método pertence pode ser vista como o *driver* do método, pois sem a classe não é possível executar um método. No paradigma procedimental o teste de unidade também é chamado de intraprocedimental; no paradigma OO é dito intra-método [41].

Por definição, uma classe engloba um conjunto de atributos e métodos que manipulam esses atributos. Assim sendo, considerando uma única classe já é possível pensar em teste de integração. Métodos da mesma classe podem interagir entre si para desempenhar funções específicas caracterizando uma integração entre métodos que deve ser testada: é o teste inter-método [41]. No paradigma procedimental essa fase de teste também pode ser chamada de teste interprocedimental.

Harrold e Rothermel [41] definem ainda outros dois tipos de teste para programas OO: teste intra-classe e teste inter-classe. No teste intra-classe são testadas interações entre métodos públicos fazendo chamada a esses métodos em diferentes seqüências. O objetivo é identificar possíveis seqüências de ativação de métodos inválidas que levem o objeto a um estado inconsistente. Segundo os autores, como o usuário pode invocar seqüências de métodos públicos em uma ordem indeterminada, o teste intra-classe aumenta a confiança de que diferentes seqüências de chamadas interagem adequadamente. No teste inter-classe 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 serem baseados em critérios funcionais, não apresentam diferenças fundamentais entre o teste procedimental e OO.

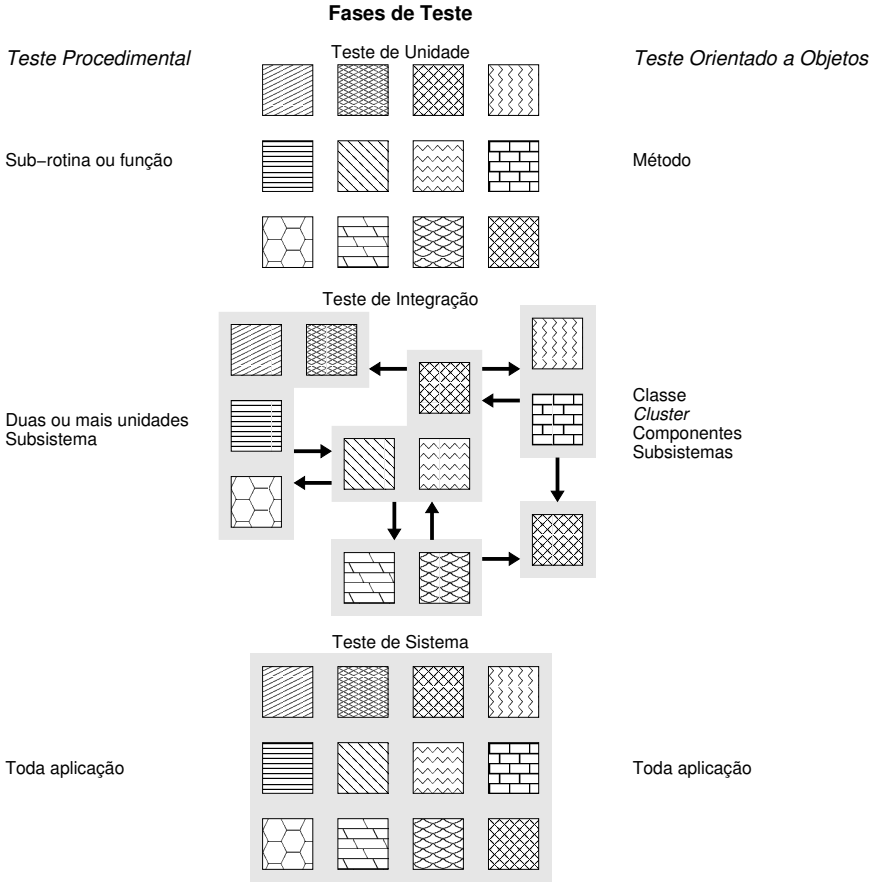


Figura 11: Relacionamento entre teste de unidade, de integração e de sistema: programas procedimentais e OO [8].

Pequenas variações quanto à divisão das fases de teste para programas OO são identificadas na literatura. Alguns autores entendem que a menor unidade de um programa OO é a classe e não o método [3, 8, 63, 72]. Nessa direção, o teste de unidade poderia envolver o teste intra-método, inter-método e intra-classe, enquanto o teste de integração corresponderia ao teste inter-classe. Na Tabela 7 são sintetizados os tipos de teste que podem ser aplicados em cada uma das fases, tanto em programas procedimentais quanto em programas OO, considerando o método ou a classe como sendo a menor unidade.

Tabela 7: Relação entre fases de teste de programas procedimentais e OO.

Menor Unidade: Método		
Fase	Teste Procedimental	Teste Orientado a Objetos
Unidade	Intraprocedimental	Intra-método
Integração	Interprocedimental	Inter-método, Intra-classe e Inter-classe
Sistema	Toda Aplicação	Toda Aplicação
Menor Unidade: Classe		
Fase	Teste Procedimental	Teste Orientado a Objetos
Unidade	Intraprocedimental	Intra-método, Inter-método e Intra-classe
Integração	Interprocedimental	Inter-classe
Sistema	Toda Aplicação	Toda Aplicação

3.2 Impacto da Orientação a Objetos na Testabilidade

O paradigma de programação OO possui um conjunto de construções que, apesar de poderosas, apresentam riscos de erros e problemas de teste. Nesta seção é discutido o quanto características como encapsulamento, herança, polimorfismo e acoplamento dinâmico podem impactar na condução da atividade de teste.

3.2.1 Encapsulamento

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, 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 em desenvolvimento.

Embora o encapsulamento não contribua diretamente para a ocorrência de erros, ele pode constituir um obstáculo para a atividade de teste, limitando a controlabilidade e observabilidade. Teste requer um relatório completo do estado concreto e abstrato de um objeto, bem como a possibilidade de alterar esse estado facilmente [8]. As linguagens OO dificultam a atividade de se obter (*get*) ou alterar (*set*) o estado de um objeto. No caso específico de C++, as funções amigas (*friend functions*) foram desenvolvidas para solucionar esses problemas. Entretanto, no caso de linguagens que não possuem esse recurso, outras providências devem ser tomadas. Harrold [38], referindo-se ao 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. No entanto, como destacam Rosa e Martins [77], algumas linguagens não permitem 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 no trabalho desenvolvido pelas mesmas [77].

3.2.2 Herança

Herança é essencial à programação OO pois permite a reusabilidade via o compartilhamento de características presentes em uma classe já definida anteriormente. Entretanto, como destacado por Binder [8], a herança enfraquece o encapsulamento e pode ser responsável pela criação de um risco de erro 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, pode-se desenvolver classes que aparentemente funcionam corretamente, mas violam condições implícitas requeridas para garantir a correção das classes ancestrais. Grandes encadeamentos de herança podem dificultar a compreensão, aumentar a chance de ocorrência de erros e reduzir a testabilidade das classes.

Como comentado por Offutt e Irvine [68], a utilização de herança pode levar a uma falsa conclusão de que subclasses que herdam características de superclasses não precisam ser testadas, reduzindo assim o esforço com os testes. Perry e Kaiser [72] observam que mesmo que um método seja herdado integralmente de uma superclasse, sem nenhuma modificação, este deverá ser retestado no contexto da subclasse. Harrold et al. [40] utilizaram os resultados de Perry e Kaiser [72] e desenvolveram uma estratégia de teste incremental baseada na hierarquia de herança das classes – Estratégia Incremental Hierárquica. A idéia é identificar quais métodos herdados necessitam de novos casos de teste para serem testados e quais métodos podem ser retestados aproveitando os casos de teste elaborados para o teste da superclasse. Com essa estratégia, o esforço requerido

para o teste é reduzido, visto que muitos casos de teste que já foram elaborados podem ser reutilizados no teste das subclasses. Além disso, deve-se observar que a implementação do mecanismo de herança varia de linguagem para linguagem, influenciando a estratégia de teste a ser utilizada.

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 [72] destacam que, embora herança múltipla leve a pequenas mudanças sintáticas, ela pode levar a grandes mudanças semânticas, dificultando ainda mais a realização dos testes.

3.2.3 Polimorfismo

Polimorfismo refere-se à capacidade de se fazer referência a mais de um tipo de objeto por meio de um mesmo nome ou variável. No polimorfismo estático essa associação ocorre em tempo de compilação. Por exemplo, classes genéricas (*templates* do C++) permitem a realização de polimorfismo estático. Já o polimorfismo dinâmico permite que, em tempo de execução, associações com diferentes tipos de objetos sejam realizadas. Métodos polimórficos utilizam o recurso de acoplamento dinâmico para determinar, em tempo de execução, qual método deve responder a uma determinada mensagem, baseado no tipo do objeto e no conjunto de parâmetros que são enviados junto com a mensagem.

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, o qual precisa ser testado. Posteriormente, o método x é sobrescrito. A correção do método x na subclasse não é garantida pois as pré-condições e pós-condições na subclasse para a execução do método x podem não ser as mesmas da superclasse [8].

Cada possibilidade de acoplamento de uma mensagem polimórfica é uma computação única. O fato de diversos acoplamentos polimórficos trabalharem corretamente não garante que todos irão trabalhar. 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 associações [93].

3.2.4 Acoplamento Dinâmico

O acoplamento dinâmico faz com que, em tempo de execução, uma mensagem seja enviada para uma classe servidora que implemente aquela mensagem. Como classes servidoras são freqüentemente desenvolvidas e revisadas sem levar em consideração as classes clientes, a utilização de métodos que antes funcionavam adequadamente na classe cliente pode levar a resultados inesperados. Uma classe cliente pode solicitar um método que não mais está implementado na classe servidora, usar incorretamente os métodos disponíveis ou invocar os métodos com parâmetros incorretos [93].

Além dos problemas apresentados acima, Binder [8] descreve ainda erros relacionados com seqüências de mensagens e estados dos objetos. O empacotamento de métodos dentro de uma classe é fundamental em 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, ocupando espaço na memória da máquina. Cada nova configuração assumida por esse espaço de memória caracteriza um novo estado do objeto. Assim, além do comportamento encapsulado por um objeto por meio de seus métodos e atributos, objetos também encapsulam estados.

Examinando como a execução de um método pode alterar o estado de um objeto, quatro possibilidades são observadas [63]: (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; e (4) ele pode alterar o estado para um estado não apropriado. A terceira e a quarta opções caracterizam estados de erro. A primeira opção pode caracterizar um erro se o método executado comportar-se como na segunda opção, e vice-versa.

3.3 Critérios de Teste OO

Conforme discutido na Seção 2, técnicas e critérios de teste têm sido investigados visando a fornecer uma maneira sistemática e rigorosa para selecionar um subconjunto do domínio de entrada e, ainda assim, ser eficaz para revelar a presença dos erros existentes, respeitando as restrições de tempo e custo associados a um projeto de software. Nesta seção são apresentados alguns dos principais critérios de teste para programas OO identificados na literatura. São brevemente discutidos critérios funcionais, baseados em estados, estruturais e baseados em erros.

Critérios funcionais, conforme observado anteriormente, podem ser aplicados diretamente tanto no teste de programas procedimentais como no teste de programas OO, visto que derivam seus requisitos de teste somente com base na especificação do programa. Visando a avaliar a adequação de um critério funcional utilizado no teste procedimental para revelar defeitos em programas OO, Offutt e Irvine [68] investigaram a utilização do Método de Partição-Categoria. O método oferece um procedimento que é utilizado pelo testador para produzir casos de teste a partir da especificação. Em linhas gerais, o trabalho do testador é definir categorias que representem as principais características do domínio de entrada da função sendo testada e particionar cada categoria em classes de equivalência de entradas, chamadas *choices*. Por definição, as *choices* dentro de uma categoria devem ser disjuntas e, quando unidas, devem cobrir todo o domínio de entrada de dada categoria. O estudo de caso investigando a eficácia do Método Partição-Categoria na detecção de falhas em programas OO é descrito em detalhes em [68].

Os critérios baseados em estado são bastante utilizados no contexto de OO para representar o aspecto comportamental dos objetos [7, 8, 45, 63, 64, 87]. Segundo Binder, o grande desafio do teste de software OO é projetar conjuntos de casos de teste que exercitem combinações de seqüências de mensagens e interações de estados dando confiança de que o software funciona corretamente [7]. Em algumas situações, casos de teste baseados em seqüência de mensagens ou estados são suficientes. Entretanto, o teste baseado em estados não é capaz de detectar todos os tipos de defeitos, exigindo que critérios de teste baseados em programa sejam utilizados, visando a maximizar a detecção de defeitos [8]. Métodos de uma mesma classe têm acesso às mesmas variáveis de instância e devem cooperar entre si para o correto funcionamento da classe, considerando todas as seqüências de ativação possíveis. A visibilidade das variáveis de instância para todos os métodos da classe cria um risco de erro semelhante ao uso de variáveis globais nas linguagens de programação procedimentais. Dado que os métodos da superclasse não estão explícitos quando uma subclasse é codificada, isso pode resultar no uso inconsistente das variáveis de instância. Para revelar esse tipo de defeito é necessária a utilização de critérios de teste de Fluxo de Controle e de Fluxo de Dados que garantam a cobertura inter-método (ou intra-classe).

No caso de critérios estruturais, um dos principais trabalhos foi desenvolvido por Harrold e Rothermel [41], que estenderam o teste de Fluxo de Dados para o teste de classes. Os autores comentam que os critérios baseados em Fluxo de Dados destinados ao teste de programas procedimentais 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 [42, 76]. Entretanto, tais critérios não consideram interações de fluxo de dados quando os usuários de uma classe invocam seqüência de métodos em uma ordem arbitrária. Para resolver esse problema, os autores apresentam

uma abordagem que permite testar diferentes tipos de interações de fluxo de dados entre classes. A abordagem proposta usa as técnicas tradicionais de fluxo de dados para testar os métodos individuais e as interações entre os métodos dentro de mesma classe. Para testar os métodos que são acessíveis fora da classe e podem ser utilizados por outras classes, uma nova representação, denominada Grafo de Fluxo de Controle de Classe (CCFG – *Class Control Flow Graph*), foi desenvolvida. A partir dessa representação novas associações inter-método e intra-classe podem ser derivadas.

Vincenzi et al. [92, 93] também têm investigado o estabelecimento de critérios estruturais para o teste de programas OO e teste de componentes. O objetivo principal é definir/adaptar critérios estruturais tradicionais, tais como os critérios de fluxo de controle Todos-Nós e Todos-Arcos, e os critérios de fluxo de dados Todos-Usos e Todos-Potenciais-Usos, para o teste de unidade (intra-método) de programas OO e componentes de software.

Para avaliar a aplicabilidade dos critérios de teste definidos por Vincenzi [92] optou-se pela linguagem Java. Tal escolha foi feita, sobretudo, em função da grande gama de aplicações e componentes que vêm sendo desenvolvidos utilizando essa linguagem. Mais precisamente, a idéia é viabilizar o teste estrutural de programas Java a partir do bytecode[5] Java, permitindo, com isso, também o teste estrutural de componentes para os quais os códigos-fonte nem sempre se encontram disponíveis.

Após a escolha da linguagem alvo, um modelo de fluxo de dados subjacente é definido, caracterizando as instruções de bytecode responsáveis pela definição e/ou o uso de variáveis. De posse de modelo de fluxo de dados, um modelo de representação de programa – o Grafo Definição-Uso (*DU*) – é construído, considerando os mecanismos de tratamento de exceção, em geral, presentes nas linguagens OO. Desse modo, o grafo *DU* é utilizado para representar o fluxo de controle e o fluxo de dados intra-método, tanto durante a execução normal do programa quanto na presença de exceções. Uma vez que o grafo *DU* de cada método tenha sido obtido, critérios de teste podem ser definidos para derivar diferentes requisitos de teste, os quais podem ser utilizados tanto para avaliar a qualidade de um determinado conjunto de teste quanto para a própria geração de dados de teste [92].

Ao todo, oito critérios de teste estruturais foram definidos. Vincenzi [92] optou por separar os requisitos de teste em dois conjuntos disjuntos: (1) os que podem ser cobertos durante a execução normal do programa, denominados independentes de exceção; e (2) os que para serem cobertos exigem, obrigatoriamente, que uma exceção tenha sido gerada, denominados dependentes de exceção. Desse modo, foram estabelecidos os seguintes critérios:

- Todos-Nós: Todos-Nós-Independentes-de-Exceção ($\texttt{Todos-Nós}_{ei}$) e Todos-Nós-Dependentes-de-Exceção ($\texttt{Todos-Nós}_{ed}$);
- Todas-Arestas: Todas-Arestas-Independentes-de-Exceção ($\texttt{Todas-Arestas}_{ei}$) e Todas-Arestas-Dependentes-de-Exceção ($\texttt{Todas-Arestas}_{ed}$);
- Todos-Usos: Todos-Usos-Independentes-de-Exceção ($\texttt{Todos-Usos}_{ei}$) e Todos-Usos-Dependentes-de-Exceção ($\texttt{Todos-Usos}_{ed}$); e
- Todos-Potenciais-Usos: Todos-Potenciais-Usos-Independentes-de-Exceção ($\texttt{Todos-Pot-Usos}_{ei}$) e Todos-Potenciais-Usos-Dependentes-de-Exceção ($\texttt{Todos-Pot-Usos}_{ed}$).

[5] 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.

Um exemplo ilustrando a aplicação dos critérios estruturais definidos por Vincenzi [92] para o teste intra-método, utilizando-se a ferramenta *JaBUTi* [92, 94], é apresentado na próxima seção.

Considerando os critérios baseados em erros, um ponto importante a ser destacado é a flexibilidade de estender os conceitos do teste de mutação a diversas "entidades executáveis". Conforme discutido na Seção 2, o teste de mutação, desenvolvido inicialmente para o teste de unidade de programas procedimentais, já foi estendido para o teste de integração de programas procedimentais [15,16] e para o teste de especificações baseadas em Máquinas de Estado Finito [26,27], Redes de Petri [28, 80], Statecharts [23, 85], Estelle [74, 84], SDL [86] e Especificações Algébricas [98].

Especificamente no que diz respeito ao teste de programas OO, o teste de mutação vem sendo utilizado para o teste de aspectos referentes a concorrência, comunicação entre processos e teste de programas Java e C++ em nível de unidade e de integração: (1) Kim et al. [50] utilizaram uma técnica denominada HAZOP (*Hazzard and Operability Studies*) para a definição de um conjunto de operadores de mutação para o teste de programas Java; (2) Ma et al. [56] propuseram um conjunto mais abrangente de operadores de mutação para o teste de programas Java, os quais incluem o conjunto de operadores definidos por Kim et al. [50]; (3) Gosh e Mathur [35] definiram um conjunto de operadores de mutação visando ao teste de interfaces de comunicação entre componentes distribuídos (CORBA); (4) Delamaro et al. [18] definiram operadores específicos para o teste de programas concorrentes implementados em Java; e (5) Vincenzi [92] definiu três conjuntos distintos de operadores de mutação os quais podem ser utilizados no teste intra-método, inter-método e inter-classe, considerando as linguagens Java e C++.

Finalmente, é importante ressaltar que além dos critérios de teste descritos nesta seção, outros exemplos podem ser encontrados na literatura, dentre eles os trabalhos de Rosenblum [78] e Harrold et al. [39]. Além disso, destaca-se a importância da realização de estudos teóricos e empíricos procurando avaliar e comparar os diversos critérios de teste OO existentes [92].

3.3.1 A Ferramenta *JaBUTi*

A ferramenta *JaBUTi* (***Ja**va **B**ytecode **U**nderstanding and **T**est**i**ng*) [92, 94], desenvolvida no ICMC/USP em colaboração com a UNIVEM/Marília, visa a ser um ambiente completo para o entendimento e teste de programas e componentes Java. A idéia básica da ferramenta é viabilizar o teste de programas Java em nível de bytecode, possibilitando, com isso, não somente o teste de programas Java para os quais o código-fonte esteja disponível, mas também o teste de componentes Java.

JaBUTi fornece ao testador diferentes critérios de teste estruturais para a análise de cobertura, um conjunto de métricas estáticas para se avaliar a complexidade das classes que compõem do programa/componente, e implementa, ainda, algumas heurísticas de particionamento de programas que visam a auxiliar a localização de defeitos. Neste texto, é dada ênfase à parte responsável pela análise de cobertura. Mais informações sobre as demais funcionalidades da ferramenta podem ser obtidas em [92, 94].

Considerando o suporte à análise de cobertura de programas Java, a ferramenta implementa atualmente seis dos oito critérios de teste intra-métodos definidos por Vincenzi [92], sendo quatro critérios de Fluxo de Controle (`Todos-Nós`$_{ei}$, `Todos-Nós`$_{ed}$, `Todas-Arestas`$_{ei}$,`Todas-Arestas`$_{ed}$) e dois critérios de Fluxo de Dados (`Todos-Usos`$_{ei}$ e `Todos-Usos`$_{ed}$). Os critérios `Todos-Pot-Usos`$_{ei}$ e `Todos-Pot-Usos`$_{ed}$ ainda estão fase de implementação. Como descrito na Seção 3, os pares de critérios `Todos-Nós`$_{ei}$, `Todos-Nós`$_{ed}$ e `Todas-Arestas`$_{ei}$, `Todas-Arestas`$_{ed}$ compõem os critérios `Todos-Nós` e `Todos-Arcos`, respectivamente. Da mesma forma, os pares de critérios `Todos-Usos`$_{ei}$, `Todos-Usos`$_{ed}$ e `Todos-Pot-Usos`$_{ei}$, `Todos-Pot-Usos`$_{ed}$ compõem os critérios `Todos-Usos` e `Todos-Pot-Usos`, respectivamente.

Para ilustrar os aspectos operacionais da *JaBUTi*, um exemplo simples, adaptado de Orso et al. [71], é utilizado. O exemplo implementa o comportamento de uma máquina de venda (*vending machine*) típica e é composto de duas classes: uma que implementa um componente `Dispenser` e outra, `VendingMachine`, que utiliza o componente `Dispenser`. O código-fonte em Java de ambas as classes é apresentado na Figura 12.

```
/*01*/ package vending;
/*02*/
/*03*/ public class VendingMachine {
/*04*/
/*05*/    final private int COIN = 25;
/*06*/    final private int VALUE = 50;
/*07*/    private int totValue;
/*08*/    private int currValue;
/*09*/    private Dispenser d;
/*10*/
/*11*/    public VendingMachine() {
/*12*/       totValue = 0;
/*13*/       currValue = 0;
/*14*/       d = new Dispenser();
/*15*/    }
/*16*/
/*17*/    public void insertCoin() {
/*18*/       currValue += COIN;
/*19*/       System.out.println("Current value = " + currValue);
/*20*/    }
/*21*/
/*22*/    public void returnCoin() {
/*23*/       if (currValue == 0)
/*24*/          System.err.println("No coins to return");
/*25*/       else {
/*26*/          System.out.println("Take your coins");
/*27*/          currValue = 0;
/*28*/       }
/*29*/    }
/*30*/
/*31*/    public void vendItem(int selection) {
/*32*/       int expense;
/*33*/
/*34*/       expense = d.dispense(currValue, selection);
/*35*/       totValue += expense;
/*36*/       currValue -= expense;
/*37*/       System.out.println("Current value = " + currValue);
/*38*/    }
/*39*/ } // class VendingMachine
```

```
/*01*/ package vending;
/*02*/
/*03*/ public class Dispenser {
/*04*/    final private int MINSEL = 1;
/*05*/    final private int MAXSEL = 20;
/*06*/    final private int VAL = 50;
/*07*/
/*08*/    private int[] valSel =
/*09*/          { 1, 2, 3, 4, 6, 7, 8, 9, 10,
/*10*/             11, 12, 13, 14, 15, 16, 17, 19};
/*11*/
/*12*/    public int dispense(int credit, int sel) {
/*13*/       int val = 0;
/*14*/
/*15*/       if (credit == 0)
/*16*/          System.err.println("No coins inserted");
/*17*/       else if ((sel < MINSEL) || (sel > MAXSEL))
/*18*/          System.err.println("Wrong selection " + sel);
/*19*/       else if (!available(sel))
/*20*/          System.err.println("Selection " + sel + " unavailable");
/*21*/       else {
/*22*/          val = VAL;
/*23*/          if (credit < val) {
/*24*/             System.err.println("Enter " + (val – credit) + " coins");
/*25*/          } else
/*26*/             System.out.println("Take selection");
/*27*/       }
/*28*/       return val;
/*29*/    }
/*30*/
/*31*/    private boolean available(int sel) {
/*32*/       boolean ans = false;
/*33*/       try {
/*34*/          for (int i = 0; i < valSel.length && !ans; i++)
/*35*/             if (valSel[i] == sel)
/*36*/                ans = true;
/*37*/       } catch (NullPointerException npe) {
/*38*/          ans = false;
/*39*/       }
/*40*/       return ans;
/*41*/    }
/*42*/
/*43*/    public void setValSel(int[] v) {
/*44*/       valSel = v;
/*45*/    }
/*46*/ } // class Dispenser
```

Figura 12: Exemplo de uma aplicação Java (`VendingMachine`) e um componente (`Dispenser`) [71].

O componente `Dispenser` é responsável por manter as informações sobre o preço de cada item e quais deles são válidos e estão disponíveis. O método mais importante da classe `Dispenser` é o método `Dispenser.dispense()`, o qual é responsável por receber como parâmetros a quantia em dinheiro depositada na máquina e o número do item selecionado pelo usuário, e decidir se o item pode ou não ser entregue ao usuário. Tal método realiza os seguintes passos:

1. Verifica se pelo menos uma moeda foi depositada na máquina;

2. Verifica se um item válido foi selecionado;

3. Verifica se o item válido encontra-se disponível (para isso utiliza o método `Dispenser.available()`);

4. Verifica se o valor fornecido é suficiente para comprar o item válido e disponível selecionado.

Se todos os pré-requisitos acima forem satisfeitos, o componente `Dispenser` entrega o item desejado pelo usuário. Do contrário, caso alguma das condições acima não seja satisfeita, uma mensagem de erro é emitida e nenhum item é entregue ao usuário.

Considerando a utilização da ferramenta *JaBUTi* via interface gráfica, o primeiro passo para conduzir uma atividade de teste é a criação de um projeto de teste, o qual contém informações sobre as classes a serem testadas. Para a criação do projeto de teste o testador deve, primeiramente, fornecer o nome de uma classe base, ou seja, o nome de uma classe tipo aplicação a partir da qual as demais classes relacionadas serão derivadas. Fornecido o nome da classe base, a ferramenta exibe a janela do Gerenciador de Projeto (*Project Manager*), como ilustrado na Figura 13. Do lado esquerdo dessa janela encontra-se o conjunto completo das classes que foram identificadas a partir da classe base e que podem ser selecionadas para serem testadas. No exemplo, duas foram as classes selecionadas – `VendingMachine` e `Dispenser`.

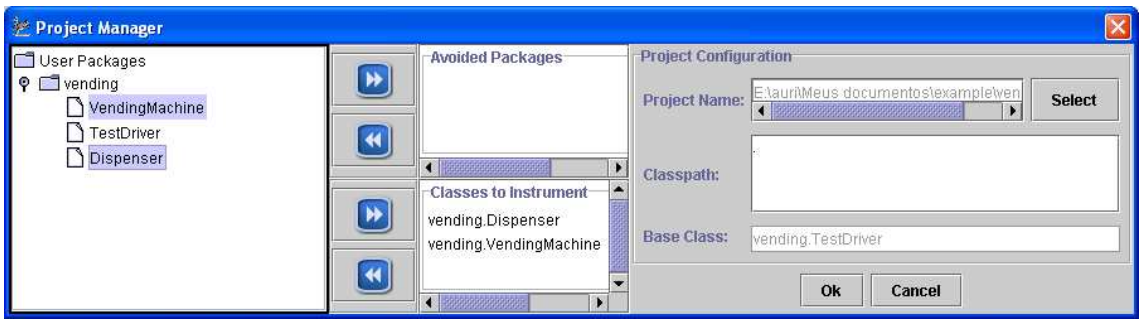
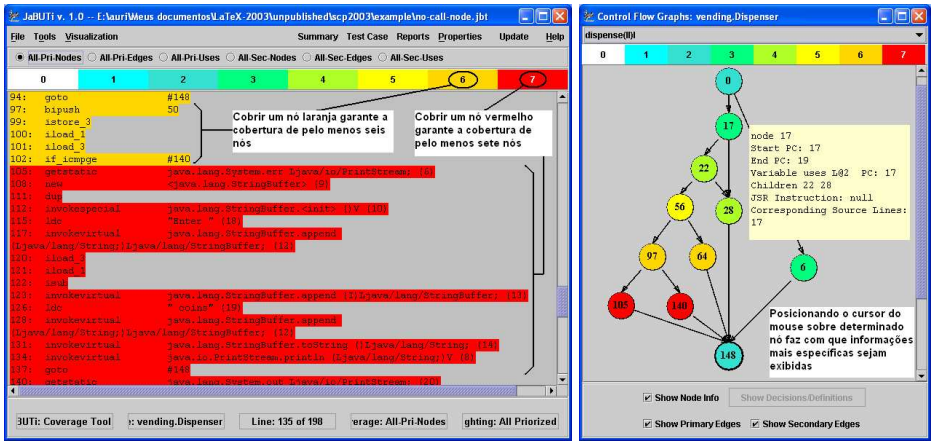


Figura 13: Janela do gerenciador de projetos.

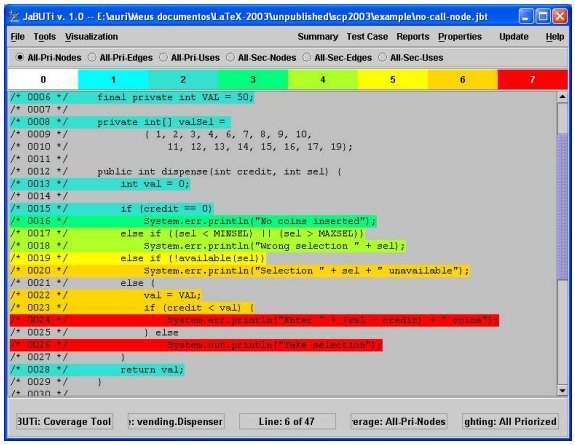
Pressionando o botão `Ok`, *JaBUTi* cria um novo projeto (`vending.jbt` no exemplo), constrói o grafo Definição-Uso (*DU*) para cada método de cada classe a ser testada, deriva os requisitos de teste de cada critério, calcula o peso desses requisitos, e apresenta na tela o bytecode de uma das classes sendo testadas, como ilustrado na Figura 14(a)[6]. Além da visualização do bytecode, a ferramenta oferece ainda a visualização do grafo *DU* de cada método (Figura 14(b)), e também do código-fonte correspondente ao bytecode (Figura 14(c)), quando tal código encontra-se disponível.

Uma vez que o conjunto de requisitos de teste de cada critério foi determinado, tais requisitos podem ser utilizados para avaliar a qualidade de um conjunto de teste existente e/ou para desenvolver novos casos de teste visando a melhorar a cobertura dos requisitos pelo conjunto de teste. O testador pode, por exemplo, decidir criar um conjunto de teste com base em critérios de teste funcionais ou mesmo gerar um conjunto de teste *ad-hoc* e avaliar a cobertura desse conjunto de teste em relação a cada um dos critérios de teste estruturais da *JaBUTi*. Por outro lado, o testador pode visualizar o conjunto de requisitos de teste de cada critério gerado para cada um dos métodos

[6]As telas apresentadas nesta seção são referentes a uma versão da ferramenta na qual os critérios independentes de exceção eram denominados Critérios Primários e os critérios dependentes de exceção eram denominados Critérios Secundários. Assim, o critério `All-Pri-Nodes` que aparece nas telas corresponde ao critério `Todos-Nós`$_{ei}$ e o critério `All-Sec-Nodes` corresponde ao critério `Todos-Nós`$_{ed}$. A mesma consideração é válida para os demais critérios de teste. No texto é utilizado o nome dos critérios conforme definido na Seção 3.



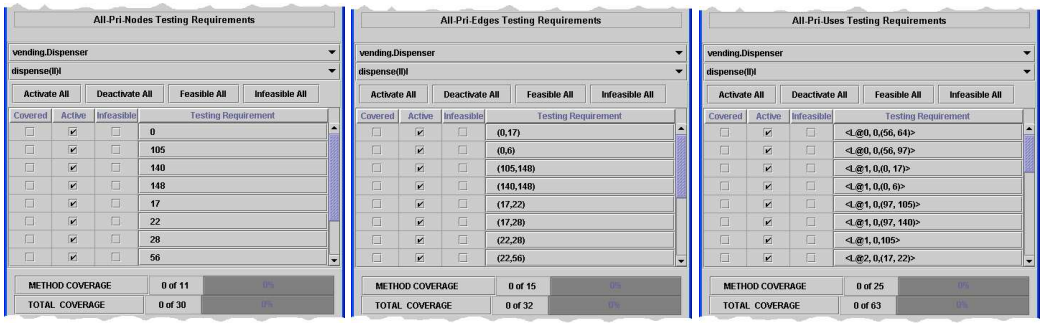
(a) Bytecode inicial. (b) *DU* inicial.



(c) Código-fonte inicial.

Figura 14: Tela da *JaBUTi* considerando ativo o critério Todos-Nós$_{ei}$.

das classes sendo testadas, verificar quais deles ainda não foram cobertos por algum caso de teste e então desenvolver um novo caso de teste que satisfaça tais requisitos. As figuras 15(a), 15(b), e 15(c) ilustram parte dos requisitos de teste do método `Dispenser.avaliable()` gerados pelos critérios Todos-Nós$_{ei}$, Todas-Arestas$_{ei}$, e Todos-Usos$_{ei}$, respectivamente.



(a) Todos-Nós$_{ei}$. (b) Todas-Arestas$_{ei}$. (c) Todos-Usos$_{ei}$.

Figura 15: Parte dos requisitos de teste de três critérios estruturais para o método `Dispenser.dispense()`.

Ainda, como pode ser observado na Figura 15, a ferramenta permite ao testador ativar/desativar diferentes combinações de requisitos de teste, bem como marcar um determinado requisito de teste como não-executável quando não existir um caso de teste capaz de cobri-lo.

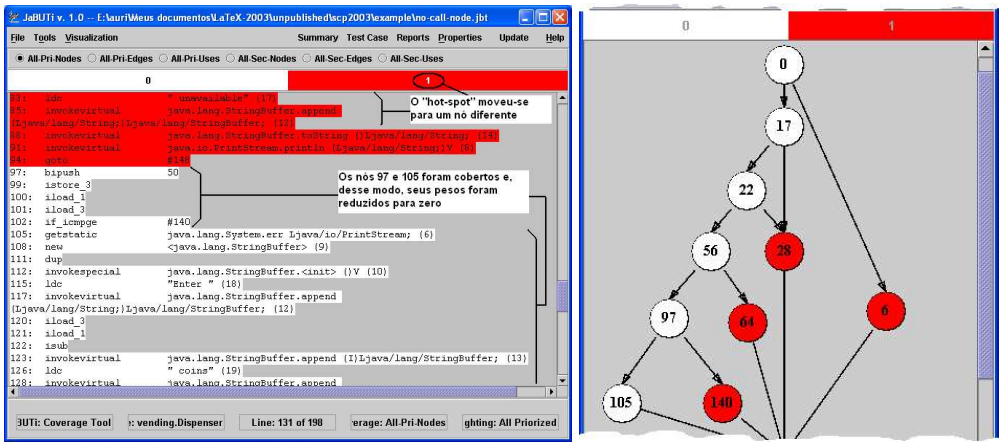
Estudos têm demonstrado que, para programas C, conjuntos de teste que determinam as maiores coberturas têm uma maior probabilidade de detectar defeitos no programa em teste [97]. O mesmo se aplica para Java sem perda de generalidade. Uma vez que, em geral, existe um grande número de requisitos de teste para serem cobertos, a ferramenta *JaBUTi* utiliza cores diferentes dando indicações ao testador para facilitar a geração de casos de teste que satisfaçam um maior número de requisitos em menor tempo. As diferentes cores representam diferentes pesos que são associados aos requisitos de teste de cada critério. Informalmente, os pesos correspondem ao número de requisitos de teste que são cobertos quando um requisito de teste particular é satisfeito. Assim, cobrir os requisitos de teste de maior peso leva a um aumento na cobertura de forma mais rápida.

Observe que os pesos são calculados considerando somente informações sobre cobertura e por esse motivo devem ser vistos como "indicações" ao testador. No cálculo do peso não é levado em conta, por exemplo, a complexidade ou a criticalidade de uma determinada parte do programa. Além disso, o cálculo do peso por meio do uso de superblocos e de dominadores não é necessário quando não se pode garantir a execução normal do método até um nó de saída, o que pode ocorrer em Java quando uma exceção é levantada. O testador, baseado em sua experiência, pode decidir cobrir outros requisitos de teste que estejam com trechos de código que apresentam uma alta complexidade e não tenham os maiores pesos. Posteriormente, após os trechos de código desejado terem sido suficientemente testados, o testador pode então utilizar as indicações para melhorar a cobertura do conjunto de teste de forma mais rápida.

A Figura 14 mostra parte do bytecode, do grafo *DU* e do código-fonte do método `Dispenser.dispense()` antes da execução de qualquer caso de teste. As cores correspondem aos diferentes pesos dos requisitos do critério `Todos-Nós`$_{ei}$. Observe que a barra de corres vai do branco (peso 0) ao vermelho (peso 7 neste exemplo). O nó 105 da Figura 14(b), composto das instruções de bytecode que vão do `pc` 105 ao `pc` 112 (Figura 14(a)), é um dos nós de maior peso. Isso representa que um caso de teste que exercite o nó 105 irá aumentar a cobertura em relação ao critério `Todos-Nós`$_{ei}$ em pelo menos 7 nós. Requisitos com peso zero indicam requisitos que já foram cobertos e são pintados em branco. Por exemplo, o caso de teste `0001`, desenvolvido para executar o comando localizado no nó 105 da Figura 14(b), determina uma cobertura de 56% em relação ao critério `Todos-Nós`$_{ei}$, o que corresponde a execução de 17 dos 30 nós independentes de exceção requeridos pelo critério em relação aos métodos de todas as classes sendo testadas (veja relatório da Figura 18(c)). Toda vez que um novo caso de teste é inserido, as telas da ferramenta são atualizadas considerando as possíveis mudanças nos pesos dos requisitos de teste.

A Figura 16(a) ilustra os novos pesos dos requisitos de teste do critério `Todos-Nós`$_{ei}$ para o método `Dispenser.dispense()`. Nessa figura muitos blocos estão pintados de branco porque foram cobertos pelo caso de teste `0001`. Observe, ainda, que o requisito de maior peso passou para outra parte do código, dando indicações a respeito de qual o próximo caso de teste que deveria ser gerado visando a maximizar o número de requisitos cobertos. A diferença no peso dos requisitos após a execução do caso de teste `0001` pode ser facilmente identificada comparando as figuras 14(a) e 16(a). No exemplo, o requisito de maior peso foi reduzido de 7 para 1. Isso é consistente com o entendimento dos critérios baseados em cobertura para os quais torna-se cada vez mais difícil melhorar a cobertura após alguns casos de teste terem sido executados.

A título de comparação, supondo que ao invés de utilizar as indicações fornecidas pela ferramenta o testador gerasse um outro caso de teste, por exemplo, o caso de teste `0003` da Fi-



(a) Bytecode atualizado. (b) Grafo *DU* atualizado.

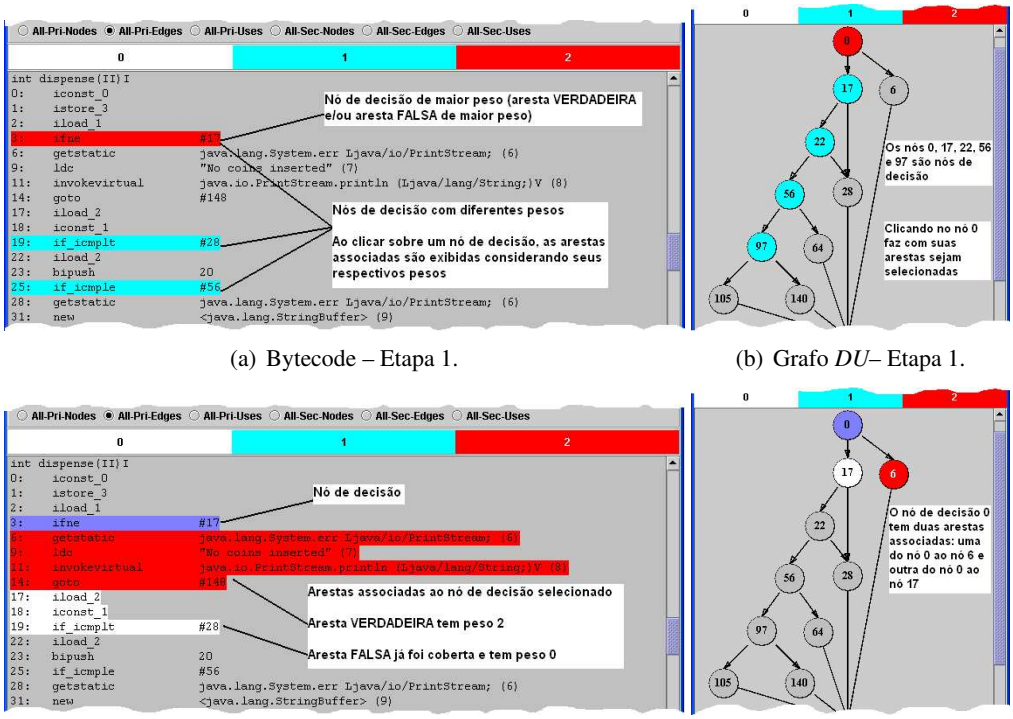
Figura 16: Tela atualizada do método `Dispenser.dispense()` para o critério `Todos-Nós`$_{ei}$ após a execução do caso de teste `0001`.

gura 18(c), o número máximo de nós independentes de exceção que teria sido coberto seria 7. Com isso, é possível notar uma diferença significativa entre a cobertura total do critério `Todos-Nós`$_{ei}$ determinada pelo caso de teste `0001`, que foi de 56%, e aquela determinada pelo caso de teste `0003`, que foi de 23%. Uma diferença semelhante é também encontrada em relação aos demais critérios de teste.

A ferramenta *JaBUTi* também permite que os requisitos de teste de cada um de seus critérios possam ser visualizados no bytecode, código-fonte e no grafo *DU* de cada método de cada uma das classes em teste. Diferentes cores são associadas a esses requisitos para indicar os seus pesos. Por exemplo, as figuras 14(a) e 14(b) ilustram os requisitos do critério `Todos-Nós`$_{ei}$. Considerando os critérios `Todas-Arestas`$_{ei}$ e `Todas-Arestas`$_{ed}$, seus requisitos (arestas do grafo *DU*) são coloridos em duas etapas. Para o critério `Todas-Arestas`$_{ei}$, somente os nós que apresentam mais de uma aresta regular saindo, ou seja, nós de decisão, aparecem pintados na primeira etapa para indicar os pontos do código onde os comandos de decisão estão localizados. Por exemplo, as figuras 17(a) e 17(b) ilustram parte dos nós de decisão do método `Dispenser.dispense()` e como eles são coloridos na primeira etapa após três casos de teste terem sido executados.

Para cada nó de decisão, seu peso é o maior peso de seus arcos correspondentes. Assim, supondo um nó de decisão n_d com duas arestas saindo, uma com peso 0 e outra com peso 2, n_d terá peso 2. Este é o caso do nó 0 no grafo *DU* do exemplo utilizado (Figura 17(b)). Os cinco nós de decisão da figura (0, 17, 22, 56 e 97) estão em diferentes cores porque possuem diferentes pesos associados. Um nó de decisão tem peso zero se e somente se todos os arcos a ele associados tiverem sido cobertos. Tal nó é pintado de branco nesse caso. A seleção de um nó de decisão faz com que a segunda etapa seja realizada (figuras 17(c) e 17(d)), identificando-se todas as arestas regulares associadas com o ponto de decisão selecionado. No caso da Figura 17(d), são duas as arestas regulares correspondentes: uma do nó 0 para o nó 6 que ainda não foi coberta e está pintada de vermelho, e outra do nó 0 para o nó 17 que já foi coberta e aparece pintada de branco. Os demais critérios de Fluxo de Dados também são pintados de forma similar, em duas etapas. Mais informações podem ser obtidas em [92].

A ferramenta permite ainda a geração de diferentes relatórios de teste, em diferentes granularidades, dependendo do nível de detalhe desejado, para avaliar o andamento da atividade de teste. Por exemplo, o testador pode estar interessado em avaliar a cobertura do critério `Todos-Nós`$_{ei}$

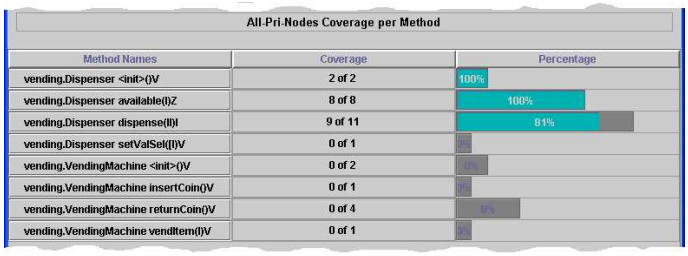


(a) Bytecode – Etapa 1. (b) Grafo *DU*– Etapa 1.

(c) Bytecode – Etapa 2. (d) Grafo *DU*– Etapa 2.

Figura 17: Etapas da exibição dos requisitos do critério `Todas-Arestas`$_{ei}$ para o método `Dispenser.dispense()`.

em relação a cada método para descobrir quais desses métodos ainda não foi suficientemente testado. Esse tipo de relatório é fornecido pela *JaBUTi* como ilustrado na Figura 18(a).



(a) Cobertura de cada método: critério `Todos-Nós`$_{ei}$.

(b) Cobertura obtida em relação a cada critério. (c) Cobertura de cada caso de teste: critério `Todos-Nós`$_{ei}$.

Figura 18: Relatórios de teste da cobertura em relação a cada método, critério e caso de teste.

Também pode ser importante avaliar a cobertura de todo o projeto em relação a cada um dos critérios de teste. Essa informação pode ajudar o testador a decidir se o "efeito de saturação"

de determinado critério de teste já foi atingido. Em caso afirmativo, um critério de teste mais forte pode ser utilizado para continuar a evolução do conjunto de teste; por exemplo, passar do critério Todos-Nós$_{ei}$ para o critério Todas-Arestas$_{ei}$. A ferramenta *JaBUTi* gera esse tipo de relatório considerando os seis critérios de teste estruturais implementados pela mesma. A Figura 18(b) ilustra esse tipo de relatório.

Além disso, pode ser interessante identificar os *slices* de execução de cada caso de teste, onde o *slice* pode ser definido em relação a um conjuntos de nós independentes/dependentes de exceção (considerando o critério Todos-Nós$_{ei}$/Todos-Nós$_{ed}$), arestas independentes/dependentes de exceção, associações independentes/dependentes de exceção e potenciais-associações independentes/dependentes de exceção, obtidas a partir dos critérios de teste correspondentes. Tais *slices* podem ser utilizados, por exemplo, de forma similar às metodologias desenvolvidas para programas C, para auxiliar aos testadores em atividades de depuração e entendimento de programas/componentes Java. Para um dado caso de teste, seu *slice* de execução pode ser facilmente construído se a cobertura determinada por tal caso de teste é conhecida, tal como o relatório mostrado na Figura 18(c). Observe que é possível ativar/desativar diferentes combinações de casos de teste de modo que o *slice* de execução de cada um, tanto no bytecode quando no grafo *DU*, possa ser visualizado, como mostrado anteriormente na Figura 14(a), por exemplo.

É importante observar que a disponibilidade da ferramenta *JaBUTi* viabiliza a condução de diversos estudos empíricos, tanto no que diz respeito ao desenvolvimento de estratégias de teste incrementais, quanto na utilização das informações obtidas durante os testes em atividades de depuração e entendimento de programas OO. Além disso, *JaBUTi* é a única ferramenta que apóia a aplicação do teste de Fluxo de Dados em programas e componentes Java, podendo ser utilizada tanto na transferência tecnológica de conceitos/critérios de teste para a indústria como na atividade de ensino/aprendizagem em disciplinas da área de teste de software. Informações adicionais a respeito da ferramenta *JaBUTi* podem ser obtidas em [92, 94].

Cabe ressaltar, por fim, a existência de duas outras versões da ferramenta *JaBUTi*: (1) *JaBUTi/AJ* (*Java Bytecode UnderstandiIng and Testing / AspectJ*) [54], para o teste de unidade de programas orientados a aspectos (OA), baseados na linguagem AspectJ; e (2) *JaBUTi/MA* (*Java Bytecode UnderstandiIng and Testing / Mobile Agents*) [19], para o teste estrutural de agentes móveis.

4 Estudos Teóricos e Empíricos

Em virtude da diversidade de critérios de teste existente, saber qual deles deve ser utilizado ou como utilizá-los de forma complementar a fim de obter o melhor resultado com o menor custo é uma questão complicada. Nesse sentido, estudos teóricos e empíricos têm sido conduzidos na tentativa de avaliar as vantagens e desvantagens dos critérios de teste [14].

Os estudos teóricos têm explorado, sobretudo, a relação de inclusão entre os critérios e a complexidade dos mesmos [67, 76, 95]. A relação de inclusão estabelece uma ordem parcial e caracteriza uma hierarquia entre eles. Assim, diz-se que um critério C_1 inclui um critério C_2 se para qualquer programa P e qualquer conjunto de casos de teste T_1 C_1-adequado, T_1 for também C_2-adequado e para algum programa P e um conjunto T_2 C_2-adequado, T_2 não for C_1-adequado. A complexidade é definida como o número máximo de casos de teste exigidos por um critério, no pior caso. Além desses itens, também do ponto de vista teórico, alguns autores têm abordado a questão de eficácia dos critérios de teste, definindo outras relações de inclusão entre eles, as quais procurem captar a capacidade em revelar erros [30, 31, 96, 100].

Os estudos empíricos, por sua vez, procuram comparar a adequação dos critérios de teste a partir de três fatores básicos: custo, eficácia e dificuldade de satisfação (*strength*). O custo refere-

se ao esforço necessário na utilização de um critério. Pode ser medido pelo número de casos de teste requeridos 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 os mutantes equivalentes, caminhos e associações não executáveis, construir manualmente os casos de teste e aprender a utilizar as ferramentas de teste. A eficácia refere-se à capacidade de um critério em detectar um maior número de erros em relação a outro. Dificuldade de satisfação refere-se à probabilidade de satisfazer um critério tendo satisfeito outro.

Utilizando-se tais fatores comparativos, estudos teóricos e empíricos são conduzidos com o objetivo de encontrar formas econômicas e produtivas para a realização dos testes. Uma visão geral a respeito dos principais estudos realizados pode ser encontrada em [58, 100]. Uma síntese quanto à avaliação e comparação entre critérios de teste OO está disponível em [92].

5 Conclusões e Direções na Área de Teste

O teste é uma atividade crucial no processo de desenvolvimento de software, tendo forte relação com aspectos relacionados à garantia da qualidade do produto em questão [14]. Neste texto foi dada uma visão geral a respeito da atividade de teste, sendo apresentados alguns conceitos e critérios pertinentes, com ênfase naqueles considerados mais promissores a curto e médio prazo – os critérios baseados em Fluxo de Dados e os critérios baseados em Mutação. Mostrou-se, também, que os conceitos e mecanismos desenvolvidos originalmente para o teste de programas procedimentais podem ser utilizados no contexto do paradigma de desenvolvimento de software orientado a objetos, com as devidas adaptações. Extensões de critérios de teste baseados em Fluxo de Controle, Fluxo de Dados e em Mutação foram brevemente discutidas nesse contexto. Também foram discutidas algumas iniciativas e esforços de automatização de critérios de teste. Em especial, foram apresentadas as ferramentas *PokeTool*, *Proteum* e *JaBUTi*.

No que se refere a direções futuras, Harrold [38] discute as perspectivas, as necessidades e as tendências na área de teste, visando ao desenvolvimento de métodos e ferramentas que permitam a transferência de tecnologia para indústria. Como ressalta Harrold, dentre as principais linhas de pesquisa na área destacam-se: (1) o teste de sistemas baseado em componentes de software (o qual inclui o teste de programas OO); (2) o desenvolvimento de processos de teste efetivos; e (3) a demonstração da eficácia de critérios e estratégias de teste. De fato, um aspecto relevante é dar subsídios para o desenvolvimento de software baseado em componentes. Com o aumento no desenvolvimento de produtos desse tipo, é necessária a definição de modos efetivos e eficientes de testá-los. É necessário, ainda, entender e desenvolver técnicas e critérios de teste que exercitem as várias questões associadas, tais como segurança e tolerância a falhas. Além disso, os tópicos de verificação, validação e teste de software também começam a ser investigados e discutidos no contexto de programação orientada a aspectos (POA) [54].

Para finalizar, ressalta-se que o conhecimento e as contribuições na área de teste – divididos basicamente em conhecimento *teórico*, *empírico* e *de ferramentas de suporte* – devem ser constantemente atualizados, assim como nas demais áreas. Nessa perspectiva, a organização de uma base histórica sobre o custo e a eficácia das técnicas e critérios de teste, em diferentes domínios de aplicação, em relação a diferentes classes de erros, certamente facilitaria o planejamento de futuros desenvolvimentos de software. Facilitaria, ainda, o estabelecimento de estratégias de teste que explorem os aspectos complementares das técnicas e critérios, viabilizando a detecção do maior número de erros possível e com o menor custo, o que contribuiria para a liberação de produtos de software de maior qualidade a um menor custo [14].

Referências

[1] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Mutation analysis. Technical Report GIT-ICS-79/08, Georgia Institute of Technology, Atlanta, GA, September 1979.

[2] H. Agrawal, R. A. DeMillo, R. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and 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, March 1989.

[3] T. R. Arnold and W. A. Fuson. In a perfect world. *Communications of the ACM*, 37(9):78–86, September 1994.

[4] E. F. Barbosa, J. C. Maldonado, A. M. R. Vincenzi, M. E. Delamaro, S. R. S. Souza, and M. Jino. Introdução ao teste de software. Minicurso apresentado no XIV Simpósio Brasileiro de Engenharia de Software (SBES 2000), October 2000.

[5] E. F. Barbosa, A. M. R. Vincenzi, and J. C. Maldonado. Uma contribuição para a determinação de um conjunto essencial de operadores de mutação no teste de programas C. In *XII Simpósio Brasileiro de Engenharia de Software (SBES 98)*, pages 103–120, Maringá, PR, October 1998.

[6] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold Company, New York, 2nd edition, 1990.

[7] R. V. Binder. Modal testing strategies for OO software. *Computer*, 29(11):97–99, November 1996.

[8] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*, volume 1. Addison Wesley Longman, Inc., 1999.

[9] T. A. Budd. *Mutation Analysis of Program Test Data*. PhD thesis, Yale University, New Haven, CT, 1980.

[10] T. A. Budd. *Mutation Analysis: Ideas, Example, Problems and Prospects*, chapter Computer Program Testing. North-Holand Publishing Company, 1981.

[11] M. L. Chaim. PokeTool – Uma ferramenta para suporte ao teste estrutural de programas baseado em análise de fluxo de dados. Master's thesis, DCA/FEEC/UNICAMP, Campinas, SP, April 1991.

[12] T. S. Chow. Testing software design modelled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, 1978.

[13] T. Chusho. Test data selection and quality estimation based on concept of essential branches for path testing. *IEEE Transactions on Software Engineering*, 13(7):509–517, 1987.

[14] A. R. C. da Rocha, J. C. Maldonado, and K. C. Weber. *Qualidade de Software: Teoria e Prática*. Prentice Hall, São Paulo, SP, 2001.

[15] M. E. Delamaro and J. C. Maldonado. Proteum – a tool for the assessment of test adequacy for C programs. In *Conference on Performability in Computing Systems (PCS 96)*, pages 79–95, Brunswick, NJ, July 1996.

[16] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur. Interface mutation: An approach for integration testing. *IEEE Transactions on Software Engineering*, 27(3):228–247, March 2001.

[17] M. E. Delamaro, J. C. Maldonado, and A. M. R. Vincenzi. Proteum/IM 2.0: An integrated mutation testing environment. In *Mutation 2000 Symposium*, pages 91–101, San Jose, CA, October 2000. Kluwer Academic Publishers.

[18] M. E. Delamaro, M. Pèzze, A. M. R. Vincenzi, and J. C. Maldonado. Mutant operators for testing concurrent Java programs. In *XV Simpósio Brasileiro de Engenharia de Software (SBES 2001)*, Rio de Janeiro, RJ, October 2001.

[19] M. E. Delamaro and A. M. R. Vincenzi. Structural Testing of Mobile Agents. In Egidio Astesiano Nicolas Guelfi and Gianna Reggio, editors, *International Workshop on Scientific Engineering of Java Distributed Applications (FIDJI 2003)*, volume 2952 of *Lecture Notes on Computer Science*, pages 73–85. Springer, November 2003.

[20] R. A. Demillo. Mutation analysis as a tool for software quality assurance. In *COMPSAC80*, Chicago, IL, October 1980.

[21] R. A Demillo. *Software Testing and Evaluation*. The Benjamin/Cummings Publishing Company Inc., 1987.

[22] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–43, April 1978.

[23] 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*. PhD thesis, IFSC-USP, São Carlos – SP, October 1996.

[24] S. C. P. F. Fabbri, J. C. Maldonado, M. E. Delamaro, and P. C. Masiero. Proteum/FSM: A tool to support finite state machine validation based on mutation testing. In *XIX International Conference of the Chilean Computer Science Society (SCCC 99)*, pages 96–104, Talca, Chile, 1999.

[25] S. C. P. F. Fabbri, J. C. Maldonado, and P. C. Masiero. Mutation analysis in the context of reactive system specification and validation. In *5th Annual International Conference on Software Quality Management*, pages 247–258, Bath, UK, March 1997.

[26] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero, and M. E. Delamaro. Análise de mutantes baseada em máquinas de estado finito. In *XI Simpósio Brasileiro de Redes de Computadores (SBRC 93)*, pages 407–425, Campinas, SP, May 1993.

[27] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero, and M. E. Delamaro. Mutation analysis testing for finite state machines. In *5th International Symposium on Software Reliability Engineering (ISSRE 94)*, pages 220–229, Monterey, CA, November 1994.

[28] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero, and M. E. Delamaro. Mutation analysis applied to validate specifications based on petri nets. In *8th IFIP Conference on Formal Descriptions Techniques for Distribute Systems and Communication Protocols (FORTE 95)*, pages 329–337, Montreal, Canada, October 1995.

[29] S. C. P. F. Fabbri, J. C. Maldonado, T. Sugeta, and P. C. Masiero. Mutation testing applied to validate specifications based on statecharts. In *ISSRE – International Symposium on Software Reliability Systems*, pages 210–219, November 1999.

[30] P. G. Frankl and E. J. Weyuker. An analytical comparison of the fault-detecting ability of data flow testing techniques. In *XV International Conference on Software Engineering*, pages 415–424, May 1993.

[31] P. G. Frankl and E. J. Weyuker. A formal analysis of the fault-detecting ability of testing methods. *IEEE Transactions on Software Engineering*, 19(3):202–213, March 1993.

[32] A. D. Friedman. *Logical Design of Digital Systems*. Computer Science Press, 1975.

[33] S. Fujiwara, G. V. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6), June 1991.

[34] C. Ghezzi and M. Jazayeri. *Programming Languages Concepts*. John Wiley and Sons, New York, 2 edition, 1987.

[35] S. Ghosh and A. P. Mathur. Interface mutation. In *Mutation 2000 Symposium*, pages 227–247, San Jose, CA, October 2000. Kluwer Academic Publishers.

[36] G. Gönenç. A method for design of fault-detection experiments. *IEEE Transactions on Computers*, 19(6):551–558, June 1970.

[37] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[38] M. J. Harrold. Testing: A roadmap. In *22th International Conference on Software Engineering*, June 2000.

[39] M. J. Harrold, D. Liang, and S. Sinha. An approach to analyzing and testing component-based systems. In *First International ICSE Workshop on Testing Distributed Component-Based Systems*, Los Angeles, CA, May 1999.

[40] M. J. Harrold, J. D. McGregor, and K. J. Fitzpatrick. Incremental testing of object-oriented class structures. In *14th International Conference on Software Engineering*, pages 68–80, Los Alamitos, CA, May 1992. IEEE Computer Society Press.

[41] M. J. Harrold and G. Rothermel. Performing data flow testing on classes. In *Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 154–163, New York, December 1994. ACM Press.

[42] M. J. Harrold and M. L. Soffa. Interprocedural data flow testing. In *Third Testing, Analysis, and Verification Symposium*, pages 158–167, December 1989.

[43] M. J. Harrold and M. L. Soffa. Selecting and using data for integration test. *IEEE Software*, 8(2):58–65, March 1991.

[44] P. M. Herman. A data flow analysis approach to program testing. *Australian Computer Journal*, 8(3), November 1976.

[45] D. Hoffman and P. Strooper. ClassBrench: A framework for automated class testing. *Software Practice and Experience*, pages 573–597, May 1997.

[46] J. R. Horgan and P. Mathur. Assessing testing tools in research and education. *IEEE Software*, 9(3):61–69, May 1992.

[47] W. E. Howden. Methodology for the generation of program test data. *IEEE Computer*, C-24(5):554–559, May 1975.

[48] W. E. Howden. *Software Engineering and Technology: Functional Program Testing and Analysis*. McGrall-Hill Book Co, New York, 1987.

[49] IEEE. IEEE Standard Glossary of Software Engineering Terminology. Standard 610.12, IEEE Press, 1990.

[50] S. Kim, J. A. Clark, and 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)*, December 1999. Disponível em http://www-users.cs.york.ac.uk/ jac/, Último acesso: 12/2003.

[51] S. Kim, J. A. Clark, and J. A. Mcdermid. Class mutation: Mutation testing for object-oriented programs. In *FMES*, 2000. Disponível em http://www-users.cs.york.ac.uk/ jac/, Último acesso: 12/2003.

[52] D. C. Kung, P. Hsia, and J. Gao. *Testing Object-Oriented Software*. IEEE Computer Society Press, Los Alamitos, CA, 1998.

[53] J. W. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, 9(3), May 1983.

[54] O. A. L. Lemos. Teste de programas orientados a aspectos: uma abordagem estrutural para AspectJ. Master's thesis, ICMC-USP, São Carlos, SP, 2005.

[55] U. Linnenkugel and M. Müllerburg. Test data selection criteria for (software) integration testing. In *First International Conference on Systems Integration*, pages 709–717, Morristown, NJ, April 1990.

[56] Y. S. Ma, Y.-R. Kwon, and J. Offutt. Inter-class mutation operators for java. In *13th International Symposium on Software Reliability Engineering (ISSRE 2002)*, pages 352–366, Annapolis, MD, November 2002.

[57] J. C. Maldonado. *Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software*. PhD thesis, DCA/FEEC/UNICAMP, Campinas, SP, July 1991.

[58] J. C. Maldonado, E. F. Barbosa, A. M. R. Vincenzi, M. E. Delamaro, S. R. S. Souza, and M. Jino. Teste de software: Teoria e prática. Minicursos – XVII Simpósio Brasileiro de Engenharia de Software (SBES 2003), October 2003.

[59] J. C. Maldonado, M. L. Chaim, and M. Jino. Arquitetura de uma ferramenta de teste de apoio aos critérios potenciais usos. In *XXII Congresso Nacional de Informática*, São Paulo, SP, September 1989.

[60] J. C. Maldonado, M. E. Delamaro, S. C. P. F. Fabbri, A. S. Simão, T. Sugeta, A. M. R. Vincenzi, and P. C. Masiero. Proteum: A family of tools to support specification and program testing based on mutation. In *Mutation 2000 Symposium – Tool Session*, pages 113–116, San Jose, CA, October 2000. Kluwer Academic Publishers.

[61] J. C. Maldonado, A. M. R. Vincenzi, E. F. Barbosa, S. R. S. Souza, and M. E. Delamaro. Aspectos teóricos e empíricos de teste de cobertura de software. Technical Report 31, Instituto de Ciências Matemáticas e de Computação – ICMC-USP, June 1998.

[62] A. P. Mathur. On the relative strengths of data flow and mutation testing. In *Ninth Annual Pacific Northwest Software Quality Conference*, pages 165–181, Portland, OR, October 1991.

[63] R. McDaniel and J. D. McGregor. Testing polymorphic interactions between classes. Technical Report TR-94-103, Clemson University, March 1994.

[64] J. D. McGregor. Functional testing of classes. In *Proc. 7th International Quality Week*, San Francisco, CA, May 1994. Software Research Institute.

[65] G. J. Myers, Corey Sandler, Tom Badgett, and Todd M. Thomas. *The Art of Software Testing*. John Wiley & Sons, 2nd. edition, 2004.

[66] S. C. Ntafos. On required element testing. *IEEE Transactions on Software Engineering*, SE-10:795–803, November 1984.

[67] S. C. Ntafos. A comparison of some structural testing strategies. *IEEE Transactions on Software Engineering*, 14(6):868–873, July 1988.

[68] A. J. Offutt and A. Irvine. Testing object-oriented software using the category-partition method. In *17th International Conference on Technology of Object-Oriented Languages and Systems*, pages 293–304, Santa Barbara, CA, August 1995.

[69] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering Methodology*, 5(2):99–118, April 1996.

[70] A. J. Offutt, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation. In *15th International Conference on Software Engineering (ICSE 93)*, pages 100–107, Baltimore, MD, May 1993.

[71] A. Orso, M. J. Harrold, D. Rosenblum, G. Rothermel, H. Do, and M. L. Soffa. Using component metacontent to support the regression testing of component-based software. In *IEEE International Conference on Software Maintenance (ICSM 2001)*, 2001.

[72] D. E. Perry and G. E. Kaiser. Adequate testing and object-oriented programming. *Journal on Object-Oriented Programming*, pages 13–19, January/February 1990.

[73] Roger S. Pressman. *Software Engineering - A Practitioner's Approach*. McGraw-Hill, 5 edition, 2001.

[74] R. L. Probert and F. Guo. Mutation testing of protocols: Principles and preliminary experimental results. In *Third International Workshop on Protocol Test Systems (IFIP TC6)*, pages 57–76. North-Holland, 1991.

[75] S. Rapps and E. J. Weyuker. Data flow analysis techniques for program test data selection. In *6th International Conference on Software Engineering*, pages 272–278, Tokio, Japan, September 1982.

[76] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, April 1985.

[77] A. C. A. Rosa and E. Martins. Using a reflexive architecture to validate object-oriented applications by fault injection. In *Workshop on Reflexive Programming in C++ and Java*, pages 76–80, Vancouver, Canada, 1998. (http://www.dc.unicamp.br/ eliane).

[78] D. S. Rosenblum. Adequate testing of component-based software. Technical Report UCI-ICS-97-34, University of California, Irvine, CA, August 1997.

[79] K. K. Sabnani and A. Dahbura. Protocol test generation procedure. *Computer Networks and ISDN Systems*, 15(4):285–297, April 1988.

[80] A. S. Simão. Proteum-RS/PN: Uma ferramenta para a validação de redes de petri baseada na análise de mutantes. Master's thesis, ICMC-USP, São Carlos, SP, February 2000.

[81] A. S. Simão and J. C. Maldonado. Geração de seqüências para redes de Petri baseadas em mutação. In *III Workshop de Métodos Formais*, João Pessoa, October 2000.

[82] A. S. Simão, J. C. Maldonado, and 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)*, João Pessoa, October 2000.

[83] S. R. S. Souza. Avaliação do custo e eficácia do critério análise de mutantes na atividade de teste de programas. Master's thesis, ICMC-USP, São Carlos, SP, June 1996.

[84] S. R. S. Souza, J. C. Maldonado, S. C. P. F. Fabbri, and W. Lopes de Souza. Mutation testing applied to estelle specifications. *Software Quality Journal*, 8(4):285–302, April 2000. Kluwer Academic Publishers.

[85] T. Sugeta. Proteum-RS/ST : Uma ferramenta para apoiar a validação de especificações statecharts baseada na análise de mutantes. Master's thesis, ICMC-USP, São Carlos, SP, November 1999.

[86] T. Sugeta, J. C. Maldonado, and W. E. Wong. Mutation testing applied to validate SDL specifications. In *16th IFIP International Conference on Testing of Communicating Systems (TestCom 2004)*, Oxford, United Kingdom.

[87] C. D. Turner and D. J. Robson. The state-based testing of object-oriented programs. In *IEEE Conference on Software Maintenance*, pages 302–310, 1993.

[88] H. Ural and B. Yang. A structural test selection criterion. *Information Processing Letters*, 28:157–163, 1988.

[89] S. R. Vergílio, J. C. Maldonado, and M. Jino. Uma estratégia para a geração de dados de teste. In *VII Simpósio Brasileiro de Engenharia de Software (SBES 93)*, pages 307–319, Rio de Janeiro, RJ, October 1993.

[90] P. R. S. Vilela. *Critérios Potenciais Usos de Integração: Definição e Análise*. PhD thesis, DCA/FEEC/UNICAMP, Campinas, SP, April 1998.

[91] P. R. S. Vilela, J. C. Maldonado, and M. Jino. Program graph visualization. *Software Practice and Experience*, 27(11):1245–1262, November 1997.

[92] A. M. R. Vincenzi. *Orientação a Objetos: Definição, Implementação e Análise de Recursos de Teste e Validação*. PhD thesis, Instituto de Ciências Matemáticas e de Computação – ICMC-USP, São Carlos, SP, March 2004.

[93] A. M. R. Vincenzi, J. C. Maldonado, M. E. Delamaro, E. S. Spoto, and E. Wong. *Desenvolvimento Baseado em Componentes: Conceitos e Técnicas*, chapter Software Baseado em Componentes: Uma Revisão sobre Teste. Editora Ciência Moderna Ltda., 2005.

[94] A. M. R. Vincenzi, W. E. Wong, M. E. Delamaro, and J. C. Maldonado. JaBUTi: A coverage analysis tool for java programs. In *XVII Simpósio Brasileiro de Engenharia de Software (SBES 2003)*, Manaus, AM, October 2003.

[95] E. J. Weyuker. The complexity of data flow for test data selection. *Information Processing Letters*, 19(2):103–109, August 1984.

[96] E. J. Weyuker and B. Jeng. Analyzing partition testing strategies. *IEEE Transactions on Software Engineering*, 17(7):703–711, July 1991.

[97] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set size and block coverage on fault detection effectiveness. In *Fifth IEEE International Symposium on Software Reliability Engineering*, pages 230–238, Monterey, CA, November 1994.

[98] M. R. Woodward. Mutation testing – its origin and evolution. *Information and Software Technology*, 35(3):163–169, March 1993.

[99] M. R. Woodward, D. Heddley, and M. A. Hennel. Experience with path analysis and testing of programs. *IEEE Transactions on Software Engineering*, SE-6:278–286, May 1980.

[100] H. Zhu. A formal analysis of the subsume relation between software test adequacy criteria. *IEEE Transactions on Software Engineering*, SE-22(4):248–255, April 1996.

View publication stats