Introdução ao Teste de Software*

Ellen Francine Barbosa
José Carlos Maldonado
Auri Marcelo Rizzo Vincenzi
Universidade de São Paulo — ICMC/USP
{francine, jcmaldon, auri}@icmc.sc.usp.br

Márcio Eduardo Delamaro
Universidade Estadual de Maringá — DIN/UEM
delamaro@din.uem.br

Simone do Rocio Senger de Souza
Universidade Estadual de Ponta Grossa — UEPG
rocio@icmc.sc.usp.br

Mario Jino
Universidade Estadual de Campinas — DCA/FEEC/UNICAMP
jino@dca.fee.unicamp.br

Resumo

Neste texto são apresentados alguns critérios de teste de software, conceitos pertinentes e ferramentas de apoio. São abordados critérios de teste funcional, estrutural – baseados em fluxo de controle e em fluxo de dados – e baseados em mutação. Ênfase é dada no teste de mutação. Apesar de sua eficácia em revelar a presença de erros, o teste de mutação apresenta problemas de custo relacionados ao grande número de mutantes gerados e à determinação de mutantes equivalentes, mesmo para pequenos programas. Visando a reduzir os custos de aplicação do teste de mutação diversos estudos teóricos e empíricos vêm sendo realizados. Uma síntese dos principais estudos empíricos relacionados ao teste de mutação é apresentada. Esses estudos procuram estabelecer uma estratégia de teste que viabilize a utilização do teste de mutação no teste de produtos comerciais de software. Ilustram-se a atividade de teste e os problemas pertinentes utilizando-se as ferramentas *PokeTool*, *Proteum* e *PROTEUM/IM*, que apóiam, respectivamente, critérios estruturais, o critério Análise de Mutantes e o critério Mutação de Interface. Identificam-se ainda outras iniciativas e esforços da comunidade para a automatização desses critérios.

*Partes deste trabalho foram extraídas de [1] e [2].

1 Introdução

A Engenharia de Software evoluiu significativamente nas últimas décadas procurando estabelecer técnicas, critérios, métodos e ferramentas para a produção de software, em conseqüência da crescente utilização de sistemas baseados em computação em praticamente todas as áreas da atividade humana, o que provoca uma crescente demanda por qualidade e produtividade, tanto do ponto de vista do processo de produção como do ponto de vista dos produtos gerados. A Engenharia de Software pode ser definida como uma disciplina que aplica os princípios de engenharia com o objetivo de produzir software de alta qualidade a baixo custo [3]. Através de um conjunto de etapas que envolvem o desenvolvimento e aplicação de métodos, técnicas e ferramentas, a Engenharia de Software oferece meios para que tais objetivos possam ser alcançados.

O processo de desenvolvimento de software envolve uma série de atividades nas quais, apesar das técnicas, métodos e ferramentas empregados, erros 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. 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 [4].

A atividade de teste consiste de uma análise dinâmica do produto e é uma atividade relevante para a identificação e eliminação de erros que persistem. Do ponto de vista de qualidade do processo, o teste sistemático é uma atividade fundamental para a ascensão ao Nível 3 do Modelo CMM do *Software Engineering Institute — SEI* [5]. Ainda, o conjunto de informação oriundo da atividade de teste é significativo para as atividades de depuração, manutenção e estimativa de confiabilidade de software [3, 6–9]. Salienta-se que a atividade de teste tem sido apontada como uma das mais onerosas no desenvolvimento de software [3, 10, 11]. Apesar deste fato, Myers observa que aparentemente conhece-se muito menos sobre teste de software do que sobre outros aspectos e/ou atividades do desenvolvimento de software [10].

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 [3, 4, 10, 11]. Essas atividades devem ser desenvolvidas ao longo do próprio processo de desenvolvimento de software, e em geral, concretizam-se em três fases de teste: de unidade, de integração e de sistema. O teste de unidade concentra esforços na menor unidade do projeto de software, ou seja, procura 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 [3].

Um ponto crucial na atividade de teste, independentemente da fase, é o projeto e/ou a avali-

ação da qualidade de um determinado conjunto de casos de teste T utilizado para o teste de um produto P, dado que, em geral, é impraticável utilizar todo o domínio de dados de entrada para avaliar os aspectos funcionais e operacionais de um produto em teste. O objetivo é utilizarem-se casos de teste que tenham alta probabilidade de encontrar a maioria dos defeitos com um mínimo de tempo e esforço, por questões de produtividade. Segundo Myers [10], o principal objetivo do teste de software é revelar a presença de erros no produto. Portanto, o teste bem sucedido é aquele que consegue determinar casos de teste para os quais o programa em teste falhe. Tem-se observado que a própria atividade de projeto de casos de teste é bastante efetiva em evidenciar a presença de defeitos de software.

Em geral, os critérios de teste de software são estabelecidos, basicamente, a partir de três técnicas: a funcional, a estrutural e a baseada em erros. Na técnica funcional, os critérios e requisitos de teste são estabelecidos a partir da função de especificação do software; na técnica estrutural, os critérios e requisitos são derivados essencialmente a partir das características de uma particular implementação em teste; e, na técnica baseada em erros, os critérios e requisitos de teste são oriundos do conhecimento sobre erros típicos cometidos no processo de desenvolvimento de software. Observa-se também o estabelecimento de critérios de geração de seqüências de teste baseados em Máquinas de Estados Finito [12, 13]. Esses últimos têm sido aplicados no contexto de validação e teste de sistemas reativos e de sistemas orientados a objetos [14–22].

Segundo Howden [23], 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 da técnica funcional são baseados em especificação e tanto os critérios estruturais quanto baseados em erros são considerados critérios baseados em implementação.

No teste baseado em especificação (ou teste caixa-preta) o objetivo é determinar se o programa satisfaz aos requisitos funcionais e não-funcionais que foram 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 [24]. Entretanto, os critérios de teste baseados em especificação podem ser utilizados em qualquer contexto (procedimental ou orientado a objetos) e em qualquer fase de teste sem a necessidade de modificação. Exemplos desses critérios são: particionamento em classe de equivalência [3], análise do valor limite [3], grafo de causa-efeito [3] e teste baseado em estado [16, 19].

Ao contrário do teste baseado em especificação, o teste baseado em programa (ou teste caixa-branca) requer a inspeção do código fonte e a seleção de casos de teste que exercitem partes do código e não de sua especificação [24].

É importante ressaltar que as técnicas de teste devem ser vistas como complementares e a questão está em 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, ou seja, eficaz e de baixo custo. As técnicas e critérios de teste fornecem ao desenvolvedor uma abordagem sistemática e teoricamente fundamentada, além de constituírem um mecanismo que pode auxiliar a avaliar a qualidade e a adequação da atividade de teste. Critérios de teste podem ser utilizados tanto para auxiliar na geração de conjunto de casos de teste como para auxiliar na avaliação da adequação desses conjuntos.

Dada a diversidade de critérios que têm sido estabelecidos [3,4,10,11,25–35] e reconhecido o caráter complementar das técnicas e critérios de teste [35–44], um ponto crucial que se coloca nessa perspectiva é a escolha e/ou a determinação de uma estratégia de teste, que em última análise passa pela escolha de critérios de teste, de forma que as vantagens de cada um desses critérios sejam combinadas objetivando uma atividade de teste de maior qualidade. Estudos teóricos e empíricos de critérios de teste são de extrema relevância para a formação desse conhecimento, fornecendo subsídios para o estabelecimento de estratégias de baixo custo e alta eficácia. Identificam-se diversos esforços da comunidade científica nessa direção [35, 41–43, 45–51].

Fundamental se faz o desenvolvimento de ferramentas de teste para o suporte à atividade de teste propriamente dita, uma vez que essa atividade é muito propensa a erros, além de improdutiva, se aplicada manualmente, bem como para dar apoio a estudos empíricos que visem a avaliar e a comparar os diversos critérios de teste. Assim, a disponibilidade de ferramentas de teste propicia maior qualidade e produtividade para as atividades de teste. Observam-se na literatura esforços da comunidade científica nessa direção [11, 35, 43, 52–65].

Pode-se observar que os critérios baseados em análise de fluxo de dados [27, 29–33] e o critério Análise de Mutantes (*Mutation Analysis*) [25, 34, 66] têm sido fortemente investigados por diversos pesquisadores em diferentes aspectos [7, 37, 38, 41, 42, 44, 46, 50, 57, 59, 67–77]. 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. Uma forte evidência nessa direção é o esforço alocado pela *Telcordia Technologies* (USA) no desenvolvimento da *xSuds* [78], um ambiente que apóia a aplicação de critérios baseados em análise de fluxo de controle e de dados em programas C e C++.

De uma maneira geral, pode-se classificar as contribuições para a área de Teste de Software em: Estudos Teóricos, Estudos Empíricos e Desenvolvimento de Ferramentas. Este texto aborda esses três aspectos, dando-se ênfase a estudos teóricos e empíricos de critérios baseados em análise de fluxo de dados e do critério Análise de Mutantes para o teste de programas em nível de unidade, assim como as ferramentas que apóiam a aplicação desses critérios. Com esse objetivo em mente, o texto está organizado da seguinte forma: na Seção 2 são introduzidos a terminologia e os conceitos básicos pertinentes à atividade de teste. Na Seção 3 são apresentados os critérios de teste mais difundidos das técnicas funcional, estrutural e baseada em erros e ilustrados os principais aspectos operacionais das ferramentas *PokeTool*, *Proteum* e *PROTEUM/IM* que apóiam a aplicação de critérios estruturais e dos critérios Análise de Mutantese Mutação de Interface, respectivamente. Na Seção 4 são identificados os principais esforços e iniciativas de automatização e na Seção 5 são sintetizados os principais resultados de estudos empíricos envolvendo os critérios apresentados neste texto. Na Seção 6 são apresentadas as conclusões e perspectivas de trabalhos futuros.

2 Terminologia e Conceitos Básicos

A IEEE tem realizado vários esforços de padronização, entre eles para padronizar a terminologia utilizada no contexto de Engenharia de Software. O padrão IEEE número 610.12-1990 [79]

diferencia os termos: **defeito** (*fault*) – passo, processo ou definição de dados incorreto, como por exemplo, uma instrução ou comando incorreto; **engano** (*mistake*) – ação humana que produz um resultado incorreto, com por exemplo, 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. De uma forma geral, os erros são classificados em: **erros computacionais** – o erro provoca uma computação incorreta mas o caminho executado (seqüências de comandos) é igual ao caminho esperado; e **erros de domínio** – o caminho efetivamente executado é diferente do caminho esperado, ou seja, um caminho errado é selecionado.

A atividade de teste é permeada por uma série de limitações [23, 31, 80–82]. Em geral, os seguintes problemas são indecidíveis: dados dois programas, se eles são equivalentes; dados duas seqüências de comandos (caminhos) de um programa, ou de programas diferentes, se eles computam a mesma função; e dado um caminho se ele é executável ou não, ou seja, se existe um conjunto de dados de entrada que levam à execução desse caminho. Outra limitação fundamental é a correção coincidente – o programa pode apresentar, coincidentemente, um resultado correto para um item particular de um dado $d \in D$, ou seja, um particular item de dado ser executado, satisfazer a um requisito de teste e não revelar a presença de um erro.

Diz-se que um programa P com domínio de entrada D é correto com respeito a uma especificação S se $S(d) = P^*(d)$ para qualquer item de dado d pertencente a D, ou seja, se o comportamento do programa está de acordo com o comportamento esperado para todos os dados de entrada. Dados dois programas P_1 e P_2, se $P_1^*(d) = P_2^*(d)$, para qualquer $d \in D$, diz-se que P_1 e P_2 são **equivalentes**. No teste de software, pressupõe-se a existência de um oráculo – o testador ou algum outro mecanismo – que possa determinar, para qualquer item de dado $d \in D$, se $S(d) = P^*(d)$, dentro de limites de tempo e esforços razoáveis. Um oráculo decide simplesmente se os valores de saída são corretos. Sabe-se que o teste exaustivo é impraticável, ou seja, testar para todos os elementos possíveis do domínio de entrada é, em geral, caro e demanda muito mais tempo do que o disponível. Ainda, deve-se salientar que não existe um procedimento de teste de propósito geral que possa ser usado para provar a corretitude de um programa. Apesar de não ser possível, através de testes, provar que um programa está correto, os testes, se conduzidos sistemática e criteriosamente, contribuem para aumentar a confiança de que o software desempenha as funções especificadas e evidenciar algumas características mínimas do ponto de vista da qualidade do produto.

Assim, duas questões são chaves na atividade de teste: "Como os dados de teste devem ser selecionados?" e "Como decidir se um programa P foi suficientemente testado?". Critérios para selecionar e avaliar conjuntos de casos de teste são fundamentais para o sucesso da atividade de teste. Tais critérios devem fornecer indicação de quais casos de teste devem ser utilizados de forma a aumentar as chances de revelar erros ou, quando erros não forem revelados, estabelecer um nível elevado de confiança na correção do programa. Um caso de teste consiste de um par ordenado $(d, S(d))$, no qual $d \in D$ e $S(d)$ é a respectiva saída esperada.

Dados um programa P e um conjunto de casos de teste T, definem-se:

- **Critério de Adequação de Casos de Teste**: predicado para avaliar T no teste de P;
- **Método de Seleção de Casos de Teste**: procedimento para escolher casos de teste para o teste de P.

Existe uma forte correspondência entre métodos de seleção e critérios de adequação de casos de teste pois, dado um critério de adequação C, existe um método de seleção MC que estabelece: selecione T tal que T seja adequado a C. De forma análoga, dado um método de seleção M, existe um critério de adequação CM que estabelece: T é adequado se foi selecionado de acordo com M. Desse modo, costuma-se utilizar o termo "critério de adequação de casos de teste" (ou simplesmente critério de teste) também para designar método de seleção [4, 55]. Dados P, T e um critério C, diz-se que o conjunto de casos de teste T é C-adequado para o teste de P se T preencher os requisitos de teste estabelecidos pelo critério C. Outra questão relevante nesse contexto é dado um conjunto T C_1-adequado, qual seria um critério de teste C_2 que contribuiria para aprimorar T? Essa questão tem sido investigada tanto em estudos teóricos quanto em estudos empíricos.

Em geral, pode-se dizer que as propriedades mínimas que devem ser preenchidas por um critério de teste C são:

1. garantir, do ponto de vista de fluxo de controle, a cobertura de todos os desvios condicionais;
2. requerer, do ponto de vista de fluxo de dados, ao menos um uso de todo resultado computacional; e
3. requerer um conjunto de casos de teste finito.

As vantagens e desvantagens de critérios de teste de software podem ser avaliadas através de estudos teóricos e empíricos. Do ponto de vista de estudos teóricos, esses estudos têm sido apoiados principalmente por uma relação de inclusão e pelo estudo da complexidade dos critérios [31, 81, 83]. A relação de inclusão estabelece uma ordem parcial entre os critérios, caracterizando uma hierarquia entre eles. 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 existir um programa P e um conjunto T_2 C_2-adequado que não seja C_1-adequado. A complexidade é definida como o número máximo de casos de teste requeridos por um critério, no pior caso. No caso dos critérios baseados em fluxo de dados, esses têm complexidade exponencial, o que motiva a condução de estudos empíricos para determinar o custo de aplicação desses critérios do ponto de vista prático. Mais recentemente, alguns autores têm abordado do ponto de vista teórico a questão de eficácia de critérios de teste, e têm definido outras relações, que captem a capacidade de revelar erros dos critérios de teste [37, 38, 84, 85].

Do ponto de vista de estudos empíricos, três aspectos costumam ser avaliados: custo, eficácia e *strength* (ou dificuldade de satisfação) [40, 41, 46, 71, 76, 86]. O fator custo reflete o esforço necessário para que o critério seja utilizado; em geral é medido pelo número de casos de teste necessários para satisfazer o critério. A eficácia refere-se à capacidade que um critério possui

de detectar a presença de erros. O fator *strength* refere-se à probabilidade de satisfazer-se um critério tendo sido satisfeito um outro critério [41].

Uma atividade muito citada na condução e avaliação da atividade de teste é a análise de cobertura, que consiste basicamente em determinar o percentual de elementos requeridos por um dado critério de teste que foram exercitados pelo conjunto de casos de teste utilizado. A partir dessa informação o conjunto de casos de teste pode ser aprimorado, acrescentando-se novos casos de teste para exercitar os elementos ainda não cobertos. Nessa perspectiva, é fundamental o conhecimento sobre as limitações teóricas inerentes à atividade de teste, pois os elementos requeridos podem ser não executáveis, e em geral, determinar a não executabilidade de um dado requisito de teste envolve a participação do testador.

3 Técnicas e Critérios de Teste

Conforme mencionado anteriormente, para se conduzir e avaliar a qualidade da atividade de teste têm-se as técnicas de teste funcional, estrutural e baseada em erros. Tais técnicas diferenciam-se pela origem da informação utilizada na avaliação e construção dos conjuntos de casos de teste [4]. Neste texto apresentam-se com mais detalhes as duas últimas técnicas, mais especificamente os critérios Potenciais-Usos [4], o critério Análise de Mutantes [34] e o critério Mutação de Interface [35], e as ferramentas de suporte *PokeTool* [60, 61, 87], *Proteum* [62] e *PROTEUM/IM* [35, 88]. Através desses critérios, ilustram-se os principais aspectos pertinentes à atividade de teste de cobertura de software. Para propiciar uma visão mais abrangente, apresentam-se primeiramente uma visão geral da técnica funcional e os critérios mais conhecidos dessa técnica. O programa *identifier* (Figura 1) será utilizado para facilitar a ilustração dos conceitos desenvolvidos neste texto.

3.1 Técnica Funcional

O teste funcional também é conhecido como teste caixa preta [10] pelo fato de tratar 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. Na técnica de teste funcional são verificadas as funções do sistema sem se preocupar com os detalhes de implementação.

O teste funcional envolve dois passos principais: identificar as funções que o software deve realizar e criar casos de teste capazes de checar se essas funções estão sendo realizadas pelo software [3]. As funções que o software deve possuir são identificadas a partir de sua especificação. Assim, uma especificação bem elaborada e de acordo com os requisitos do usuário é essencial para esse tipo de teste.

Alguns exemplos de critérios de teste funcional são [3]:

- **Particionamento em Classes de Equivalência**: a partir das condições de entrada de dados identificadas na especificação, divide-se o domínio de entrada de um programa em classes de equivalência válidas e inválidas. Em seguida seleciona-se o menor número

```
/*********************************************************************************
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).

possível de casos de teste, baseando-se na hipótese 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 mais sistematicamente 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, pois nesses pontos 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. Este 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 é construído 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.

Um dos problemas relacionado 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. Por outro lado, para a aplicação desses critérios é essencial apenas que se identifiquem as entradas, a função a ser computada e a saída do programa, o que os tornam aplicáveis praticamente em todas fases de teste (unidade, integração e sistema) [35].

A título de ilustração, considerando o programa *identifier* e o critério Particionamento em Classes de Equivalência, são identificadas na Tabela 1 as condições de entrada e classes de equivalência válidas e inválidas. A partir dessas classes o seguinte conjunto de casos de teste poderia ser elaborado: 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, lança-se mão de critérios estruturais de teste, apresentados a seguir, como critérios de adequação ou critérios de cobertura para se analisar questões como essas, propiciando a quantificação e a qualificação da atividade de teste de acordo com o critério escolhido. Quanto mais rigoroso o critério utilizado e se erros não forem revelados, maior a confiança no produto em desenvolvimento.

3.2 Técnica Estrutural

A técnica estrutural apresenta uma série de limitações e desvantagens decorrentes das limitações inerentes às atividades de teste de programa enquanto estratégia de validação [31, 80, 81, 89]. Esses aspectos introduzem sérios problemas na automatização do processo de validação de

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)

software [MAL91]. Independentemente dessas desvantagens, essa técnica é vista como complementar à técnica funcional [3] e informações obtidas pela aplicação desses critérios têm sido consideradas relevantes para as atividades de manutenção, depuração e confiabilidade de software [3, 6–9].

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 ($G = (N, E, s)$) 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 $s \in N$ 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 componentes que devem ser executados, caracterizando assim o teste estrutural. Considere o programa *identifier*. Na Figura 1 identifica-se a caracterização dos blocos de comandos através 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* [64].

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 onde 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.

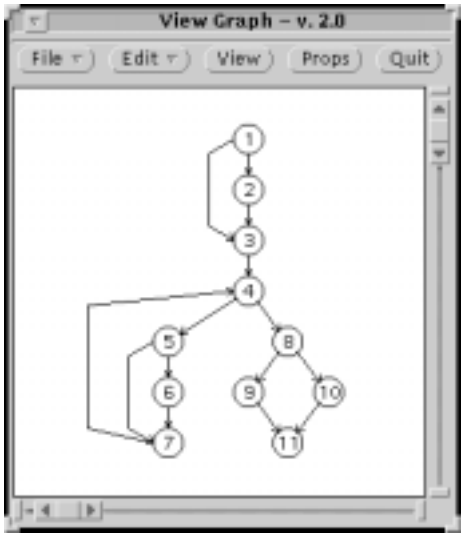


Figura 2: Grafo de Fluxo de Controle do Programa *identifier* gerado pela *ViewGraph*.

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

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

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

- **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 **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 [3]. Outros critérios dessa categoria são: **Cobertura de Decisão**; **Cobertura de Condição**; **Cobertura de Condições Múltiplas**; **LCSAJ** (*Linear Code Sequence and Jump*) [90]; o critério **Boundary-Interior** [91]; e a família de critérios ***K*-tuplas** requeridas de Ntafos [32].

- **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 [31]. Exemplos dessa classe de critérios são os **Critérios de Rapps e Weyuker** [30, 31] e os **Critérios Potenciais-Usos** [4]. Visto que tais critérios serão utilizados nos estudos comparativos a serem realizados durante o desenvolvimento deste trabalho, as próximas seções destinam-se a descrevê-los mais detalhadamente.

- **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 [3].

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 [43].

Um problema relacionado ao teste estrutural é a impossibilidade, em geral, de se determinar automaticamente se um caminho é ou não executável, 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 [92]. Assim, é preciso a intervenção do testador para determinar quais são os caminhos não executáveis para o programa sendo testado.

3.2.1 Critérios Baseados em Fluxo de Dados

Em meados da década de 70 surgiram os critérios baseados em análise de fluxo de dados [27], os quais utilizam informações do fluxo de dados para derivar os requisitos de teste. Uma característica comum dos critérios baseados em fluxo de dados é que eles requerem que sejam testadas as interações que envolvam definições de variáveis e subseqüentes referências a essas definições [27, 29, 31–33]. Uma motivação para a introdução dos critérios baseados na análise de fluxo de dados foi a indicação de que, mesmo para programas pequenos, o teste baseado unicamente no fluxo de controle não ser eficaz para revelar a presença até mesmo de erros simples e triviais. A introdução dessa classe de critérios procura fornecer uma hierarquia entre os critérios Todos-Arcos e Todos-Caminhos, procurando tornar o teste mais rigoroso, já que o teste de Todos-Caminhos é, em geral, impraticável. Segundo Ural [33], esses critérios são mais

adequados para certas classes de erros, como erros computacionais, uma vez que dependências de dados são identificadas, e portanto, segmentos funcionais são requeridos como requisitos de teste.

Rapps e Weyuker propuseram o Grafo **Def-Uso** (*Def-Use Graph*) que consiste em uma extensão do grafo de programa [30, 31]. Nele são adicionadas informações a respeito do fluxo de dados do programa, caracterizando associações entre pontos do programa onde é atribuído um valor a uma variável (chamado de **definição** da variável) e pontos onde esse valor é utilizado (chamado de referência ou **uso** de 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 [4], 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 através de parâmetros pode ser por valor, referência ou por nome [93]. 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.

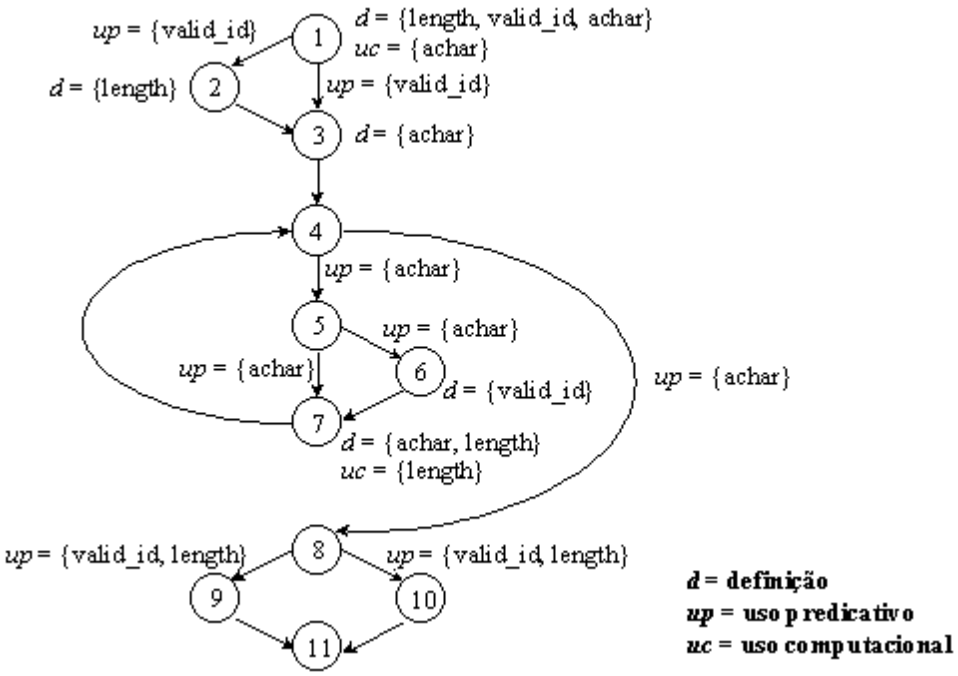


Figura 3: Grafo Def-Uso do Programa *identifier*.

O critério mais básico dos critérios baseados em análise de fluxo de dados é o critério Todas-

Definições (*all-defs*) e faz parte da família de critérios definidos por Rapps e Weyuker [31]. Entre os critérios dessa família o critério Todos-Usos (*all-uses*) 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.

Por 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, e qualquer caminho completo que o inclua também é não executável. Se qualquer um dos demais caminhos for exercitado, o requisito de teste estaria sendo satisfeito, e para satisfazer o critério Todas-Definições esta análise teria que ser feita para toda definição que ocorre no programa. Em relação ao critério Todos-Usos, com respeito à mesma definição, seriam requeridas as seguinte 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) seria suficiente para exercitá-la. Esta mesma análise deveria 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.

Com a introdução do conceito **potencial-uso** são definidos vários critérios, denominados critérios **Potenciais-Usos** [4], cujos elementos requeridos são caracterizados independentemente da ocorrência explícita de uma referência – um uso – a uma determinada definição; se um uso dessa definição pode existir, ou seja, existir um caminho livre de definição até um certo nó ou arco – um potencial-uso – a **potencial-associação** entre a definição e o potencial-uso é caracterizada, e eventualmente requerida. Na realidade, pode-se dizer que, com a introdução do conceito potencial-uso, procura-se explorar todos os possíveis efeitos a partir de uma mudança de estado do programa em teste, decorrente de definição de variáveis em um determinado nó i. Da mesma forma como os demais critérios baseados na análise de 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, denominado de Grafo Def [4]. Por exemplo, as potenciais-associações (1,6,`length`) e (7,6,`length`) são requeridas pelo critério Todos-Potenciais-Usos [4], mas não seriam requeridas pelos demais critérios de fluxo de dados que não fazem uso do conceito potencial-uso. Observe-se 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.

- **Todos-Potenciais-Usos**: requer, basicamente, para todo nó i e para toda variável x, para a qual existe uma definição em i, que pelo menos um caminho livre de definição com relação à variável (c.r.a) x do nó i para todo nó e para todo arco possível de ser alcançado a partir de i por um caminho livre de definição c.r.a. x seja exercitado.

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-Defs e Todos-Arcos, diz-se que tais critérios são incomparáveis [31]. Deve-se observar que os critérios Potenciais-Usos são os únicos critérios baseados em análise de 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 análise de fluxo de dados os inclui. Um aspecto relevante é que alguns dos critérios Potenciais-Usos "*bridge 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, uma vez que os requisitos de teste por eles exigidos limitam-se ao escopo da unidade. Vários esforços de pesquisa no sentido de estender o uso de critérios estruturais para o teste de integração podem ser identificados. Haley e Zweben propuseram um critério para selecionar caminhos em um módulo que deveria ser testado novamente na fase de integração com base em sua interface [94]. Linnenkugel e Müllerburg apresentaram uma série de critérios que estendem os critérios baseados em fluxo de controle e em fluxo de dados para o teste de integração [68]. Harrold e Soffa propuseram uma técnica para determinar as estruturas de definição-uso interprocedurais permitindo a aplicação dos critérios baseados em análise de fluxo de dados em nível de integração [95]. Jin

e Offutt definiram alguns critérios baseados em uma classificação de acoplamento entre módulos [96]. Vilela, com base no conceito de potencial-uso, estendeu os critérios Potenciais-Usos para o teste de integração [97].

3.2.2 A Ferramenta de Teste *PokeTool*

Várias são as iniciativas de desenvolvimento de ferramentas de teste para apoiar a aplicação de critérios de teste [11, 35, 43, 52–65]. Para ilustrar os conceitos abordados acima será utilizada a ferramenta *PokeTool* (*Potential Uses Criteria Tool for Program Testing*) [60, 61], desenvolvida na Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas – UNICAMP. Essa ferramenta apóia a aplicação dos critérios Potenciais-Usos e também de outros critérios estruturais como Todos-Nós e Todos-Arcos. Inicialmente foi desenvolvida para o teste de unidade de programas escritos em C [61], mas atualmente, devido à sua característica de multi-linguagem, já existem configurações para o teste de programas em Cobol e FORTRAN [98, 99]. A Figura 4 mostra a tela principal da ferramenta e as funções fornecidas.

A ferramenta *PokeTool* é orientada a **sessão de trabalho**. O termo sessão trabalho ou de teste é utilizado para designar as atividades envolvendo um teste. O teste pode ser realizado em etapas onde 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** [100], 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. 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 Figura 5 mostra a criação de uma sessão de teste para o programa *identifier* utilizando todos os critérios apoiados pela ferramenta.

A *PokeTool* 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 através de interface gráfica ou linha de comando (*shell scripts*).

Considerando-se os critérios Todos-Arcos e Todos-Potenciais-Usos e o programa *identifier*, as Tabelas 3 e 4 trazem os elementos requeridos por esses critérios, respectivamente. Introduz-se a notação $\langle i, (j,k), \{v_1, \ldots, v_n\}\rangle$ para representar o conjunto de associações $\langle i, (j,k), \{v_1\}\rangle$, ..., $\langle i, (j,k), \{v_n\}\rangle$; ou seja, $\langle i, (j,k), \{v_1, \ldots, v_n\}\rangle$ indica que existe pelo menos um caminho livre de definição c.r.a $v_1, \ldots, v_n$ do nó i ao arco (j,k). Observe-se que podem existir outros caminhos livres de definição c.r.a algumas das variáveis $v1, \ldots, v_n$ mas que não sejam, simultaneamente, livres de definição para todas as variáveis $v_1, \ldots, v_n$.

Utilizando o conjunto de casos de teste T_0 = {(a1, Válido), (2B3, Inválido), (Z-12, Inválido), (A1b2C3d, Inválido)} gerado anteriormente procurando satisfazer o critério Particionamento em Classes de Equivalência, observa-se qual a cobertura obtida em relação aos critérios

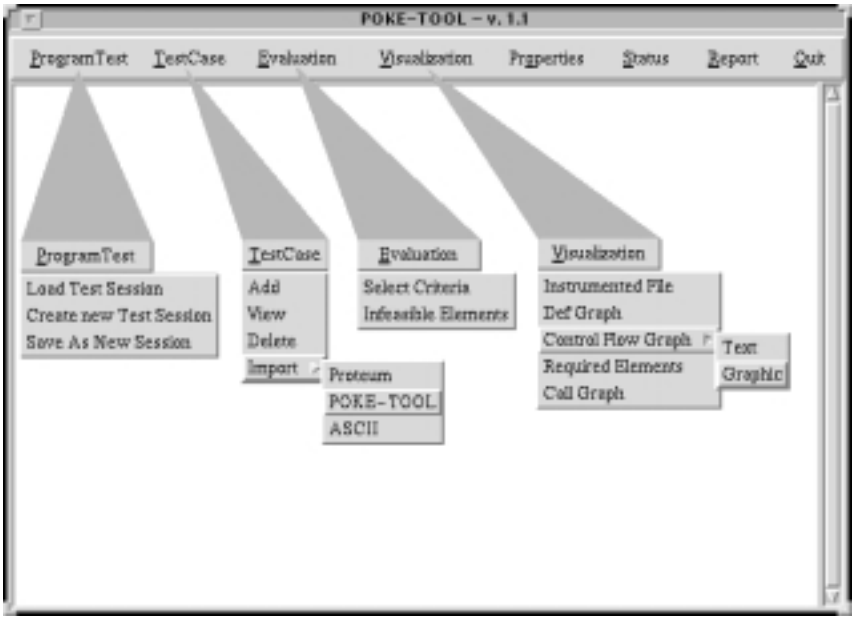


Figura 4: Opções disponíveis na ferramenta *PokeTool*.

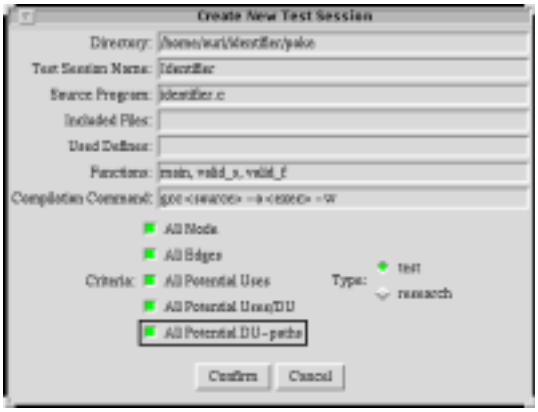


Figura 5: Tela para criar uma sessão de teste na *PokeTool*.

Tabela 3: Elementos requeridos pelo critério Todos-Potenciais-Usos.

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

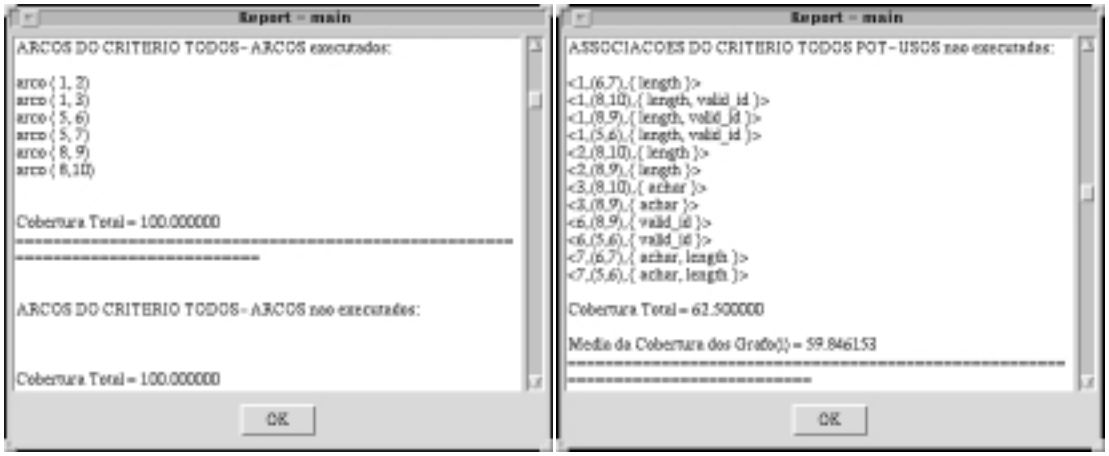
Todos-Arcos e Todos-Potenciais-Usos (Figura 6(a) e Figura 6(b), respectivamente). Ainda na Figura 6(b), são ilustrados para o critério Todos-Potenciais-Usos os elementos requeridos e não executados quando a cobertura é inferior a 100%.

Observa-se que somente com os casos de teste funcionais foi possível cobrir o critério

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) $\langle 7, (5,6), \{\texttt{achar}, \texttt{length}\}\rangle$

Todos-Arcos ao passo que para se cobrir o critério Todos-Potenciais-Usos ainda é necessário analisar as associações que não foram executadas. Deve-se ressaltar que o conjunto T_0 é Todos-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.



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

Figura 6: Relatórios gerados pela *PokeTool* em relação ao programa *identifier*.

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) $\langle 1,(8,9),\{\texttt{length},\texttt{valid\_id}\}\rangle$	$\times$	$\times$	$\times$	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) $\langle 6,(8,9),\{\texttt{valid\_id}\}\rangle$	$\times$	$\times$	$\times$
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) $\langle 2,(8,10),\{\texttt{length}\}\rangle$	$\times$	$\times$	$\times$	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-se 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 [101]. 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 na próxima seção), que do ponto de vista teórico é incomparável com os critérios baseados em fluxo de dados, mas em geral de maior custo de aplicação.

3.3 Teste Baseado 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** (*Error Seeding*) [25] e **Análise de Mutantes** (*Mutation Analysis*) [34] são critérios típicos que se concentram em erros. Neste texto dá-se ênfase ao critério Análise de Mutantes.

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 [102]. O critério Análise

de Mutantes 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, através da execução de P, as diferenças de comportamento existentes entre P e seus mutantes [66].

A seguir dá-se uma visão geral do critério Análise de Mutantes e da ferramenta de apoio *Proteum*, desenvolvida no ICMC-USP [62]. Informações mais detalhadas sobre a Análise de Mutantes e sobre a ferramenta *Proteum* podem ser obtidas em [1, 62, 103].

3.4 O Critério Análise de Mutantes

Um dos primeiros artigos que descrevem a idéia de teste de mutantes foi publicado em 1978 [34]. 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 através de pequenos desvios sintáticos que, embora não causem erros sintáticos, alteram a semântica do programa e, conseqüentemente, conduzem o programa 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 [104].

Uma outra hipótese explorada na aplicação do critério Análise de Mutantes é o **efeito de acoplamento** (*coupling effect*) [34], a qual assume que erros complexos estão relacionados a erros simples. Assim sendo, espera-se, e alguns estudos empíricos já confirmaram esta hipótese [105, 106], 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) [46].

A seguir, 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 [34].

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 [25].

Um ponto importante destacado por DeMillo [52] é que a Análise de Mutantes fornece uma medida objetiva do nível de confiança da adequação dos casos de teste analisados através 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 [43].

Um dos maiores problemas para a aplicação do critério Análise de Mutantes 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. A utilização de arquiteturas de hardware avançadas para diminuir o tempo de execução dos mutantes [107–110] e o uso da análise estática de anomalias de fluxo de dados para reduzir o número de mutantes gerados [111] são algumas dessas estratégias. Além disso, critérios alternativos derivados da Análise de Mutantes também foram criados com o intuito de reduzir o custo a ela associado: **Mutação Aleatória** (*Randomly Selected X% Mutation*), **Mutação Restrita** (*Constrained Mutation*) e **Mutação Seletiva** (*Selective Mutation*). Tais critérios procuram selecionar apenas um subconjunto do total de mutantes gerados, reduzindo o custo associado, mas com a expectativa de não se reduzir a eficácia do critério.

3.4.1 A Ferramenta de Teste *Proteum*

Como ressaltado anteriormente, a aplicação de critérios de teste sem o apoio de uma ferramenta de software é propensa a erros. Várias são as iniciativas de desenvolvimento de ferramentas de apoio à aplicação do critério Análise de Mutantes [35, 43, 52, 53, 62, 103, 112]. A *Proteum* [62, 103], desenvolvida no ICMC-USP, é a única ferramenta que apóia o teste de mutação para programas C existente atualmente. Além disso, devido a características de multi-linguagem, ela também pode ser configurada para o teste de programas escritos em outras linguagens. A *Proteum* está disponível para os sistemas operacionais SunOS, Solaris e Linux. Na Figura 7 é apresentada a tela principal da ferramenta bem como as funções disponíveis. Basicamente, a ferramenta *Proteum* oferece ao testador recursos para, através da aplicação do critério Análise de Mutantes, avaliar a adequação de 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.

Os recursos oferecidos pela ferramenta (Figura 7) 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) [35, 62]. 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. Através desse 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 [62, 103].

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 (Figura 8) [62]: mutação de comandos (*statement mutations*), mutação de operadores (*operator mutations*), mutação de variáveis (*variable mutations*) e mutação de constantes (*constant mutations*). 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 *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. Para o programa *identifier*, o processo de criação de uma sessão de teste utilizando a interface gráfica é ilustrado na Figura 9 abaixo.

Uma sessão de teste com o apoio das ferramentas *Proteum* e *PokeTool* pode ser conduzida através de uma interface gráfica ou através de *scripts*. A interface gráfica permite ao usuário

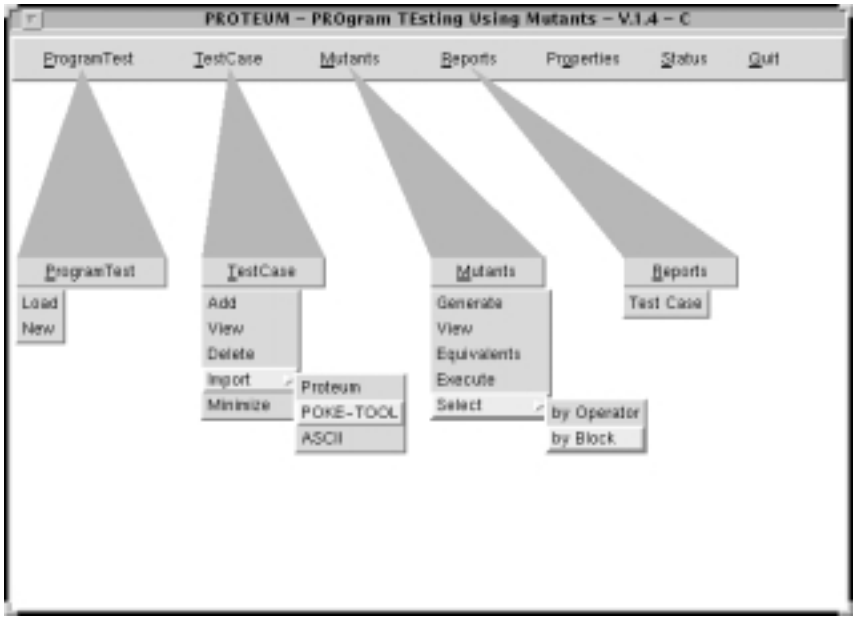


Figura 7: Opções disponíveis na ferramenta *Proteum*.

Figura 8: Classes e operadores de mutação existentes na *Proteum*.

iniciante explorar e aprender os conceitos de teste relacionados ao critério em uso e da própria ferramenta. Além disso, oferece melhores recursos para a visualização dos casos de teste e dos requisitos de teste, por exemplo dos mutantes, no caso da *Proteum*, facilitando algumas tarefas como a identificação dos mutantes equivalentes.

Conduzir uma sessão de teste através da interface gráfica é provavelmente mais fácil, porém menos flexível do que quando se utiliza a chamada direta aos programas que compõem as ferramentas. A interface gráfica depende de constante interação do testador, ao passo que a utilização de *scripts* possibilita a execução de longas sessões de teste em *batch*. O usuário pode construir um programa especificando o teste a ser realizado e a ferramenta simplesmente

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.

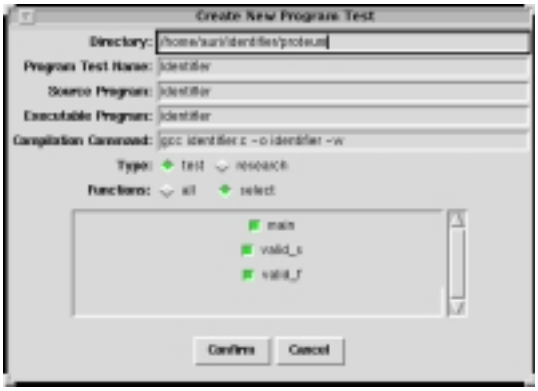


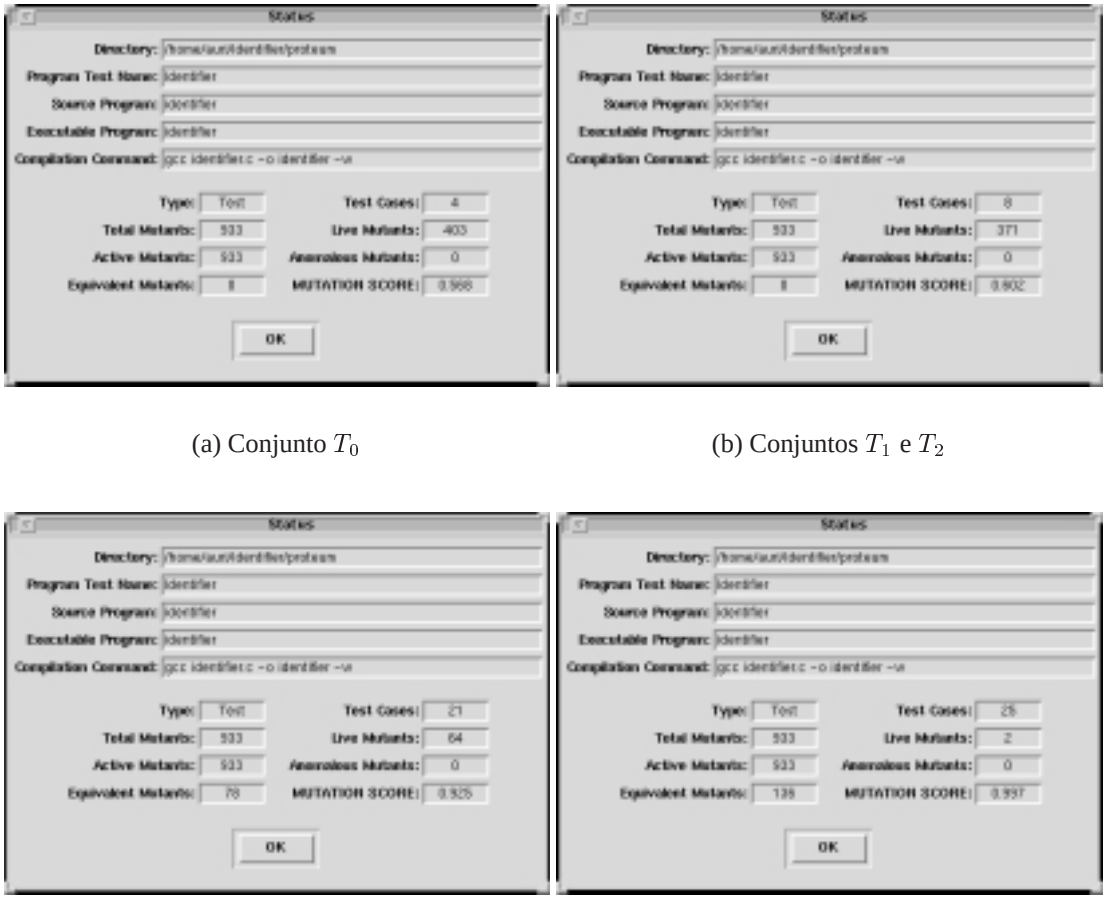
Figura 9: Criando uma sessão de teste para o programa *identifier* na *Proteum*.

executa esse programa, permitindo que se economize tempo na atividade de teste devido à redução do número de interações com a ferramenta. Por outro lado, a elaboração de *scripts* exige um esforço de programação e completo domínio tanto dos conceitos sobre o teste baseado em mutação quanto dos próprios programas que compõem as ferramentas, devendo ser utilizado pelo testador mais experiente [35]. *Scripts* de teste têm se mostrado de grande utilidade na condução de estudos empíricos, onde uma mesma seqüência de passos deve ser executada várias vezes até que os resultados obtidos sejam significantes do ponto de vista estatístico.

A seguir, será avaliada a adequação da atividade de teste do programa *identifier*, realizada até este ponto com o uso da ferramenta *PokeTool*, em relação ao critério Análise de Mutantes, com o apoio da ferramenta *Proteum*; ou seja, será 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 10(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 10(b)). Observa-se 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), ([, Inválido), ({, Inválido), (x/, Inválido), (x:, Inválido), (x18, Válido), (x[[, Inválido), (x{{, Inválido)}. A Figura 11 ilustra dois dos mutantes vivos que foram analisados. O mutante da Figura 11 (a) é um mutante equivalente e o mutante da Figura 11 (b) é um mutante que morre com o caso de teste ([, Inválido), presente em T_3. Os pontos nos quais as mutações foram aplicadas está destacado em negrito. A Figura 10(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 10(c)).



(a) Conjunto T_0

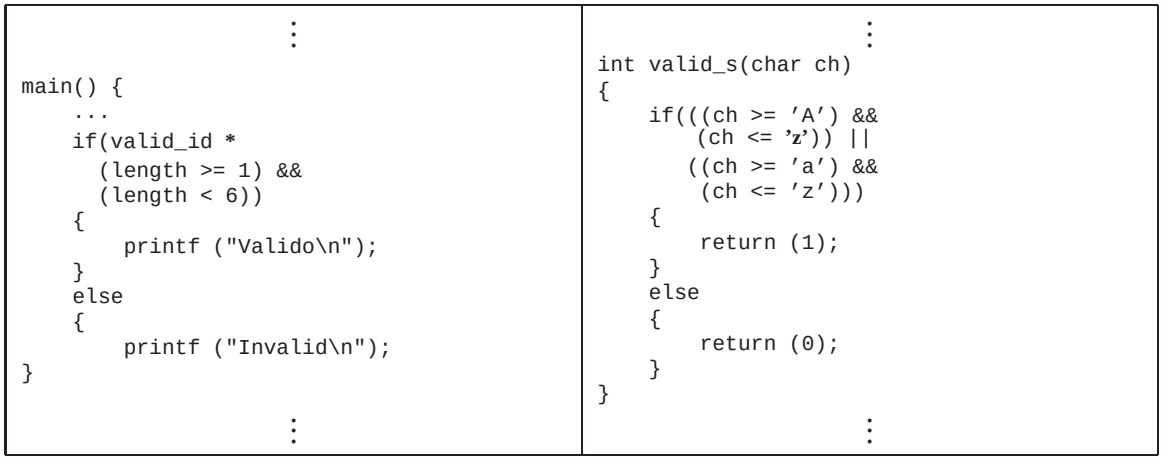
(b) Conjuntos T_1 e T_2

(c) Conjunto T_3

(d) Conjuntos T_4

Figura 10: Telas de *status* da sessão de teste da ferramenta *Proteum*.

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



```
                    ⋮

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

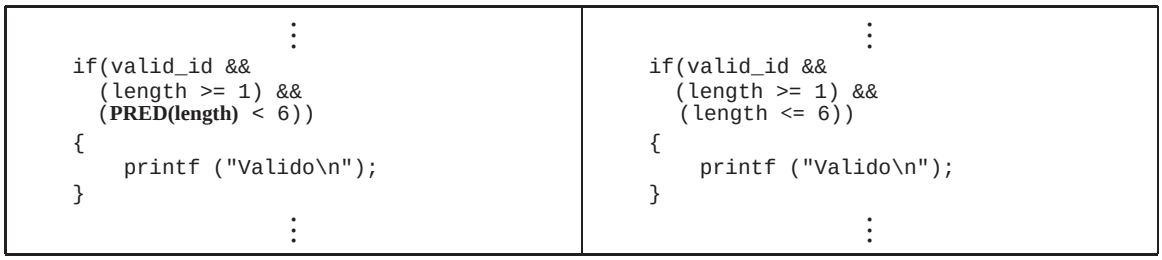
(a) – Mutante equivalente

```
                    ⋮

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

(b) – Mutante não-equivalente

Figura 11: 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 12: Mutantes vivos do programa *identifier*.

processo, mais quatro casos de teste foram construídos ($T_4 = T_3 \cup$ {(@, Inválido), (‘, Inválido), (x@, Inválido), (x‘, Inválido)}). A Figura 10(d) mostra o resultado final obtido. Observa-se que ainda restaram dois mutantes vivos (Figura 12 (a) e (b)). Esses mutantes são mutantes *error-revealing* e um deles representa o programa correto: Figura 12 (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*, Figura 12 (a) e (b), foram gerados pelos operadores de mutação ORRN e VTWD e que necessariamente o erro presente na versão do programa *identifier* será revelado ao elaborar-se qualquer caso de teste que seja capaz de distinguir o comportamento desses mutantes e a versão do programa *identifier* em teste. Os mutantes Figura 12 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 13). 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

```
/*****************************************************************************
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 13: Versão do programa *identifier* corrigida.

340 mutantes, representando uma economia de aproximadamente 63%. Os operadores de mutação ilustrados na Tabela 8 constituem um conjunto de operadores essenciais para a linguagem C [77], ou seja, um conjunto de casos de teste que seja capaz de distinguir os mutantes gerados por esses 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-se 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.

3.5 O Critério Mutação de Interface

O critério Mutação de Interface [35] é uma extensão da Análise de Mutantes e preocupa-se em assegurar que as interações entre unidades sejam testadas. Assim, o objetivo do critério

Mutação de Interface é inserir perturbações nas conexões entre duas unidades.

Utilizando o raciocínio do teste de mutação, casos de teste capazes de distinguir mutantes de interface também devem ser capazes de revelar grande parte dos erros de integração. Essa afirmação depende, evidentemente, de quais mutantes são utilizados ou, em outras palavras, quais operadores de mutação são aplicados [35].

Segundo Haley e Zweben [94], os erros de integração podem ser classificados em erros de integração de domínio e computacional. Dada uma função *F* que chama *G*, o primeiro ocorre quando um erro de domínio[1] em *G* causa uma saída incorreta em *F*. O segundo ocorre quando um erro computacional em *G* produz um valor incorreto que é passado para *F* que, por sua vez, produz uma saída incorreta. Em ambos os casos existe algum valor incorreto sendo passado entre as unidades, o que resulta em uma saída incorreta. Considerando esses aspectos é possível classificar os erros de integração em três categorias.

Considere um programa P e um caso de teste t para P. Suponha que em P existam funções *F* e *G* tal que *F* chama *G*. Considere $SI(G)$ como o conjunto de valores passados para *G* e $SO(G)$ os valores retornados por *G*. Ao executar P com o caso de teste t, um erro de integração é identificado na chamada de *G* a partir de *F* quando [35]:

- Erro Tipo 1 (Figura 14 (a)): os valores contidos em $SI(G)$ não são os esperados por *G*, influenciando a produção de saídas erradas antes do retorno de *G*. Esse tipo de erro ocorre, por exemplo, quando uma função é chamada com parâmetros incorretos fazendo com que a função chamada produza uma saída incorreta;

- Erro Tipo 2 (Figura 14 (b)): os valores contidos em $SI(G)$ não são os esperados por *G*, desse modo, $SO(G)$ assume valores errados fazendo com que *F* produza uma saída incorreta após o retorno de *G*. Um erro desse tipo pode ocorrer, por exemplo, quando um parâmetro incorreto passado para a função é utilizado para calcular o valor de retorno; e

- Erro Tipo 3 (Figura 14 (c)): os valores contidos em $SI(G)$ são os esperados por *G*, mas valores incorretos em $SO(G)$ são produzidos dentro de *G* e esses valores fazem com que *F* produza um resultado incorreto após o retorno de *G*. Esse tipo de erro pode ocorrer se uma função é chamada com todos os parâmetros corretos, mas internamente ela realiza um cálculo incorreto produzindo um valor de retorno não esperado que, posteriormente, leva a um resultado incorreto.

Percebe-se que esta classificação dos tipos de erros é abrangente e não especifica o local do defeito que causa o erro. Ela simplesmente considera a existência de um valor incorreto entrando ou saindo de uma função chamada. Isso exclui, por exemplo, o caso em que $SI(G)$ tem os valores esperados mas um erro dentro de *G* produz uma saída incorreta antes do retorno de *G*. Neste caso, não existe nenhuma propagação de erro através da conexão *F*-*G* e esse tipo de erro deveria ser detectado no teste de unidade.

[1] Um erro de domínio ocorre quando um caminho incorreto é executado; um erro computacional ocorre quando o caminho correto é executado mas o valor computado é incorreto.

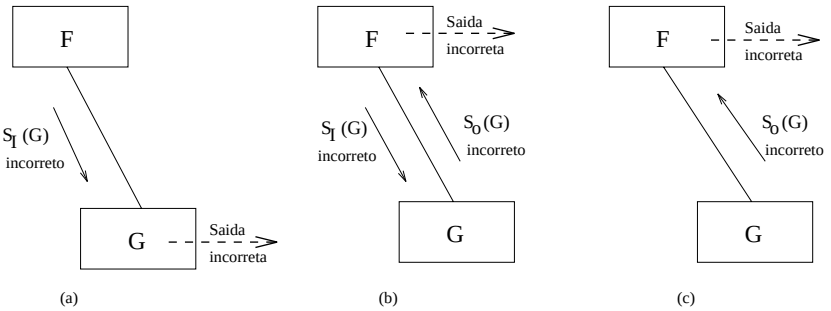


Figura 14: Tipos de Erros de Integração [35]: (a) Erro Tipo 1, (b) Erro Tipo 2, (c) Erro Tipo 3.

Operadores de mutação destinados ao teste de unidade possuem semelhanças e diferenças em relação aos operadores de mutação destinados ao teste de integração. A idéia básica de ambos é a mesma, ou seja, introduzir modificações sintáticas no programa em teste transformando-o em programas similares: mutantes. Por outro lado, os operadores de mutação de interface estão relacionados a uma conexão entre duas unidades. Desse modo, os operadores, quando utilizados para testar uma conexão *F*-*G*, são aplicados de dois modos diferentes: 1) nos pontos onde *F* chama *G*; e 2) nos pontos dentro de *G* relacionados com a interface de comunicação entre as unidades. No segundo caso é necessário um mecanismo adicional que permita identificar o ponto a partir do qual *G* foi chamada. Visto que somente a conexão *F*-*G* está sendo testada, a mutação só deve estar ativa se *G* foi chamada a partir de *F*. De outro modo, *G* deve se comportar da mesma forma que no programa original. Essa característica pode requerer, em linguagens tais como C, que a decisão de aplicar ou não a mutação seja tomada em tempo de execução [35].

3.5.1 A Ferramenta de Teste *PROTEUM/IM*

A ferramenta *PROTEUM/IM* é semelhante à ferramenta *Proteum*. A arquitetura e implementação de ambas são similares (em [35, 103] podem ser encontradas informações detalhadas a respeito da arquitetura dessas ferramentas). A diferença existente entre elas é, basicamente, o conjunto de operadores de mutação que cada uma utiliza e o fato de que a *Proteum* destina-se ao teste de unidade enquanto que a *PROTEUM/IM* oferece características para testar a conexão entre as unidades, ou seja, teste de integração [113].

Dada uma conexão entre duas unidades *F* e *G* (*F* chamando *G*), existem dois grupos de mutações: os operadores do primeiro grupo (Grupo-I) aplicam mutações no corpo da função *G*, por exemplo, incrementando a referência a um parâmetro formal. Os operadores do segundo grupo (Grupo-II) realizam mutações nos pontos onde a unidade *F* faz chamadas à unidade *G*, por exemplo, incrementando o argumento sendo passado para *G*. A ferramenta *PROTEUM/IM* possui um conjunto de 33 operadores de mutação: 24 do Grupo-I e 9 do Grupo-II. A Figura 15 ilustra a tela de geração de mutantes da *PROTEUM/IM*. A Tabela 7 apresenta a descrição de alguns dos operadores de interface.

É importante observar que a aplicação do critério Mutação de Interface está relacionada à conexão entre duas unidades e não a uma unidade somente. A Figura 16 mostra o que acontece,

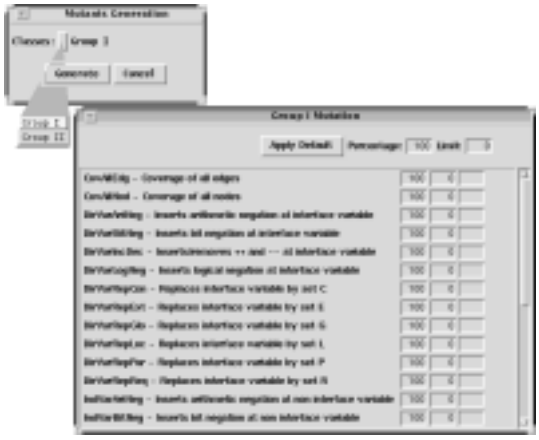


Figura 15: Grupos e operadores de mutação da *PROTEUM/IM*.

Tabela 7: Exemplos de operadores de mutação de interface.

Operador	**Descrição**
II-ArgAriNeg	Acrescenta negação aritmética antes de argumento.
I-CovAllNod	Garante cobertura de nós.
I-DirVarBitNeg	Acrescenta negação de bit em variáveis de interface.
I-IndVarBitNeg	Acrescenta negação de bit em variáveis não de interface.
I-IndVarRepGlo	Troca variável não de interface por variáveis globais utilizadas na função chamada.
I-IndVarRepExt	Troca variável não de interface por variáveis globais não utilizadas na função chamada.
I-IndVarRepLoc	Troca variável não de interface por variáveis locais, declaradas na função chamada.
I-IndVarRepReq	Troca variável não de interface por constantes requeridas.

por exemplo, quando a unidade *F* faz duas chamadas à unidade *G*. Nesse caso, os mutantes associados a primeira chamada só poderão ser mortos por casos de teste que os exercitem a partir desse ponto de chamada. Para os mutantes do Grupo-II isso é trivial visto que as mutações são realizadas exatamente nos locais onde *F* chama *G*. Entretanto, para os operadores do Grupo-I isso não ocorre. Visto que a mutação é realizada no corpo da unidade *G*, é necessário saber qual chamada será usada de modo que o mutante só possa ser morto a partir daquele ponto de chamada. Caso a unidade seja chamada a partir de outro ponto, ela deve se comportar como no programa original, ou seja, a mutação não deve ser habilitada [88].

Para ilustrar a aplicação do critério Mutação de Interface, a seguir, da mesma forma como anteriormente, a ferramenta *PROTEUM/IM* será utilizada para avaliar a adequação dos conjuntos de casos de teste adequados aos critérios Particionamento em Classes de Equivalência (T_0), Todos-Potenciais-Usos (T_2) e Análise de Mutantes (T_5). As Figuras 17(b), 17(c) e 17(d) ilustram as coberturas obtidas para esses conjuntos de casos de teste, respectivamente.

Como pode ser observado, utilizando-se todos os operadores de mutação de interface, foram gerados 456 mutantes. A título de ilustração, a Figura 18 mostra dois dos mutantes de interface gerados para o programa *identifier*. O mutante da Figura 18 (a) foi gerado pelo operador I-

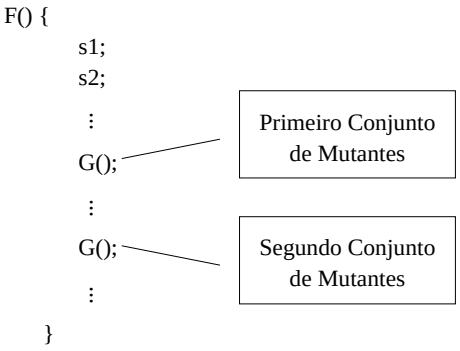
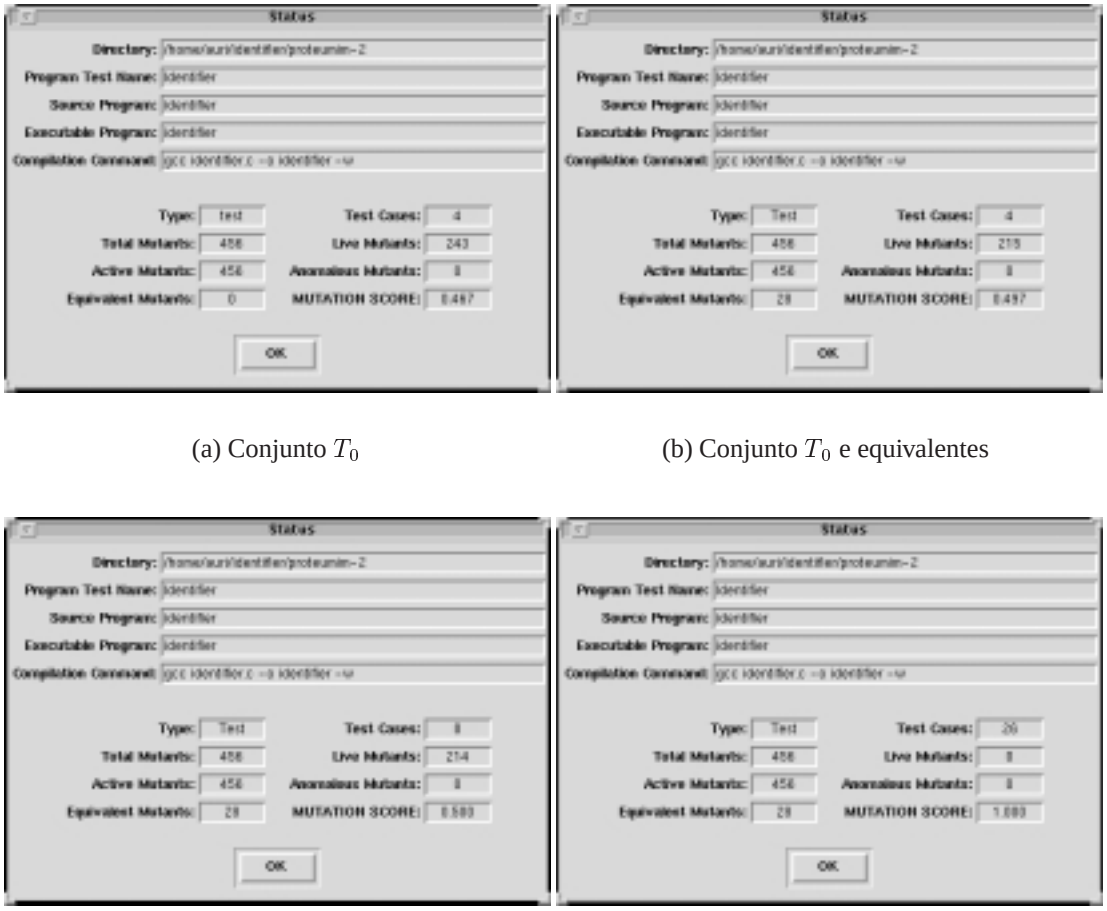


Figura 16: Conjuntos de mutantes gerados quando os operadores de interface são aplicados a uma conexão *F-G* [88].



(a) Conjunto T_0

(b) Conjunto T_0 e equivalentes

(c) Conjunto T_2

(d) Conjunto T_5

Figura 17: Telas de *status* da sessão de teste da ferramenta PROTEUM/IM.

DirVarIncDec do Grupo-I e o mutante da Figura 18 (b) foi gerado pelo operador II-ArgAriNeg do Grupo-II.

Observe que no caso do mutante da Figura 18 (a), existe uma função *PREPARE_MUTA*() no ponto de chamada da função que se deseja testar, no caso a função `valid_s`, e no corpo de `valid_s`, outra função (*IF_MUTA*()) verifica se a mesma foi chamada a partir do ponto desejado, do contrário a mutação não é ativada.

```
                    ⋮
main() {
    ...
    printf ("Identificador: ");
    achar = fgetc (stdin);
    (valid_id = PREPARE2_MUTA(valid_s(achar)));
    ...
}

int valid_s(char ch) {
    if(IF_MUTA(((ch >= 'A') && (ch <= 'Z')) ||
               ((ch >= 'a') && (ch <= 'z')),
               ((-ch >= 'A') && (ch <= 'Z')) ||
               ((ch >= 'a') && (ch <= 'z')))
    {
        return (1);
    }
    else
    {
        return (0);
    }
}
                    ⋮
```

(a) – Mutante do Grupo-I

```
                    ⋮
    printf ("Identificador: ");
    achar = fgetc (stdin);
    valid_id = valid_s(-achar);
    if(valid_id)
    {
        length = 1;
    }
                    ⋮
```

(b) – Mutante do Grupo-II

Figura 18: Exemplos de mutantes gerados pela ferramenta *PROTEUM/IM*.

Após a execução dos mutantes de interface com o conjunto T_0, 243 mutantes permaneceram vivos, resultando em um escore de mutação de, aproximadamente, 0,47 (Figura 17(a)). Analisando-se os mutantes vivos, observa-se que 28 deles eram equivalentes. Recalculando o escore de mutação, eliminando os equivalentes, o valor obtido para T_0 passou a ser de 0,497 Figura 17(b). Tal resultado demonstra que se somente o conjunto de casos de teste funcional fosse utilizado, mais de 50% dos requisitos de teste exigidos pelo critério Mutação de Interface não teriam sido cobertos por T_0.

Em seguida, na Figura 17(c) tem-se a adequação do conjunto Todos-Potenciais-Usos-adequado (T_2) em relação à Mutação de Interface. Observa-se que, mesmo tendo o dobro de casos de teste que T_0, o escore de mutação obtido com T_2 ainda não é satisfatório (0,50) e a metade dos requisitos exigidos pela Mutação de Interface ainda não foram satisfeitos.

Continuando a avaliar a cobertura obtida pelo conjunto de casos de teste adequados ao critério Análise de Mutantes (T_5) observa-se que foi possível obter um conjunto Mutação de Interface-adequado, ou seja, para o programa *identifier*, um conjunto de casos de teste Análise de Mutantes-adequado também se mostrou Mutação de Interface-adequado (Figura 17(d)). Deve-se observar que os critérios Análise de Mutantes e Mutação de Interface são incomparáveis do ponto de vista da relação de inclusão [51]. Nos experimentos conduzidos, utilizando-se

programas de maior complexidade, observou-se que conjuntos de casos de teste Análise de Mutantes-adequados não eram Mutação de Interface-adequados e vice-versa. Em nível de unidade, observou-se que os critérios Análise de Mutantes e Todos-Potenciais-Usos são incomparáveis tanto do ponto de vista teórico quanto empírico [43]. Tais resultados motivam a investigar qual a relação existente entre o critério Mutação de Interface e os critérios Potenciais-Usos de unidade [4] e de integração [97].

4 Automatização da Atividade de Teste

A qualidade e produtividade da atividade de teste são dependentes do critério de teste utilizado e da existência de uma ferramenta de teste que o suporte. Sem a existência de uma ferramenta automatizada a aplicação de um critério torna-se uma atividade propensa a erros e limitada a programas muito simples.

A disponibilidade de ferramentas de teste permite a transferência de tecnologia para as indústrias e contribui para uma contínua evolução de tais ambientes, fatores indispensáveis para a produção de software de alta qualidade. Além disso, a existência de ferramentas automatizadas auxiliam pesquisadores e alunos de Engenharia de Software a adquirir os conceitos básicos e experiência na comparação, seleção e estabelecimento de estratégias de teste.

Outro fator importante é o suporte oferecido pelas ferramentas aos testes de regressão. Os casos de teste utilizados durante a atividade de teste podem ser facilmente obtidos para revalidação do software após uma modificação. Com isso, é possível checar se a funcionalidade do software foi alterada, reduzir o custo para gerar os testes de regressão e comparar os resultados obtidos nos testes de regressão com os resultados do teste original [43].

No que diz respeito ao teste de mutação, as primeiras ferramentas começaram a surgir no final da década de 70 e início da década de 80 [25, 114]. Tais ferramentas apresentavam algumas limitações e eram destinadas ao teste de programas escritos em Fortran. A partir do final da década de 80 e durante a década de 90 novas ferramentas foram desenvolvidas a fim de suprir as deficiências encontradas nas anteriores. Entre elas destacam-se a *Mothra* [53, 107], a *Proteum* [62, 103] e a *PROTEUM/IM* [35, 115].

A *Mothra* é uma ferramenta de apoio ao critério Análise de Mutantes para o teste de programas na linguagem FORTRAN. Foi desenvolvida na *Purdue University* e *Georgia Institute of Technology* e possui 22 operadores de mutação [53, 107]. Além disso, a ferramenta apresenta interface baseada em janelas, facilitando a visualização das informações, e permite a incorporação de outras ferramentas (gerador de casos de teste, verificador de equivalência e oráculo).

As ferramentas *Proteum* [62, 103] e *PROTEUM/IM* [35, 115] foram desenvolvidas no Instituto de Ciências Matemáticas e de Computação – ICMC/USP e apóiam a aplicação os critérios Análise de Mutantes e Mutação de Interface, respectivamente. Ambas são ferramentas multilinguagem, ou seja, permitem testar programas escritos em diferentes linguagens de programação; atualmente estão configuradas para o teste de programas escritos na linguagem C. O que diferencia ambas as ferramentas é o conjunto de operadores de mutação utilizados em cada uma e o fato de que a Proteum/IM oferece características para testar a conexão entre as unidades do software.

Quanto aos critérios baseados em análise de fluxo de dados, como um dos primeiros esforços significantes tem-se a ferramenta *Asset* (*A System to Select and Evaluate Tests*), desenvolvida na *New York University* em 1985 por Frankl e Weyuker [55] para o teste de programas Pascal. Esta utiliza os critérios de adequação baseados na análise de fluxo de dados definidos por Rapps e Weyuker [30, 31].

A *Proteste* [69] tem como objetivo implementar um ambiente completo para suporte ao teste estrutural de programas escritos em Pascal, incluindo tanto critérios baseados em fluxo de controle (Todos-Nós e Todos-Arcos) quanto critérios baseados em fluxo de dados (os definidos por Rapps e Weyuker [30, 31] e os Potenciais-Usos [4]). Além de Pascal, é possível configurar o ambiente para outras linguagens através da utilização de uma ferramenta que gera analisadores de código fonte específicos para cada linguagem. O ambiente *Proteste* é um protótipo desenvolvido na Universidade Federal do Rio Grande do Sul.

Um dos esforços mais significativos no contexto de ferramentas de teste foi o desenvolvimento da *Atac* (*Automatic Test Analysis for C*), pela Telcordia Technologies [56]. A *Atac* apóia a aplicação de critérios estruturais de fluxo de controle e de dados no teste de programas escritos nas linguagens C e C++. Basicamente, a ferramenta permite verificar a adequação de um conjunto de casos de teste, visualizar código não coberto pelos casos de teste, auxiliar na geração de casos de teste e reduzir o tamanho do conjunto de teste, através da eliminação de casos de teste redundantes.

Atualmente a *Atac* está integrada ao *xSuds* (*Telcordia Software Visualization and Analysis Toolsuite*), um ambiente de suporte às atividades de teste, análise e depuração [78]. O ambiente *xSuds* vem sendo comercializado pela IBM, sendo uma forte evidência de que o uso de critérios baseados em fluxo de dados constituirá, em um futuro próximo, o estado da prática no que diz respeito ao teste de software.

As ferramentas de teste, embora implementem técnicas e critérios diferentes, apresentam características globais bastante semelhantes. Assim, pode-se identificar conjuntos básicos de operações que caracterizam atividades pertinentes ao processo de teste de software. As operações realizadas por tais ferramentas podem ser divididas em [116]: criação da sessão de teste, tratamento de casos de teste (adição, eliminação, visualização, importação e minimização do conjunto de casos de teste), geração dos requisitos de teste, análise da adequação do conjunto de casos de teste e geração de relatórios. Na Tabela 8 estão sintetizadas algumas das principais características das ferramentas *Proteum*, *PROTEUM/IM* e *PokeTool*.

5 Estudos 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. A realização de estudos empíricos procura, através da comparação entre os critérios, obter uma estratégia que seja eficaz para revelar a presença de erros no programa, ao mesmo tempo em que apresente um baixo custo de aplicação.

Para entender a importância desses estudos, considere a seguinte situação [41]: é preciso testar um programa P que será usado em um ambiente de segurança crítica e o funcionamento

Tabela 8: Principais características das ferramentas *PokeTool*, *Proteum* e *PROTEUM/IM*.

	PokeTool	*Proteum*	*PROTEUM/IM*
Linguagem	C, COBOL, FORTRAN	C	C
Geração automática de casos de teste	Não	Não	Não
Edição de casos de teste	Sim	Sim	Sim
Registro sobre caminhos não executáveis ou mutantes equivalentes	Sim	Sim	Sim
Restrição de tamanho do programa a ser testado	Não	Não	Não
Eliminação de casos de teste redundantes	Sim	Sim	Não
Interface	Menu, Janelas e *Scripts*	Janelas e *Scripts*	Janelas e *Scripts*
Sessões de teste	Sim	Sim	Sim
Apoio a experimentos	Sim	Sim	Sim
Importação de casos de teste	Sim	Sim	Sim
Geração seletiva de mutantes	Não se aplica	Sim	Sim
Ambiente compilado/interpretado	Compilado	Compilado	Compilado
Execução distribuída	Não	Não	Não
Determinação automática de mutantes equivalentes ou caminhos não executáveis (heurísticas)	Sim	Não	Não

desse sistema depende de que P tenha sido bem testado. O testador deve testar P tanto quanto for possível e, para isso, decide usar vários critérios de teste a fim de verificar a adequação dos casos de teste desenvolvidos. Inicialmente, os casos de teste são gerados de modo a satisfazerem um determinado critério C_1. Assim, uma questão que surge é: "Tendo obtido um conjunto de casos de teste T adequado ao critério C_1 e, utilizando agora o critério C_2, consegue-se melhorar o conjunto de casos de teste T?". Através de estudos empíricos procura-se responder a essa e outras questões que surgem diante da dificuldade em decidir quando um programa está suficientemente testado.

Segundo Wong *et al.* [47], **custo**, **eficácia** e **dificuldade de satisfação** (*strength*) são fatores básicos para comparar a adequação dos critérios de teste. Custo: refere-se ao esforço necessário na utilização de um critério. Pode ser medido através do 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. 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 [41]; seu objetivo é verificar o quanto consegue-se satisfazer um critério C_1 tendo satisfeito um critério C_2 (C_1 e C_2 são incomparáveis ou C_1 inclui C_2).

Utilizando-se tais fatores comparativos, estudos empíricos e teóricos são conduzidos com o objetivo de encontrar formas econômicas e produtivas para a realização dos testes. O desenvolvimento de experimentos requer a elaboração de um *framework* para sua condução. Esse *framework* é composto, basicamente, pelas seguintes atividades [42]:

- seleção e preparação dos programas;
- seleção das ferramentas de teste;
- geração de conjuntos de casos de teste;
- execução dos programas com os casos de teste gerados;
- análise dos resultados do experimento.

A geração dos conjuntos de casos de teste é feita, em geral, aleatoriamente. A geração aleatória além de ser facilmente automatizada e de gerar grandes conjuntos de casos de teste a baixo custo, também elimina possíveis influências do testador em conduzir a geração dos casos de teste de acordo com o conhecimento dos programas utilizados. Normalmente, define-se o domínio de entrada de cada programa para a geração aleatória e, quando não se consegue satisfazer o critério, casos de teste criados manualmente são adicionados ao conjunto [43].

Além dessas atividades, conforme o tipo de experimento a ser realizado, são necessárias atividades adicionais. Ao comparar o critério Análise de Mutantes com os critérios baseados em análise de fluxo de dados, por exemplo, é necessário, durante a execução dos programas, identificar os mutantes equivalentes e os caminhos/associações não executáveis.

5.0.2 Estudos Empíricos: Critérios Baseados em Análise de Fluxo de Dados e Critérios Baseados em Mutação

Do ponto de vista teórico, os critérios baseados em análise de fluxo de dados têm complexidade exponencial [4], o que motiva a condução de estudos empíricos para determinar o custo de aplicação desses critérios do ponto de vista prático.

Estudos empíricos foram conduzidos para determinação da complexidade desses critérios em termos práticos, ou seja, uma avaliação empírica desses e de outros critérios de teste objetivando a determinação de um modelo para estimativa do número de casos de teste necessários. Essa determinação é muito importante para as atividades de planejamento do desenvolvimento, por razões óbvias. Weyuker [86] caracterizou um *benchmark* para a avaliação empírica de uma família de critérios de teste estruturais; esse mesmo *benchmark* foi aplicado para uma primeira avaliação empírica dos critérios Potenciais-Usos [117]. Com a aplicação do *benchmark*, obtiveram-se resultados bastante interessantes. Em geral, pode-se dizer que os critérios Potenciais-Usos, do ponto de vista prático, são factíveis e demandam um número de casos de teste relativamente pequeno.

Através de estudos empíricos têm-se obtido evidências de que a Análise de Mutantes também pode constituir na prática um critério atrativo para o teste de programas [45]. Tais experimentos, além de mostrarem como a Análise de Mutantes se relaciona com outros critérios de teste, buscam novas estratégias a fim de reduzir os custos associados ao critério.

Mathur e Wong [45] compararam dois critérios de mutação alternativos: a Mutação Aleatória (no caso, foi selecionado 10% de cada operador de mutação) e a Mutação Restrita. Esse experimento foi conduzido para comparar qual dessas estratégias apresentava melhor relação

custo/eficácia. Segundo os autores, ambas mostraram-se igualmente eficazes, obtendo-se significativa redução no número de mutantes a serem analisados sem sensível perda na eficácia em revelar erros.

Em outro trabalho realizado por Mathur e Wong [41] foi comparada a adequação de conjuntos de casos de teste em relação aos critérios Análise de Mutantes e Todos-Usos. O objetivo do experimento era verificar a dificuldade de satisfação entre os dois critérios, bem como seus custos, uma vez que esses critérios são incomparáveis do ponto de vista teórico. Nesse estudo, os conjuntos de casos de teste Análise de Mutantes-adequados também se mostraram Todos-Usos-adequados. No entanto, os conjuntos de casos de teste Todos-Usos-adequados não se mostraram, em muitos dos casos, adequados para o critério Análise de Mutantes. Esses resultados demonstram que é mais difícil satisfazer o critério Análise de Mutantes do que o critério Todos-Usos, podendo-se dizer, na prática, que Análise de Mutantes inclui Todos-Usos [41].

Wong *et al.* [47] utilizaram a Mutação Aleatória (10%) e a Mutação Restrita para comparar o critério Análise de Mutantes com o critério Todos-Usos; o objetivo era verificar o custo, eficácia e dificuldade de satisfação desses critérios. Os autores forneceram evidências de que os critérios Todos-Usos, Mutação Aleatória (10%) e Mutação Restrita representam, nesta ordem, o decréscimo do custo necessário para a aplicação do critério (número de casos de teste requeridos), ou seja, o critério Todos-Usos requer mais casos de teste para ser satisfeito do que a Mutação Restrita. Em relação à eficácia para detectar erros, a ordem (do mais eficaz para o menos) é Mutação Restrita, Todos-Usos e Mutação Aleatória. Observou-se, com isso, que examinar somente uma pequena porcentagem de mutantes pode ser uma abordagem útil na avaliação e construção de conjuntos de casos de teste na prática. Desse modo, quando o testador possui pouco tempo para efetuar os testes (devido ao prazo de entrega do produto) pode-se usar o critério de Análise de Mutantes para testar partes críticas do software, utilizando alternativas mais econômicas, tal como a Mutação Restrita ou o critério Todos-Usos, para o teste das demais partes do software, sem comprometer significativamente a qualidade da atividade de teste.

Offutt também realizou um experimento comparando o critério Análise de Mutantes com o critério Todos-Usos [118]. Os resultados foram semelhantes àqueles obtidos por Wong *et al.* [47], ou seja, o critério Análise de Mutantes revelou um maior número de erros do que o critério Todos-Usos e mais casos de testes foram necessários para satisfazer o critério Análise de Mutantes. Além disso, os conjuntos de casos de teste Análise de Mutantes-adequados foram adequados ao critério Todos-Usos, não sendo o inverso verdadeiro, resultado semelhante ao de Mathur [41].

Nos trabalhos de Wong *et al.* [48] e Souza [43] foram comparadas seis diferentes classes de mutação restrita quanto à eficácia em revelar erros. Analisou-se a eficácia das classes de mutação obtidas a partir dos operadores de mutação da ferramenta *Proteum*. Desse experimento pode-se observar quais classes de mutação eram mais econômicas (baixo custo de aplicação) e eficazes. Com isso, foi possível o estabelecimento de uma ordem incremental para o emprego dessas classes de mutação, com base na eficácia e custo de cada uma. Desse modo, os conjuntos de casos de testes podem ser construídos inicialmente de forma a serem adequados à classe com menor relação custo$\times$eficácia. Na seqüência, quando as restrições de custo permitirem, esse conjunto pode ser melhorado de modo a satisfazer as classes de mutação com maior relação custo$\times$eficácia.

Souza [43] realizou um estudo empírico com a finalidade de avaliar o *strength* e o custo do critério Análise de Mutantes empregando, para efeito comparativo, os critérios Potenciais-Usos [4], os quais incluem o critério Todos-Usos. Os resultados demonstraram que o custo de aplicação do critério Análise de Mutantes, estimado pelo número de casos de teste necessário para satisfazer o critério, apresentou-se maior do que o custo dos critérios Potenciais-Usos. Em relação à dificuldade de satisfação (*strength*) observou-se que, de uma maneira geral, os critérios Análise de Mutantes e Todos-Potenciais-Usos (PU) são incomparáveis mesmo do ponto de vista empírico. Já os critérios Todos-Potenciais-Usos/Du (PUDU) e Todos-Potenciais-DU-Caminhos (PDU) [4] apresentaram maior *strength* que o critério Todos-Potenciais-Usos (PU) em relação à Análise de Mutantes, o que motiva a investigar-se o aspecto complementar desses critérios quanto à eficácia.

Entre os estudos empíricos que visam a estabelecer alternativas viáveis para a aplicação do critério Análise de Mutantes pode-se destacar o trabalho de Offutt *et al.* [46]. O objetivo do estudo conduzido por Offutt *et al.* [46] era determinar um conjunto essencial de operadores de mutação para o teste de programas FORTRAN, a partir dos 22 operadores de mutação utilizados pela *Mothra*. Os resultados obtidos demonstraram que apenas cinco eram suficientes para aplicar eficientemente o teste de mutação.

Nessa mesma linha, um estudo preliminar realizado por Wong *et al.* [119], comparando a Mutação Restrita no contexto das linguagens C e Fortran, resultou na seleção de um subconjunto de operadores de mutação da ferramenta *Proteum* [62, 120], constituindo uma base para a determinação do conjunto essencial de operadores de mutação para a linguagem C. A aplicação deste subconjunto de operadores possibilitou reduzir o número e mutantes gerados, mantendo a eficácia do critério em revelar a presença de erros. A seleção dos operadores de mutação foi realizada com base na experiência dos autores e os resultados motivaram a condução do trabalho de Barbosa [77]. Nesse trabalho forma conduzidos experimentos com o objetivo de investigar alternativas pragmáticas para a aplicação do critérios Análise de Mutantes e, nesse contexto, foi proposto o procedimento *Essencial* para a determinação de um conjunto essencial de operadores de mutação para a linguagem C, com base nos operadores de mutação implementados na ferramenta *Proteum*.

Para a aplicação e validação desse procedimento dois experimentos foram conduzidos. No primeiro, utilizou-se um grupo de 27 programas os quais compõem um editor de texto simplificado; no segundo, 5 programas utilitários do UNIX foram utilizados. De um modo geral, ambos os conjuntos essenciais obtidos apresentaram um alto grau de adequação em relação ao critério Análise de Mutantes, com escores de mutação acima de 0,995, proporcionando, em média, reduções de custo superiores a 65% [77].

Com a proposição do critério Mutação de Interface [35, 121] é evidente o aspecto positivo de se utilizar o mesmo conceito de mutação nas diversas fases do teste. Também é natural a indagação sobre qual estratégia utilizar para se obter a melhor relação custo$\times$eficácia quando são aplicados os critérios Análise de Mutantes e Mutação de Interface no teste de um produto. Nesse contexto, investigou-se empiricamente qual o relacionamento entre os critérios Análise de Mutantes e Mutação de Interface e como utilizar tais critérios de forma complementar na atividade de teste, tendo como objetivo contribuir para o estabelecimento de uma estratégia de teste incremental, de baixo custo de aplicação e que garanta um alto grau de adequação em

relação a ambos os critérios [44].

Inicialmente, estabeleceu-se uma estratégia incremental para a aplicação dos operadores de mutação implementados na ferramenta *PROTEUM/IM* [35], procurando reduzir o custo de aplicação do critério Mutação de Interface. A utilização dessa estratégia possibilitou uma redução no custo de aplicação do critério Mutação de Interface em torno de 25% e, ainda assim, conjuntos de casos de teste MI-adequados foram obtidos. Além disso, um estudo preliminar para a determinação do conjunto essencial de operadores de mutação interface foi conduzido, utilizando o procedimento *Essencial* definido por Barbosa [77]. O conjunto essencial obtido possibilitou a seleção de conjuntos de casos de teste altamente adequados ao critério Mutação de Interface (escores de mutação em torno de 0,998) com redução de custo, em termos do número de mutantes gerados, superior a 73% [44].

A dificuldade de satisfação entre os critérios Análise de Mutantes e Mutação de Interface também foi avaliada, observando-se que tais critérios são incomparáveis do ponto de vista da relação de inclusão [30], devendo ser utilizados em conjunto para assegurar um teste de melhor qualidade. Explorando o aspecto complementar desses critérios, algumas estratégias de aplicação dos operadores de mutação de unidade e de integração foram estabelecidas. Tais estratégias demonstraram que, mesmo com um número reduzido de operadores, é possível determinar conjuntos de casos de teste adequados ou muito próximos da adequação para ambos os critérios, a um menor custo [44].

6 Conclusão

Neste texto foram apresentados alguns critérios de teste de software e conceitos pertinentes, com ênfase naqueles considerados mais promissores a curto e médio prazo: os critérios baseados em fluxo de dados, o critério Análise de Mutantes e o critério Mutação de Interface. Foram também apresentadas as ferramentas de teste *PokeTool*, *Proteum* e *PROTEUM/IM*, assim como identificadas várias outras iniciativas e esforços de automatização desses critérios, dada a relevância desse aspecto para a qualidade e produtividade da própria atividade de teste.

Deve-se ressaltar que os conceitos e mecanismos desenvolvidos neste texto aplicam-se no contexto do paradigma de desenvolvimento de software orientado a objeto, com as devidas adaptações. Em uma primeira etapa espera-se explorar a aplicação desses critérios no teste de programas C++ e Java. Tanto no teste intra-método quanto inter-método, a aplicação dos critérios Análise de Mutantes e Mutação de Interface, respectivamente, é praticamente direta. Posteriormente, quando outras interações forem consideradas, bem como características específicas da linguagem orientada a objetos, tais como, acoplamento dinâmico, herança, polimorfismo e encapsulamento, pode ser necessário o desenvolvimento de novos operadores de mutação que modelem os erros típicos cometidos nesse contexto.

Procurou-se ressaltar o aspecto complementar das diversas técnicas e critérios de teste e a relevância de se conduzir estudos empíricos para a formação de um corpo de conhecimento que favoreça o estabelecimento de estratégias de teste incrementais que explorem as diversas características dos critérios. Nessas estratégias seriam aplicados inicialmente critérios "mais fracos"e talvez menos eficazes para a avaliação da adequação do conjunto de casos de teste, e em

função da disponibilidade de orçamento e de tempo, incrementalmente, poderiam ser utilizados critérios mais "fortes" e eventualmente mais eficazes, porém, em geral, mais caros. Estudos empíricos são conduzidos no sentindo de avaliar os aspectos de custo, *strength* e eficácia dos critérios de teste, buscando contribuir para o estabelecimento de estratégias de teste eficazes, de baixo custo e para a transformação do estado da prática, no que tange ao uso de critérios e ferramentas de teste.

Foi salientado que a atividade de teste desempenha um papel relevante na temática Qualidade de Software, tanto do ponto de vista de processo quanto do ponto de vista do produto. Por exemplo, do ponto de vista de qualidade do processo de desenvolvimento de software, o teste sistemático é uma atividade essencial para ascensão ao Nível 3 do Modelo CMM do SEI. Ainda, o conjunto de informação oriundo da atividade de teste é significativo para as atividades de depuração, estimativa de confiabilidade e de manutenção de software.

Referências

[1] M. E. Delamaro and J. C. Maldonado. Uma visão sobre a aplicação da análise de mutantes. Technical Report 133, ICMC/USP, São Carlos - SP, March 1993.

[2] 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.

[3] R. S. Pressman. *Software Engineering - A Practitioner's Approach*. McGraw-Hill, 4 edition, 1997.

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

[5] M. C. Paulk. Capability maturity model for software – version 1.1. Technical Report 93-TR-24, CMU/SEI, February 1993.

[6] T. J. Ostrand and E. J. Weyuker. Using data flow analysis for regression testing. In *Sixth Annual Pacific Northwest Software Quality Conference*, Portland - Oregon, September 1988.

[7] J. Hartmann and D. J. Robson. Techniques for selective revalidation. *IEEE Software*, pages 31–36, January 1990.

[8] A. Veevers and A. Marshall. A relationship between software coverage metrics and reliability. *Software Testing, Verification and Reliability*, 4:3–8, 1994.

[9] G. S. Varadan. Trends in reliability and test strategies. *IEEE Software*, May 1995.

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

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

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

[13] 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.

[14] E. V. Berard. *Essays on Object-Oriented Software Engineering*, volume 1. Prentice-Hall Inc, Englewood Cliffs, New Jersey, 1992.

[15] 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.

[16] D. Hoffman and P. Strooper. A case study in class testing. In *CASCON 93*, pages 472–482, IBM Toronto Laboratory, October 1993.

[17] M. D. Smith and D. J. Robson. A framework for testing object-oriented programs. *Journal of Object-Oriented Programming*, 5(3):45–53, June 1992.

[18] P. C. Jorgensen and C. Erickson. Object oriented integration testing. *Communications of the ACM*, 37(9):30–38, September 1994.

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

[20] G. C. Murphy, P. Townsend, and P. S. Wong. Experiences with cluster and class testing. *Communications of the ACM*, 37(9):39–47, September 1994.

[21] S. C. P. F. Fabbri, J. C. Maldonado, M. E. Delamaro, and P. C. Masiero. Uma ferramenta para apoiar a validação de máquinas de estado finito pelo critério de análise de mutantes. In *Software Tools Proceedings of the 9th Brazilian Symposium on Software Engineering*, pages 475–478, Recife - PE, Brazil, October 1995.

[22] 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.

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

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

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

[26] J. B. Goodenough and S. L. Gerhart. Towards a theory of test data selection. *IEEE Transactions on Software Engineering*, 2(3):156–173, September 1975.

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

[28] W. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, SE-8(4):371–379, July 1982.

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

[30] 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.

[31] 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.

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

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

[34] 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.

[35] M. E. Delamaro. *Mutação de Interface: Um Critério de Adequação Inter-procedimental para o Teste de Integração*. PhD thesis, Instituto de Física de São Carlos - Universidade de São Paulo, São Carlos, SP, June 1997.

[36] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, 10(4), July 1984.

[37] 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.

[38] 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.

[39] M. R. Girgis and M. R. Woodward. An experimental comparison of the error exposing ability of program testing criteria. In *Workshop on Software Testing*, pages 64–71, Banff - Canadá, July 1986. Computer Science Press.

[40] 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.

[41] A. P. Mathur and W. E. Wong. An empirical comparison of data flow and mutation based test adequacy criteria. *The Journal of Software Testing, Verification, and Relability*, 4(1):9–31, March 1994.

[42] W. E. Wong. *On Mutation and Data Flow*. PhD thesis, Department of Computer Science, Purdue University, W. Lafayette, IN, December 1993.

[43] 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.

[44] A. M. R. Vincenzi. Subídios para o estabelecimento de estratégias de teste baseadas na técnica de mutação. Master's thesis, ICMC-USP, São Carlos - SP, November 1998.

[45] A. P. Mathur and W. E. Wong. Evaluation of the cost of alternate mutation strategies. In *7th Brazilian Symposium on Software Engineering*, pages 320–335, Rio de Janeiro, RJ, Brazil, October 1993.

[46] 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, 1996.

[47] W. E. Wong, A. P. Mathur, and J. C. Maldonado. Mutation versus all-uses: An empirical evaluation of cost, strength, and effectiveness. In *International Conference on Software Quality and Productivity*, pages 258–265, Hong Kong, December 1994.

[48] W. E. Wong, J. C. Maldonado, M. E. Delamaro, and A. P. Mathur. Constrained mutation in C programs. In *8th Brazilian Symposium on Software Engineering*, pages 439–452, Curitiba, PR, Brazil, October 1994.

[49] E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi. Towards the determination of sufficient mutant operators for C. In *First International Workshop on Automated Program Analysis, Testing and Verification*, Limerick, Ireland, June 2000. (Accepted for publication in a special issue of the Software Testing Verification and Reliability Journal).

[50] J. C. Maldonado, E. F. Barbosa, A. M. R. Vincenzi, and M. E. Delamaro. Selective mutation for C programs: Unit and integration testing. In *Mutation 2000 Symposium*, San Jose, CA, October 2000. To appear.

[51] A. M. R. Vincenzi, J. C. Maldonado, E. F. Barbosa, and M. E. Delamaro. Unit and integration testing strategies for C programs using mutation-based. In *Symposium on Mutation Testing*, San Jose, CA, October 2000. To appear.

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

[53] R. A. DeMillo, D. S. Gwind, K. N. King, W. N. McKraken, and A. J. Offutt. An extended overview of the mothra testing environment. In *Software Testing, Verification and Analysis*, Banff, Canadá, July 1988.

[54] M. Luts. Testing tools. *IEEE Software*, 7(3), May 1990.

[55] F. G. Frankl and E. J. Weyuker. Data flow testing tools. In *Softfair II*, pages 46–53, San Francisco, CA, December 1985.

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

[57] J. R. Horgan and S. A. London. Data flow coverage and the C language. In *Symposium Software Testing, Analysis, and Verification*, pages 87–97, October 1991.

[58] B. Korel and J. W. Laski. A tool for data flow oriented program testing. In *Softfair II*, pages 34–38, San Francisco, CA, December 1985.

[59] T. J. Ostrand and E. J. Weyuker. Data flow based test adequacy for languages with pointers. In *Symposium on Software Testing*, pages 74–86, October 1996.

[60] 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.

[61] M. L. Chaim. Poke-tool – 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.

[62] M. E. Delamaro. Proteum: Um ambiente de teste baseado na análise de mutantes. Master's thesis, ICMC/USP, São Carlos - SP, October 1993.

[63] S. C. P. F. Fabbri, J. C. Maldonado, M. E. Delamaro, and P. C. Masiero. Proteum/FSM Ű uma ferramenta para apoiar a validação de máquinas de estado finito pelo critério análise de mutantes. In *IX Simpósio Brasileiro de Engenharia de Software*, pages 475–478, Recife, PE, October 1995.

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

[65] M. E. Delamaro, J. C. Maldonado, A. Pasquini, and A. P. Mathur. Interface mutation test adequacy criterion: An empirical evaluation. To be submited.

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

[67] E. J. Weyuker and T. Ostrand. Theory of program testing and the application of revealing subdomains. *IEEE Transactions on Software Engineering*, 6, June 1980.

[68] 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.

[69] A. M. Price and A. Zorzo. Visualizando o fluxo de controle de programas. In *IV Simpósio Brasileiro de Engenharia de Software*, Águas de São Pedro, SP, October 1990.

[70] R. A. DeMillo and A. J. Offutt. Constraint based automatic test data generation. *IEEE Transactions on Software Engineering*, SE-17(9):900–910, September 1991.

[71] E. J. Weyuker, S. N. Weiss, and R. G. Hamlet. Comparison of program testing strategies. In *4th Symposium on Software Testing, Analysis and Verification*, pages 154–164, Victoria, British Columbia, Canadá, 1991. ACM Press.

[72] R. A. DeMillo, A. P. Mathur, and W. E. Wong. Some critical remarks on a hierarchy of fault-detecting abilities of test methods. *IEEE Transactions on Software Engineering*, SE-21(10):858–860, October 1995.

[73] 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.

[74] Y. K. Malaiya, N. Li, J. Bieman, R. Karcick, and B. Skibe. The relationship between test coverage and reliability. In *International Symposium on Software Reliability Engineering*, pages 186–195, Monterey, CA, November 1994.

[75] W. E. Wong and A. P. Mathur. Reducing the cost of mutation testing: An empirical study. *The Journal of Systems and Software*, 31(3):185–196, December 1995.

[76] W. E. Wong and A. P. Mathur. Fault detection effectiveness of mutation and data flow testing. *Software Quality Journal*, 4(1):69–83, March 1995.

[77] 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 *12th Brazilian Symposium on Software Engineering*, pages 103–120, Florian'opolis, SC, October 1998.

[78] H. Agrawal, J. Alberi, J. R. Horgan, J. Li, S. London, W. E. Wong, S. Ghosh, and N. Wilde. Mining system tests to aid software maintenance. *ieeec*, pages 64–73, July 1998.

[79] IEEE. Ieee standard glossary of software engineering terminology. Standard 610.12, IEEE Press, 1990.

[80] F. G. Frankl. *The Use of Data Flow Information for the Selection and Evaluation of Software Test Data*. PhD thesis, Universidade de New York, New York, NY, October 1987.

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

[82] M. J. Harrold. Testing: A roadmap. In *22th International Conference on Software Engineering*, June 2000. Talk about The Future of Software Engineering.

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

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

[85] 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.

[86] E. J. Weyuker. The cost of data flow testing: an empirical study. *IEEE Transactions on Software Engineering*, SE-16(2):121–128, February 1990.

[87] M. Jino M. L. Chaim, J. C. Maldonado. Ferramenta para o teste estrutural de software baseado em análise de fluxo de dados: o caso poke-tool. In *Workshop do Projeto de Validação e Teste de Sistemas de Operação*, pages 29–39, Águas de Lindóia - SP, January 1997.

[88] M. E. Delamaro, J. C. Maldonado, and A. M. R. Vincenzi. Proteum/IM 2.0: An integrated mutation testing environment. In *Mutation 2000 Symposium*, San Jose, CA, October 2000. To appear.

[89] W. E. Howden. *Functional Program Testing and Analysis*. McGrall-Hill, New York, 1987.

[90] 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.

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

[92] 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*, pages 307–319, Rio de Janeiro, RJ, October 1993.

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

[94] A. Haley and S. Zweben. Development and application of a white box approach to integration testing. *The Journal of Systems and Software*, 4:309–315, 1984.

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

[96] Z. Jin and A. J. Offut. Integration testing based on software couplings. In *X Annual Conference on Computer Assurance (COMPASS 95)*, pages 13–23, Gaithersburg, Maryland, January 1995.

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

[98] P. S. J. Leit ao. Suporte ao teste estrutural de programas cobol no ambiente poke-tool. Master's thesis, DCA/FEE/UNICAMP, Campinas, SP, August 1992.

[99] R. P. Fonseca. Suporte ao teste estrutural de programas fortran no ambiente poke-tool. Master's thesis, DCA/FEE/UNICAMP, Campinas, SP, January 1993.

[100] 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.

[101] S. R. Vergilio. *Critérios Restritos: Uma Contribuição para Aprimorar a Eficácia da Atividade de Teste de Software*. PhD thesis, DCA/FEEC/UNICAMP, Campinas, SP, July 1997.

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

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

[104] H. Agrawal, R. A. DeMillo, R. Hataway, Wm. Hsu, W. Hsu, E. Krauser, R. J. Martin, A. P. Mathur, and E. H. Spafford. Design of mutant operators for C programming language. Technical Report SERC-TR41-P, Software Engineering Research Center, Purdue University, March 1989.

[105] 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.

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

[107] B. J. Choi, A. P. Mathur, and A. P. Pattison. pmothra: Scheduling Mutants for Execution on a Hypercube. In *3rd Symposium on Software Testing, Analysis and Verification*, pages 58–65, Key West, FL, December 1989.

[108] E. W. Krauser, A. P. Mathur, and V. J. Rego. High performance software testing on simd machines. *IEEE Transactions on Software Engineering*, SE-17(5):403–422, May 1991.

[109] A. P. Mathur and E. W. Krauser. Modeling mutation on vector processor. In *X International Conference on Software Engineering*, pages 154–161, Singapore, April 1988.

[110] B. J. Choi and A. P. Mathur. High-performance mutation testing. *The Journal of Systems and Software*, 1(20):135–152, February 1993.

[111] A. C. Marshall, D. Hedley, I. J. Riddell, and M. A. Hennell. Static dataflow-aided weak mutation analysis (sdawm). *Information and Software Technology*, 32(1), January/February 1990.

[112] S. P. F. Fabbri and J. C. Maldonado. Proteum/FSM: Uma ferramenta de teste baseada na análise de mutantes para apoiar a validação de especificações em máquinas de estado finito. *Revista da Asser*, November 1996.

[113] A. M. R. Vincenzi, E. F. Barbosa, Vincenzi S. R. S. Souza, M. E. Delamaro, and J. C. Maldonado. Critério análise de mutantes: Estado atual e perspectivas. In *Workshop do Projeto de Validação e Teste de Sistemas de Operação*, pages 15–26, Águas de Lindóia - SP, January 1997.

[114] A. T. Acree. *On Mutation*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, 1980.

[115] M.E. Delamaro and J.C. Maldonado. Interface mutation: Assessing testing quality at interprocedural level. In *19th International Conference of the Chilean Computer Science Society (SCCC'99)*, pages 78–86, Anaheim - CA, November 1999.

[116] E. Y. Nakagawa and et al. Aspectos de projeto e implementação de interfaces gráficas do usuário para ferramentas de teste. In *Workshop do Projeto de Validação e Teste de Sistemas de Operação*, pages 57–67, Águas de Lindóia - SP, January 1997.

[117] S. R. Vergílio, J. C. Maldonado, and M. Jino. Caminhos não-executáveis na automação das atividades de teste. In *VI Simpósio Brasileiro de Engenharia de Software*, pages 343–356, Gramado, RS, November 1992.

[118] A. J. Offutt, J. Pan, K. Tewary, and T. Zhang. An experimental evaluation of data flow and mutation testing. *Software Practice and Experience*, 26(2):165–176, 1996.

[119] W.E. Wong, J.C. Maldonado, M.E. Delamaro, and S.R.S. Souza. A comparison of selective mutation in C and fortran. In *Workshop of the Validation and Testing of Operational Systems Project*, pages 71–80, Águas de Lindóia - SP - Brazil, January 1997.

[120] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur. Proteum - a tool for the assesment of test adequacy for C programs - user's guide. Technical Report SERC-TR168-P, Software Engineering Research Center, Purdue University, April 1996.

[121] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur. Interface mutation: An approach for integration testing. IEEE Transactions on Software Engineering, (to appear).