**UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE FÍSICA DE SÃO CARLOS
DEPARTAMENTO DE FÍSICA E INFORMÁTICA**

“Mutação de Interface: Um Critério de Adequação Interprocedimental para o
Teste de Integração”. *on OK*

MARCIO EUDARDO DELAMARO

Tese apresentada ao Instituto de Física de São Carlos, Universidade de São Paulo, para obtenção do título de Doutor em Ciências “Física Aplicada-opção Física Computacional”.

Orientador: Prof. Dr. José Carlos Maldonado

SÃO CARLOS SÃO PAULO
1997

IFSC-USP SERVIÇO DE BIBLIOTECA E
INFORMÁTICA

DELAMARO, MARCIO EDUARDO

Mutação de Interface: Um Critério de Adequação Interprocedimental para o Teste de Integração

168 p.

Tese (Doutorado) - Instituto de Física de São Carlos. 1997

Orientador: Prof. Dr. José Carlos Maldonado

1. Mutação. 2. Critério de Adequação. 3. PROTEUM/IM

I. Título.

UNIVERSIDADE
DE SÃO PAULO

Instituto de Física de São Carlos

Av. Dr. Carlos botelho, 1465
CEP 13560-250 - São Carlos - SP
Brasil

Fone (016) 274-3444
Fax (016) 272-2218

MEMBROS DA COMISSÃO JULGADORA DA TESE DE DOUTORADO DE **MÁRCIO EDUARDO DELAMARO** APRESENTADA AO INSTITUTO DE FÍSICA DE SÃO CARLOS, UNIVERSIDADE DE SÃO PAULO, EM 17/06/1997.

COMISSÃO JULGADORA:

Prof. Dr. José Carlos Maldonado (ICMSC-USP)

Prof. Dr. Carlos Antônio Ruggiero (IFSC-USP)

Prof. Dr. Álvaro Garcia Neto (IFSC-USP)

Profa. Dra. Eliane Martins (UNICAMP)

Prof. Dr. Wanderley Lopes de Souza (UFSCar)

C:\Meus Documentos\WLA\OFDEFD.DOC

À Loise, à Márcia e ao
Dela

Agradecimentos

- Agradeço inicialmente e especialmente ao meu orientador e amigo Prof. Dr. José Carlos Maldonado por todo apoio, orientação e amizade.
- To the Software Engineering Research Center (SERC) and Computer Science Department at Purdue University. Specialy to Dr. Aditya P. Mathur who made possible my stay at Purdue as a Visiting Scholar and who enormously contributed to this work.
- To Dr. Alberto Pasquini who allowed and helped me to use the *SPACE* program in the experimental studies.
- À Márcia pelo apoio, compreensão, amor e carinho em (quase) todos os momentos.
- À minha família, sogros e cunhados.
- À Profa. Dra. Sandra Fabbri pela inestimável ajuda na revisão da tese.
- Aos companheiros do Grupo de Engenharia de Software, do ICMSC e do IFSC. Em especial à Sandra, Simone e Paulo, Elisa, Tchelo, Rô Auri e Mary Ellen.
- Ao pessoal da Santa Casa: Betão, Koxinha, Trolha, Eltão, Marião, Luciano e Elvis.
- À Wladerez e à Ivani, pela pronta ajuda, sempre que solicitada.
- Ao Marcelo pela ajuda na condução dos experimentos relatados nesta tese.
- A todos os brasileiros em Purdue, por tornarem mais quente nossa estadia naquela terra gelada.
- À CAPES pelo suporte financeiro.

Índice

Lista de Figuras	vii
Lista de Tabelas	ix
1 Introdução	1
1.1 Motivação	4
1.2 Objetivos	5
1.3 Organização da Tese	6
2 Revisão Bibliográfica	9
2.1 Definições Básicas	9
2.2 Critérios de Adequação de Casos de Teste	11
2.2.1 Critérios Funcionais	12
2.2.2 Critérios Estruturais	13
2.2.3 Critérios Baseados em Defeitos	16
2.3 Avaliação e Comparação entre Critérios	22
2.3.1 Estudos Teóricos	22
2.3.2 Estudos Empíricos	27
2.4 Ferramentas de Teste	30
2.5 Fases da Atividade de Teste	33
2.5.1 Teste de Unidade	34
2.5.2 Teste de Integração	37
2.6 Considerações Finais	47
3 Mutação de Interfaces	49
3.1 Modelo de Falhas Causadas por Erros de Integração	50
3.2 O Critério Mutação de Interface	53
3.3 Operadores de Mutação de Interface para C	57
3.3.1 Grupo I - Mutações Efetuadas na Função Chamada	62
3.3.2 Grupo II - Mutações Efetuadas nos Pontos de Chamada	69
3.4 Análise Teórica do Critério Mutação de Interface	71
3.4.1 Análise de Complexidade	72
3.4.2 Análise de Inclusão	74
3.5 Critério de Mutação de Interface Alternativos	79
3.6 Considerações Finais	82

4	<i>PROTEUM/IM</i>: Definição e Implementação	85
4.1	Funcionalidade	86
4.1.1	Interface Gráfica	87
4.1.2	Scripts de Teste	93
4.2	Aspectos de Implementação	99
4.2.1	Aplicação dos Operadores de Mutação	101
4.2.2	Execução dos Mutantes	104
4.3	Considerações Finais	108
5	Avaliação Empírica do Custo e Eficácia do Critério Mutação de Interface	111
5.1	Estudo de Caso 1: O Programa <i>Sort</i>	112
5.1.1	Descrição do Experimento	114
5.1.2	Análise dos Dados	121
5.2	Estudo de Caso 2: O Programa <i>SPACE</i>	125
5.2.1	Descrição do Experimento	129
5.2.2	Análise dos Dados	142
5.3	Considerações Finais	145
6	Conclusão	147
6.1	Contribuições deste Trabalho	150
6.2	Trabalhos Futuros	151
	Bibliografia	153
	Apêndice A: Defeitos do Programa <i>SPACE</i>	159

Lista de Figuras

1	Exemplo de grafo de fluxo de controle	14
2	Exemplo de mutações para programas FORTRAN	20
3	Hierarquia dos critérios de Rapps e Weyuker de acordo com a relação de inclusão	24
4	Hierarquia dos critérios de Rapps e Weyuker e Potenciais Usos de acordo com a relação de inclusão	24
5	Exemplo de como o critério todos-p-usos inclui o critério todas- arestas	25
6	Hierarquia dos critérios de Rapps e Weyuker e Potenciais Usos quando não se tem a restrição de que cada comando de decisão possui pelo menos uma referência a variável	26
7	Corpo do analisador léxico a ser testado	35
8	Driver do analisador léxico	36
9	Stubs do analisador léxico	37
10	Programa <i>GetMax</i>	42
11	Grafo de fluxo dos procedimentos <i>GetMax</i> e <i>PairMax</i>	43
12	Passo inicial na construção do grafo de fluxo interprocedimental para os procedimentos <i>GetMax</i> e <i>PairMax</i>	44
13	Grafo de fluxo interprocedimental para os procedimentos <i>GetMax</i> e <i>PairMax</i> com indicação dos acoplamentos de parâmetros formais e reais	45
14	Grafo de fluxo interprocedimental para os procedimentos <i>GetMax</i> e <i>PairMax</i> com os conjuntos de usos alcançáveis	46
15	Modelo de falhas causadas por erros de integração: (a) Tipo 1; (b) Tipo 2; e (c) Tipo 3	51
16	Exemplos de erros dos tipos 1, 2 e 3	52
17	Estrutura de programa, utilizada para exemplificar algumas carac- terísticas da Mutação de Interface	55
18	Programa utilizado para exemplificar a aplicação dos operadores de mutação no ponto de chamada e dentro da unidade chamada	56
19	Indicação dos conjuntos de mutantes requeridos para o teste da conexão $f-g$	60
20	Mutantes associados a diferentes chamadas da conexão $f-g$	60
21	Programa exemplo para aplicação dos operadores de mutação	64
22	Programa para exemplificar associações requeridas pelo critério de Harrold e Soffa	75
23	Programa que mostra que H&S não inclui MI	77

24	Possível sequência de aplicação de critérios MI-aleatórios	81
25	Operações disponíveis na interface da <i>PROTEUM/IM</i>	88
26	Criação de uma base de teste	88
27	Inclusão de um caso de teste; a) parâmetros de linha de comando; b) dados de tempo de execução	89
28	Visualização de casos de teste	91
29	Geração de mutantes	91
30	Visualização de mutantes	93
31	Relatório resumo sobre o estado atual da sessão	94
32	Relatório de casos de teste; a) configuração do relatório; b) exemplo de relatório	94
33	Programa para visualização e análise dos mutantes	99
34	Exemplo de um script de teste que utiliza os programas da ferra- menta <i>PROTEUM/IM</i>	100
35	Processo de aplicação dos operadores de mutação	102
36	Tradução de programa C para Linguagem Intermediária	103
37	Processo de execução dos mutantes	105
38	Exemplo de mutante-fonte com o código correspondente a dois mu- tantes distintos	106
39	Grafo de Chamadas do programa <i>Sort</i>	112
40	Processo de geração de um conjunto de teste MI-adequado para o programa <i>Sort</i>	116
41	Arquivo de entrada para programa <i>SPACE</i>	126
42	Arquivo de saída do programa <i>SPACE</i>	127
43	Processo de “teste” do programa <i>SPACE</i> utilizando Mutação de Interface	130
44	Processo de seleção de casos de teste para cada experimento do programa <i>SPACE</i>	131

Lista de Tabelas

1	Relações definidas por Frankl e Weyuker e suas implicações quanto à efetividade em revelar erros	27
2	Programas que compõem a ferramenta Proteum	32
3	Associações definição-uso interprocedimentais entre PairMax e Get-Max	47
4	Conjunto de constantes requeridas	62
5	Operadores de Mutação de Interface	63
6	Mutações relativas ao programa da Figura 21, para operadores de troca de variáveis de interface	64
7	Mutações relativas ao programa da Figura 21, para operadores de troca de variáveis não de interface	66
8	Mutações relativas ao programa da Figura 21, para operadores de incremento e decremento de variáveis e constantes	66
9	Mutações relativas ao programa da Figura 21, para operadores de inclusão de operadores unários	67
10	Mutações relativas ao programa da Figura 21, para operadores de cobertura	69
11	Mutações relativas ao programa da Figura 21, para operadores do Grupo II	70
12	Programas que compõem a ferramenta <i>PROTEUM/IM</i>	95
13	Métricas relacionadas com as funções do programa <i>Sort</i>	113
14	Defeitos implantados no programa <i>Sort</i>	114
15	Conexões onde foram aplicados os operadores de Mutação de Interface no programa <i>Sort</i>	115
16	Número de mutantes gerados, por operador, para o programa <i>Sort</i>	115
17	Resultados de efetividade	118
18	Efetividade dos critérios MI-aleatório e MI-restritos	119
19	Efetividade dos critérios MI-N-seletivos	119
20	Efetividade dos conjuntos de teste restritos a cada conexão	120
21	Cardinalidade dos conjuntos MI-adequados	121
22	Efetividade dos conjuntos randomicamente gerados	122
23	Métricas relacionadas com o programa <i>SPACE</i>	125
24	Classificação de defeitos do programa <i>SPACE</i>	128
25	Número de mutantes gerados, por operador, para a versão <i>SPACE₀</i>	133
26	Defeitos implantados no experimento I	133
27	Resultados da aplicação da Mutação de Interface no experimento I	134

28 Sumário dos resultados de efetividade e custo 135

29 Defeitos implantados no experimento II 136

30 Resultados da aplicação da Mutação de Interface no experimento II 136

31 Defeitos implantados no experimento III 136

32 Resultados da aplicação da Mutação de Interface no experimento III137

33 Defeitos implantados no experimento IV 137

34 Resultados da aplicação da Mutação de Interface no experimento IV137

35 Defeitos implantados no experimento V 138

36 Resultados da aplicação da Mutação de Interface no experimento V 138

37 Defeitos implantados no experimento VI 138

38 Resultados da aplicação da Mutação de Interface no experimento VI139

39 Defeitos implantados no experimento VII 139

40 Resultados da aplicação da Mutação de Interface no experimento
VII 139

41 Defeitos implantados no experimento VIII 140

42 Resultados da aplicação da Mutação de Interface no experimento
VIII 140

43 Defeitos implantados no experimento IX 140

44 Resultados da aplicação da Mutação de Interface no experimento IX141

45 Defeitos implantados no experimento X 141

46 Resultados da aplicação da Mutação de Interface no experimento X 141

47 Casos de teste selecionados e casos de teste que revelam defeitos 7
e 17 no experimento VII 143

48 Número de defeitos revelados X custo dos critérios 145

RESUMO

Um dos pontos fundamentais na atividade de teste de software é o projeto de casos de teste. Diversos critérios de adequação têm sido propostos com o objetivo de fornecer meios que permitam que a avaliação e elaboração de casos de teste sejam feitas de maneira sistemática e fundamentada teoricamente. Infelizmente, a maioria dos critérios de adequação de casos de teste definidos tem seu uso restrito ao teste de unidade. Para fases posteriores da atividade de teste, em particular para o teste de integração, nota-se a ausência de critérios de adequação, principalmente porque os critérios propostos definem requisitos de teste que se restringem aos limites de uma única unidade, não exercitando de maneira efetiva as interações entre as unidades, que devem ser o alvo principal no teste de integração. Com exceção de alguns poucos trabalhos que procuram estender critérios estruturais para o nível interprocedimental, tem-se utilizado nessa fase de teste, quase que exclusivamente, critérios funcionais.

Dada essa ausência de critérios e salientando ainda o caráter complementar entre as diferentes técnicas de teste, esta tese apresenta um critério de teste interprocedimental baseado em defeitos chamado de Mutação de Interface. Esse critério busca exercitar as interações entre as unidades através da seleção de casos de teste que distingam mutantes criados pela introdução de defeitos típicos e que, de acordo com um modelo definido, caracterizam erros de integração. Definiu-se um conjunto de operadores de Mutação de Interface que concentram sua aplicação em pontos do programa relacionados com as interações entre as unidades, como, por exemplo, chamadas de sub-programas e seus parâmetros. Dado o alto custo de aplicação, inerente de critérios baseados em mutação, e pelas próprias características do conjunto de operadores de Mutação de Interface, torna-se necessário definirem-se abordagens para reduzir esse custo. Assim, foram estabelecidas maneiras de se parametrizar a aplicação dos operadores de mutação, definindo-se critérios de Mutação de Interface alternativos, estendendo-se abordagens utilizadas no teste de mutação convencional como mutação aleatória e mutação restrita.

A aplicação de um critério de teste está fortemente condicionada à sua automatização. A definição de um critério de teste sem que pelo menos se apontem soluções para sua automatização tem pouca utilidade prática. Por isso, especificou-se e implementou-se a ferramenta *PROTEUM/IM* para apoiar a aplicação do critério Mutação de Interface. Essa ferramenta torna-se essencial neste trabalho à medida que permite que estudos empíricos possam ser realizados, avaliando o critério proposto.

Dois estudos de casos são apresentados. Esses estudos aplicam o critério Mutação de Interface em programas reais e buscam avaliar seu custo e sua eficácia em revelar erros. Estes estudos aplicam ainda critérios alternativos, mostrando que a Mutação de Interface é bastante efetiva em revelar erros e pode ter um custo de aplicação bastante reduzido, quando aplicada de maneira incremental, utilizando-se as parametrizações que os operadores de mutação oferecem.

IBCC-UBP SERVIÇO DE BIBLIOTECA

ABSTRACT

The project of test cases is one of the most important topics in the software testing activity. Several criteria have been proposed aiming at allowing the evaluation and selection of test cases in a systematic and theoretically well founded way. Unfortunately, the use of most of these criteria is restricted to the unit testing phase. For other testing phases, in particular for integration testing, there is a lack of such criteria, mainly because the existing criteria define test requirements only in the scope of a single unit. They are not able to effectively exercise the interactions between units, what should be the focus of integration testing. Excepting some few works that extend structural criteria to the interprocedural level, only functional testing has been used at integration testing phase.

Given this lack of criteria and the complementary characteristics of different testing techniques, this thesis presents an interprocedural fault based criterion named Interface Mutation. This criterion exercises the interactions between units through the selection of test cases that distinguish mutants created by introducing typical faults that characterize integration errors. A set of Interface Mutation operators was defined. The focus of these operators are the points of the program related to the unit interactions, for instance, subprogram calls and their parameters. Given the high cost associated to mutation testing in general and particularly to the Interface Mutation operators, it is necessary to define some approaches to reduce its application cost. Thus, some parameterizations were defined to the mutation operators, allowing to establish alternative Interface Mutation criteria, extending approaches already used in conventional mutation testing, as random mutation and constrained mutation.

The application of any testing criterion strongly depends on its automatization. The definition of a criterion, without pointing out ways to its implementation has little practical utility. So, a tool named *PROTEUM/IM* was specified and implemented to support the application of Interface Mutation. This tool is an essential point in the present work because it allows the conduction of empirical studies aiming at evaluating the proposed criterion.

Two case studies are presented. In these studies the criterion Interface Mutation is applied to real programs and the cost of its application as well as its error revealing effectiveness are evaluated. Alternative criteria are also used, showing that Interface Mutation is very effective to reveal errors and can be applied with a reduced cost if used in an incremental way, taking advantage of the parameterization characteristics provided by the Interface Mutation operators set.

Capítulo 1

Introdução

Nos dias de hoje, sistemas de computação encontram-se presentes em praticamente todas as áreas da atividade humana. A crescente complexidade e diversidade de tais sistemas, somada à necessidade de qualidade e produtividade no seu desenvolvimento, fazem com que métodos, ferramentas e procedimentos especializados sejam requeridos. A esse conjunto de métodos, ferramentas e procedimentos chamamos *Engenharia de Software*.

Apesar das diversas abordagens que procuram evitar que erros sejam introduzidos no software durante todo o processo de desenvolvimento, a fase de teste é ainda um importante elemento para aprimorar sua qualidade, dando evidências da confiabilidade do produto. Por isso, a atividade de teste de software constitui uma importante área de pesquisa dentro da Engenharia de Software.

Um dos pontos-chaves na atividade de teste é o projeto de casos de teste. Para projetar casos de teste o testador deve contar com um conjunto de técnicas, critérios e métodos que permitam que a elaboração dos casos de teste seja feita de maneira sistemática e fundamentada teoricamente. Um dos objetivos principais é fornecer ao testador meios para selecionar casos de teste que tenham alta probabilidade de revelar os possíveis erros presentes no software, com um baixo custo.

Um critério de adequação de casos de teste estabelece requisitos a serem satisfeitos por um “bom” conjunto de teste para um determinado programa. Basicamente, critérios de adequação podem ser classificados em três técnicas distintas.

A primeira é a técnica funcional que utiliza apenas informação da especificação funcional do software para derivar os requisitos de teste. Em contraste, a técnica estrutural utiliza informação da estrutura da implementação a ser testada para derivar esses requisitos. E por último, a técnica baseada em defeitos utiliza informação sobre classes de erros típicos que são cometidos no processo de desenvolvimento. Conforme destaca Pressman [1], essas diferentes técnicas de teste possuem características complementares e por isso devem ser utilizadas em conjunto, buscando aumentar a qualidade da atividade de teste.

Diversos critérios de adequação têm sido propostos, avaliados e comparados, sempre buscando critérios que possam levar a conjuntos de teste com alta probabilidade em revelar possíveis defeitos no software e cujo custo de aplicação seja baixo. Como exemplo de critérios funcionais podem ser citados o Particionamento em Classes de Equivalência [2], a Análise de valor Limite [1] e o Grafo de Causa-efeito [1]. Dentre os critérios estruturais podem ser citados a Família de Critérios de Fluxo de Dados (ou critérios de Rapps e Weyuker) [3], os Critérios Potenciais Usos [4] e os Critérios de Cobertura de Condições [2]. Na técnica baseada em defeitos destacam-se a Semeadura de Erros [5], A Análise de Mutantes [6] e a Mutação Fraca [7].

Vários autores têm também se preocupado em, além de definir critérios, avaliar e comparar critérios entre si, estabelecendo características importantes e que podem auxiliar no estabelecimento de estratégias de teste que agreguem esses critérios. Esses trabalhos podem ser divididos em dois grupos. O primeiro é composto por estudos teóricos que buscam avaliar os critérios de teste analisando suas características como custo de aplicação e efetividade em revelar erros, ou comparando critérios entre si. Isso foi feito por Rapps e Weyuker [3, 8, 9], Maldonado [4], Mathur e Wong [10], DeMillo et al. [11], Zhu [12], entre outros. O outro grupo de trabalhos também avalia critérios de teste, porém baseia-se em estudos empíricos onde, em geral, os critérios são aplicados em alguns programas selecionados e o resultado dessa aplicação é avaliada, também para que sejam medidos custo e efetividade, ou para comparar-se um critério com outro. Exemplos desse tipo de trabalho são também fartamente encontrados na literatura. Podem ser citados Offut e Rothermel [13, 14], Offut e Lee [15], Maldonado [4], Maldonado et al. [16], Mathur e Wong [17, 18, 19, 20, 21], Wong et al. [22], Weyuker [23], entre outros.

Na prática, a aplicação de um critério de teste está fortemente condicionada à sua automatização. Sua aplicação de forma manual tende a ser excessivamente trabalhosa e propensa a erros. Por isso, vários pesquisadores têm se preocupado

em especificar e desenvolver ferramentas que auxiliem na aplicação dos critérios. Essas ferramentas devem, pelo menos, determinar os requisitos de teste para um determinado programa e, dado um conjunto de teste, verificar se ele satisfaz ou não aos requisitos determinados. Várias ferramentas de teste foram especificadas e podem ser encontradas na literatura. É o caso, por exemplo, da ferramenta POKE-TOOL [24, 25, 26], da ferramenta Mothra [27], da ferramenta ATAC [28] e da ferramenta Proteum [29, 30, 31]. Tais ferramentas são muito importantes, não só para que os critérios possam ser utilizados na prática, mas também porque permitem que a própria pesquisa e aprimoramento dos critérios possam ser realizados através de estudos empíricos em que os critérios são aplicados e avaliados.

Dentre os diversos critérios de teste, duas classes têm especialmente merecido a atenção de pesquisadores. São os critérios baseados em fluxo de dados [3, 32, 4] e os baseados em mutação. Particularmente importante no escopo desta tese, o critério Análise de Mutantes (ou teste de mutação como também é chamado), enquadra-se nessa segunda classe. Esse critério, inicialmente proposto no final da década de 70 [6] para o teste de programas, mostrou-se um critério bastante flexível, podendo ser aplicado na validação de outros tipos de “entidades executáveis”, como Máquinas de Estado Finito [33, 34], Redes de Petri [35], Statecharts [36] ou especificações algébricas [37]. Experimentos mostram que esse critério é também bastante efetivo em revelar defeitos [17]. Embora teoricamente tenha sido demonstrado que esse critério e critérios de fluxo de dados são incomparáveis [10], empiricamente mostrou-se que, em geral, o teste de mutação inclui os critérios de fluxo de dados de Rapps e Weyuker [19, 21], ou seja, um conjunto de teste adequado ao teste de mutação tende a ser também adequado ao critério de Rapps e Weyuker. Já numa comparação empírica entre o teste de mutação e os Critérios Potenciais Usos, concluiu-se que mesmo na prática eles são incomparáveis [38].

Um dos sérios problemas associados ao teste de mutação é seu custo. O número de mutantes tende a ser grande, mesmo para programas pequenos. Porém, experimentos mostram também que esse custo pode ser sensivelmente reduzido, sem que se perca a efetividade em revelar defeitos. Isso é feito através da aplicação de critérios de mutação alternativos, conforme definidos por Wong e Mathur [18, 20, 39], Wong et al. [22] e Offut et al. [13].

A atividade de teste é composta, na verdade, de diversas fases. Cada uma dessas fases tem como objetivo a validação de diferentes aspectos do software. A primeira delas é a fase de teste de unidade, onde procura-se validar a implementação de cada uma das partes do software, chamadas de unidades ou módulos.

Ao agruparem-se as unidades que foram testadas separadamente, costumam surgir problemas chamados de erros de integração, em geral caracterizados por problemas na interação entre as unidades. Por isso, necessita-se do teste de integração, segunda etapa da atividade de teste. Por último, são realizados os testes de alto nível, onde busca-se validar o software como um todo, comparando-o com os requisitos especificados para o software e para o sistema onde o software deve atuar [1].

Critérios de avaliação de conjuntos de teste, funcionais, estruturais ou baseados em defeitos podem ser aplicados em cada uma das fases do teste como base para o projeto de casos de teste. Na fase de unidade, tem-se aplicado largamente as três classes de critérios. Já nas fases de integração e de alto nível, a prática tem se restringido à aplicação quase que exclusiva de critérios funcionais. Isso acontece principalmente porque critérios estruturais e baseados em defeitos derivam seus requisitos de teste limitando-os ao escopo das unidades. Por isso, casos de teste selecionados utilizando-se esses critérios, embora possam ser indicados para o teste de unidade, nem sempre o são para as outras fases de teste.

Em particular, nota-se que no teste de integração, cujo objetivo principal é validar as interações entre as unidades que compõem o software, necessita-se de critérios mais “fortes” que os critérios funcionais correntemente utilizados e que dêem indicações mais objetivas de como essas interações devem ser exercitadas de maneira completa, revelando assim possíveis erros de integração. Alguns autores procuram definir critérios chamados interprocedimentais, que definem requisitos de teste além do escopo de uma única unidade. Dessa forma, estabelecem-se requisitos que forcem a seleção de casos de teste que exercitem as interações entre as unidades, sendo assim indicados para o teste de integração. É o caso do trabalho de Haley e Zweben [40], de Linnenkugel e Müllerburg [41] e de Harrold e Soffa [32].

Esta tese propõe um critério interprocedimental baseado em mutações chamado Mutação de Interface. Esse critério é uma extensão do teste de mutação convencional utilizado no teste de unidade.

1.1 Motivação

De acordo com a introdução apresentada, podem ser destacados alguns pontos que motivaram a condução deste trabalho:

- Cada vez mais o software está presente em praticamente todas as áreas da atividade humana.
- A atividade de desenvolvimento de software está sujeita a erros e embora existam métodos, ferramentas e processos que procuram evitar que esses erros sejam introduzidos no produto de software, não se consegue evitá-los completamente.
- Dessa forma, a atividade de teste mostra-se essencial no âmbito da Engenharia de Software.
- Diversos critérios de adequação têm sido desenvolvidos para apoiar a seleção de casos de teste. Porém, com raras exceções, esses critérios estabelecem requisitos de teste no escopo de uma única unidade, não sendo capazes de exercitar aspectos da interação interprocedimental, essencial na fase de integração de software.
- Técnicas de teste são complementares e devem ser utilizadas em conjunto, também no teste de integração.
- Necessita-se de medidas objetivas para que se possa quantificar de forma objetiva a adequação de conjuntos de teste na fase de integração de teste. Em outras palavras, nota-se a necessidade da definição de critérios para essa fase de teste.
- O teste de mutação tem se mostrado um critério altamente efetivo, e embora o custo de sua aplicação seja inerentemente alto, pode-se ajustar esse custo de acordo com restrições estabelecidas pelo testador. Além disso, o critério é bastante flexível e tem sido aplicado na validação de diversos tipos de entidades, o que sugere que sua utilização no teste de integração possa também ser feita sem grandes problemas.
- O desenvolvimento de ferramentas de apoio ao teste de software é de vital importância pois permite que critérios de teste sejam utilizados na prática e aumenta a qualidade e a produtividade da atividade de teste.

1.2 Objetivos

Os objetivos principais desta tese podem ser definidos como:

- Estender, para o nível de integração de software, o critério Análise de Mutantes, que tem sido utilizado com sucesso no teste de unidade de software e na validação de especificações de sistemas reativos.
- Definir um conjunto de operadores de mutação para aplicação específica no nível de integração. Para isso é necessário que sejam identificados quais os tipos de erros que se desejam modelar com esses operadores, caracterizando-se o que são “erros de integração”.
- Definir critérios alternativos – a exemplo do que tem sido feito no nível de unidade – que possibilitem estabelecer estratégias de aplicação do critério no nível de integração de forma incremental, permitindo assim ajustar o critério a restrições de custo.
- Implementar uma ferramenta que apóie a aplicação do critério, bem como forneça facilidades para o uso dos critérios alternativos.
- Conduzir estudos empíricos que avaliem o critério quanto a custo e efetividade em revelar defeitos.

1.3 Organização da Tese

No Capítulo 2 são apresentados conceitos básicos sobre o teste de software e sobre critérios de adequação de conjuntos de teste. São apresentados também os principais trabalhos nessa área e que estão relacionados com o conteúdo desta tese.

No Capítulo 3 apresenta-se o critério Mutação de Interface, mostrando-se as idéias básicas que levaram à sua definição. São apresentados também os operadores de mutação elaborados para o critério. As principais características dos operadores, bem como uma análise da complexidade associada a sua aplicação e uma comparação com critérios estruturais de fluxo de dados são também apresentadas. Mostra-se ainda, a necessidade da elaboração de abordagens que restrinjam o número de mutantes, através da definição de critérios de Mutação de Interface alternativos.

Na prática, a aplicação de qualquer critério de teste requer sua automatização, ao menos parcial. No Capítulo 4 são apresentados aspectos da funcionalidade e da implementação da ferramenta de teste *PROTEUM/IM* que apóia a aplicação do critério Mutação de Interface, implementando os operadores de mutação propostos

e fornecendo as facilidades necessárias para a condução de estudos empíricos, bem como para a aplicação de critérios de Mutação de Interface alternativos.

No Capítulo 5 faz-se uma avaliação empírica da aplicação do critério Mutação de Interface através de dois estudos de caso. Esses estudos de caso mostram as medidas de efetividade em revelar erros e de custo da aplicação do critério em dois programas distintos e utilizando dois métodos ligeiramente distintos.

O Capítulo 6 sumariza os resultados obtidos, apresenta as conclusões e contribuições desta tese e aponta para trabalhos futuros decorrentes da proposição deste trabalho.

O Apêndice A descreve os defeitos encontrados no programa *SPACE* e utilizados em um dos experimentos do Capítulo 5.

Capítulo 2

Revisão Bibliográfica

Neste capítulo são apresentados os principais trabalhos relacionados com esta tese. Inicialmente, são feitas algumas definições de termos e algumas considerações básicas relacionadas com a atividade de teste e em particular com critérios de adequação de casos de teste.

Depois, apresentam-se alguns dos mais importantes critérios de teste, divididos nas três técnicas básicas: funcional, estrutural e baseada em defeitos. Discutem-se também fatores que têm sido utilizados para avaliar e comparar critérios de teste e ferramentas de teste disponíveis para apoiar a aplicação de alguns desses critérios.

Em seguida, são descritas as etapas básicas que constituem o teste de software, destacando-se a fase de teste de unidade e teste de integração, apresentando-se uma análise de critérios de teste mais adequados para cada uma dessas fases.

Finalizando o capítulo, discute-se como esta tese relaciona-se com os demais trabalhos nele abordados.

2.1 Definições Básicas

Várias interpretações podem ser dadas ao termo “teste”, mesmo dentro da Engenharia de Software. No escopo desta tese, chama-se de teste de software à atividade de, dado um programa **P**, executá-lo com um conjunto de dados de entrada

e verificar se seu comportamento é o desejado, de acordo com uma especificação **S**.

Dada uma especificação **S** que define o conjunto **D** (domínio dos dados de entrada), o conjunto **R** (intervalo dos dados de saída) e o mapeamento de **D** para **R**, $(\mathbf{D} \xrightarrow{\mathbf{S}} \mathbf{R})$, diz-se que o programa **P** está correto se o mapeamento $(\mathbf{D} \xrightarrow{\mathbf{P}} \mathbf{R})$ que **P** estabelece de **D** para **R** é equivalente a $(\mathbf{D} \xrightarrow{\mathbf{S}} \mathbf{R})$ [13]. Em outras palavras, um programa é dito correto se sua execução para todo elemento x de seu domínio de entrada apresenta os resultados esperados. Dados os programas **P** e **Q** que têm o mesmo domínio de entrada **D**, diz-se que **P** e **Q** são **equivalentes** se para todo $x \in \mathbf{D}$ tem-se que $\mathbf{P}(x) = \mathbf{Q}(x)$, ou seja a execução de **P** com x apresenta o mesmo resultado da execução de **Q** com x .

Dado $x \in \mathbf{D}$ (chamado de **dado de teste**) dá-se o nome de **saída esperada** a $\mathbf{S}(x)$. Ao par $\langle x, \mathbf{S}(x) \rangle$ chama-se **caso de teste**. Um **oráculo** é o instrumento capaz de, dado $\langle x, \mathbf{P}(x) \rangle$, decidir se $\mathbf{P}(x) = \mathbf{S}(x)$. Em outras palavras, um oráculo identifica se a saída produzida pela execução de **P** para um determinado dado de teste é a esperada ou não. Muitas vezes a função de oráculo é exercida pelo próprio testador, responsável por analisar os resultados da execução do programa para determinados dados de teste e decidir sobre sua correção.

Um **defeito** (fault) em um programa é um passo, processo ou definição de dados incorreto, como por exemplo, uma instrução ou comando incorreto. Um **engano** (mistake) é uma ação humana que produz um resultado incorreto, como por exemplo, uma ação incorreta tomada pelo programador. Um defeito pode levar um programa a produzir resultados diferentes do esperado, produzindo o que costuma-se chamar de **falha** (failure). Um **erro** (error) é uma diferença entre um valor corrente e o valor esperado, ou seja, qualquer estado intermediário ou resultado inesperado na execução de um programa constitui um erro. Esses termos estão definidos dessa maneira no padrão da IEEE número 610.12-1990 [42]. A literatura, porém, tem utilizado esses termos de diversas maneiras. Nesta tese procurou-se manter o significado desses termos. Entretanto, quando tratarem-se de referências a trabalhos de outros autores, os termos originais são mantidos, a não ser que possam causar dúvidas. Nesse caso, o significado de cada termo será explicitamente colocado.

2.2 Critérios de Adequação de Casos de Teste

De acordo com Myers [2], o objetivo da atividade de teste é achar erros no software. Portanto, o teste bem sucedido é aquele que consegue determinar casos de teste para os quais o programa em teste falhe. Assim, critérios para selecionarem-se e avaliarem-se casos de teste são fundamentais para o sucesso de tal atividade. 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.

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

Critério de adequação de casos de teste é um predicado para avaliar T para o teste de P .

Método de seleção de casos de teste é um procedimento para escolher casos de teste para o teste de P .

Para cada critério de adequação corresponde um método de seleção e vice-versa. Dado um critério C , corresponde a ele um método de seleção M_C que determina: selecione T tal que T seja adequado a C . Analogamente, dado o método de seleção M , define-se um critério C_M que estabelece: T é adequado se foi selecionado de acordo com M . Por isso, costuma-se utilizar o termo “critério de adequação de casos de teste” (ou simplesmente critério de teste) também para designar-se método de seleção [4].

Um exemplo simples de critério de teste é aquele que estabelece que T é adequado para o teste de P se a execução de P com todos os casos de teste em T faz com que cada um dos comandos de P seja executado pelo menos uma vez. Esse requisito serve também como um guia para a escolha de casos de teste.

De acordo com Zhu e Hall [43], a definição de um critério de adequação como um predicado que indica se um determinado conjunto de casos de teste é adequado ou não, não fornece uma gradação sobre a adequação do conjunto de teste; ou ele é um bom conjunto (adequado) ou um conjunto ruim (não adequado). De acordo com esses autores, um critério deveria ser definido como uma função contínua do conjunto de programas, especificações e conjuntos de teste, no intervalo $[0,1]$. Dessa maneira, dados o programa P , a especificação S , e o conjunto de teste T , denota-se $M_P^S(T) = r$, indicando que de acordo com o critério M , o conjunto de teste T é de adequação r para o programa P , de acordo com a especificação S . Por exemplo, para o critério que requer que todos os comandos sejam executados pelo

menos uma vez, pode-se considerar como medida de adequação a porcentagem de comandos do programa que foram executados pelo conjunto de teste.

Basicamente, pode-se classificar um critério de teste em três categorias distintas: funcional, estrutural ou baseada em defeitos. O que diferencia essas três categorias é o tipo de informação utilizada para derivar os requisitos de teste. Critérios funcionais utilizam apenas a especificação do programa; critérios estruturais utilizam informação sobre a estrutura do programa; e critérios baseados em defeitos utilizam o conhecimento sobre tipos específicos de defeitos que se pretende localizar. Cada um desses tipos de critérios é apresentado e exemplificado nas seções seguintes.

2.2.1 Critérios Funcionais

Critérios funcionais são também chamados de critérios **caixa preta** pois o software em teste é tratado como uma caixa da qual não se conhece o conteúdo, mas apenas seu lado externo, ou seja, as entrada que o programa aceita e as saídas que o programa deve produzir. Essa é a informação utilizada para derivar os requisitos a serem satisfeitos pelos casos de teste. Detalhes da implementação não são considerados.

Um exemplo de critério funcional é o **Particionamento em Classes de Equivalência**. Ele procura dividir o domínio de entrada em classes que têm como característica o fato de que a execução do programa com qualquer elemento de uma classe produz o mesmo resultado. Em outras palavras, se um caso de teste revela um erro, todos os elementos da mesma classe devem também revelar o mesmo erro. Dessa maneira pode-se selecionar apenas um subconjunto do domínio de entrada que, em tese, possui a mesma capacidade de revelar erros que teria o teste exaustivo (com todos os elementos do domínio). Na verdade, procura-se selecionar classes que aproximem-se dessa característica pois, em geral, é impossível caracterizar-se exatamente as classes de equivalência [2].

A **Análise de Valor Limite** acrescenta algumas características ao critério de Partição em Classes de Equivalência. Em vez de selecionar qualquer elemento de uma classe, os casos de teste são escolhidos nas fronteiras das classes pois é aí que se concentra um grande número de erros. Além disso, o espaço de saída do programa também é particionado e são exigidos casos de teste que produzam resultados nos limites dessas classes de saída [1].

Outro exemplo é a utilização do **Grafo de Causa-efeito**. Nesse critério são

primeiramente levantadas as possíveis condições de entrada (causas) e as possíveis ações (efeitos) do programa. Em seguida, é construído um grafo relacionando as causas e efeitos levantados. Esse grafo é convertido em uma tabela de decisão que é então utilizada na construção dos casos de teste [1].

Um dos problemas com critérios funcionais é que muitas vezes a especificação do programa é feita de forma descritiva e não formal. Dessa maneira, os requisitos de teste derivados de tais especificações são também de certa forma imprecisos e informais. Como consequência, 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. Por isso eles podem facilmente ser aplicados, tanto para unidades individuais, como para partes do software ou mesmo para o software como um todo, sendo assim aplicáveis em qualquer fase de teste.

2.2.2 Critérios Estruturais

Algumas vezes, critérios estruturais são chamados de **caixa branca** ou, mais adequadamente, de **caixa de vidro** pois o software pode ser comparado a uma caixa da qual, além do lado externo (especificação), deve-se conhecer o lado de dentro (estrutura da implementação) para que sejam estabelecidos os requisitos de teste.

Diversos critérios estruturais utilizam uma representação do programa chamada de **Grafo de Fluxo de Controle** ou **Grafo de Programa** para determinar as estruturas que devem ser cobertas pelos casos de teste. Um grafo de programa é um grafo orientado onde 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 da execução para nenhum comando dentro do bloco, exceto o primeiro. A Figura 1 mostra um exemplo simples de um programa e seu grafo de fluxo de controle.

Utilizando-se o grafo de programa, podem ser selecionados componentes que devem ser executados, caracterizando assim os critérios estruturais. Critérios baseados na complexidade utilizam informação sobre a complexidade do grafo (conseqüentemente do programa) para derivar requisitos de teste. Um critério desse tipo, bastante conhecido, é o critério de McCabe que utiliza a complexi-

Figura 1: Exemplo de grafo de fluxo de controle

dade ciclomática para estabelecer um conjunto de caminhos do grafo linearmente independentes que devem ser exercitados pelos casos de teste [1].

Crítérios baseados no fluxo de controle usam apenas características de controle da execução do programa para determinar quais estruturas são requeridas. Como exemplo de tais critérios, podem ser citados os critérios **todos-nós** e **todos-arcos**. O primeiro requer que cada vértice do grafo de programa seja executado pelo menos uma vez pelos casos de teste, o que corresponde à execução – pelo menos uma vez – de cada comando do programa. O segundo requer que cada aresta seja exercitada pelo menos uma vez, o que corresponde à execução de todos os ramos do programa [1].

Crítérios baseados no fluxo de dados associam informação sobre o uso das variáveis ao grafo de programa, determinando associações entre pontos onde um valor é atribuído a uma variável (chamados de **definição** da variável) e pontos onde esse valor é utilizado (chamados de **uso** da variável). Essas associações devem então ser exercitadas pelos casos de teste. Rapps e Weyuker [3] propuseram uma família de critérios baseados no fluxo de dados. Os principais critérios dessa família e seus requisitos são:

Todas-Definições: requer que cada definição de variável seja exercitada pelo menos por um uso dessa variável.

Todos-Usos: requer que todas as associações entre uma definição de variável

e todos os usos da mesma 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.

Todos-Du-Caminhos: requer que toda associação entre uma definição de variável e um uso dessa variável seja exercitada por todos caminhos sem laço livres de definição que cubram essa associação.

A família de critérios de Rapps e Weyuker define ainda variações desses critérios que levam em conta apenas usos predicativos (p-usos) ou usos computacionais (c-usos) das variáveis. Um uso predicativo é aquele que ocorre em um predicado que determina um desvio na execução do programa. Os demais usos são usos computacionais. Dessa maneira, são definidos os critérios Todos-P-Usos, Todos-P-Usos/Algum-C-Uso e Todos-C-Usos/Algum-P-Uso.

Maldonado [4] propõe uma família de critérios semelhantes aos propostos por Rapps e Weyuker. Esses critérios, porém, não exigem o uso explícito de uma variável para que uma associação seja estabelecida. Qualquer ponto onde o valor de uma variável possa ser utilizada determina uma associação com o ponto onde esse valor foi definido. Essa família de critérios é chamada de Critérios Potenciais Usos.

Uma das vantagens de critérios de fluxo de dados sobre os de fluxo de controle apontada por Rapps e Weyuker [3], é o fato de que os primeiros são construídos de forma que associações críticas entre pontos onde variáveis são definidas e onde os seus valores são usados são examinadas durante o teste. Dessa maneira, variáveis de entrada são monitoradas, seus valores intermediários exercitados, até o ponto em que são finalmente utilizados para produzir alguma saída. Além disso, segundo as autoras, da mesma maneira que se pode argumentar que um teste no qual não foram executados todos os comandos do programa é um teste ruim, pode-se argumentar que um teste que não exercita, para cada cálculo, todos os valores produzidos, também é de má qualidade.

Costuma-se dizer que com tais critérios mede-se a **cobertura** dos casos de teste em relação ao programa sendo testado. A medida de cobertura é dada pela porcentagem de associações cobertas pelo conjunto de teste.

Um dos problemas relativos aos critérios de fluxo de dados é a existência de associações não executáveis, ou seja, associações que são requeridas pelo critério mas para as quais não existe nenhum dado de teste que as exercite. De maneira geral, determinar se uma associação é ou não executável é um problema indecidível, o que torna impossível a completa automatização da aplicação dos critérios [8].

A família de critérios de Cobertura de Condições propostos por Myers [2] requer que casos de teste sejam selecionados de modo a exercitar cada expressão que controla os desvios de programas, como em comandos de seleção ou em comandos repetitivos. O critério de **Cobertura de Decisão** requer que cada uma dessas expressões (chamadas pelo autor de decisões) assuma pelo menos uma vez o valor FALSO e pelo menos uma vez o valor VERDADEIRO. Myers chama de “condição” cada sub-expressão que compõe uma decisão e define o critério de **Cobertura de Condição** que exige que cada uma delas assuma pelo menos uma vez o valor FALSO e pelo menos uma vez o valor VERDADEIRO. O critério de **Cobertura de Decisão e Condição** é uma combinação das duas anteriores, exigindo casos de teste que exercitem cada decisão e cada condição. Finalmente, o critério de **Cobertura de Condições Múltiplas** requer que cada possível combinação dos valores das condições seja exercitada pelo menos uma vez.

Ainda nessa classe, diversos outros critérios podem ser citados, como o critério LCAJ (Linear Code Sequence and Jump) de Woodward et al. [44], o critério “boundary-interior” de Howden [45], a família de critérios K-tuplas requeridas de Ntafos [46], entre outros.

Como já citado, uma das desvantagens do teste estrutural é a existência de caminhos requeridos e que são não executáveis. Existe também o problema de caminhos ausentes, ou seja, quando uma certa funcionalidade deixa de ser implementada no programa, não existe um caminho que corresponda àquela funcionalidade, e como consequência, nenhum caso de teste será requerido para exercitá-la. Por outro lado, esses critérios estabelecem precisamente os requisitos de teste a serem exercitados, em termos de caminhos, associações definição-uso, ou outras estruturas do programa. Por isso, esses critérios fornecem medidas bastante objetivas sobre a adequação de um conjunto de teste, expressa pelo número desses requisitos que foram ou não cobertos pelos casos de teste, o que torna possível automatizar a sua aplicação.

Os critérios estruturais têm sido utilizados principalmente no teste de unidade. Isso porque os requisitos de teste por eles exigidos limitam-se ao escopo de uma unidade. Critérios estruturais baseados em fluxo de dados foram estendidos para o nível interprocedimental e são descritos adiante, neste capítulo.

2.2.3 Critérios Baseados em Defeitos

Critérios baseados em defeitos utilizam informação sobre defeitos frequentemente introduzidos no software durante o desenvolvimento e, mais especificamente, no

processo de codificação do software [47].

O critério de **Semeadura de Erros** é um exemplo de critérios baseados em defeitos. Inicialmente são introduzidos defeitos de maneira uniforme por todo código do programa em teste. A adequação de um conjunto de casos de teste é medida pela sua capacidade em revelar defeitos semeados. A idéia é que a relação entre o número de defeitos semeados revelados, o número total de defeitos semeados e o número de defeitos naturais revelados deve dar uma indicação do número de defeitos naturais restantes [5].

Outros critérios desta classe são os baseados em mutações. A **Análise de Mutantes** – ou **mutação forte** ou ainda simplesmente **teste de mutação** – é o mais popular deles. Dado o programa P , esse critério considera um conjunto de programas alternativos $\Phi(P)$, do qual P é membro. Esse conjunto é chamado de **vizinhança** de P . Com esse conjunto procura-se mostrar que P não contém erros através de um processo de eliminação, mostrando que nenhum programa na vizinhança $\Phi(P)$, não equivalente a P , é correto. Isso deve ser feito utilizando-se um conjunto de teste T para o qual todos os membros de $\Phi(P)$, não equivalentes a P , falhem em pelo menos um caso de teste. Tal conjunto é definido como:

Um conjunto de casos de teste T é adequado para P em relação a $\Phi(P)$ se para cada programa Q pertencente a $\Phi(P)$, ou Q é equivalente a P ou Q difere de P em pelo menos um caso de teste t pertencente a T [48].

Deve-se observar que esta definição não aborda o problema da indecidibilidade sobre a equivalência entre dois programas. A relação entre a correção de P em relação a F , que especifica a função a ser computada, e a adequação de T é dada pelo seguinte teorema:

Dado um programa P e uma vizinhança $\Phi(P)$, se algum programa em $\Phi(P)$ computa a função correta F , especificada para P , e T é adequado para P em relação a $\Phi(P)$ e P coincide com F em T , então P é correto [48].

Ao tomar-se $\Phi(P)$ como sendo o conjunto de todos os programas possíveis numa determinada linguagem, garante-se que existe pelo menos um programa $Q \in \Phi(P)$ que computa a função F e, dessa forma, ao encontrar-se um conjunto de teste T adequado, consegue-se provar a correção de P , se P computa F para todos os casos de teste de T . O problema porém, é que a escolha de tal vizinhança

torna impossível mostrar a adequação de qualquer conjunto $\mathbf{T}$ pelo fato de ser essa uma vizinhança infinita [49]. Para vizinhanças infinitas, nem sempre existe um conjunto $\mathbf{T}$ que seja adequado. Pior que isso, existem vizinhanças para as quais não se consegue determinar se existe ou não um conjunto de teste adequado [5].

Para vizinhanças Φ finitas, é demonstrado que existe sempre um conjunto $\mathbf{T}$ que seja adequado para $\mathbf{P}$ em relação a $\Phi(\mathbf{P})$. É importante notar-se porém, que ao se limitar a vizinhança a um subconjunto de todos os possíveis programas, abandona-se a idéia de provar a correção de $\mathbf{P}$ e procura-se mostrar sua “correção relativa” em relação a um pequeno conjunto de programas. Na Análise de Mutantes, a **Hipótese do Programador Competente** afirma que qualquer programa, produzido por um programador competente, ou está correto ou está próximo da correção. Dessa forma, pode-se, durante o teste, concentrar as atenções sobre uma vizinhança bem menor do que todo o universo possível de programas, ou seja, pode-se concentrar a atenção naqueles programas que são “parecidos” com $\mathbf{P}$. Pode-se ver essa restrição da vizinhança de $\mathbf{P}$ como a delimitação dos tipos de erros que se deseja considerar na atividade de teste. Além disso, segundo o **Efeito de Acoplamento**, que é a segunda hipótese fundamental do teste de mutação, conjuntos de teste que são adequados a uma vizinhança $\Phi(\mathbf{P})$ que representem defeitos comuns de programação, devem ser também bons conjuntos para revelar outros tipos de defeitos, mais complexos [6].

Visto que para $\Phi(\mathbf{P})$ finito sempre existe um conjunto $\mathbf{T}$ adequado para $\mathbf{P}$, resta determinar se existem procedimentos para: 1) dado um conjunto $\mathbf{T}$, determinar se ele é adequado a $\mathbf{P}$ e $\Phi(\mathbf{P})$; e 2) construir $\mathbf{T}$ adequado a $\mathbf{P}$ e $\Phi(\mathbf{P})$. Mais precisamente, definem-se

- Um procedimento $A(\mathbf{P}, \mathbf{T})$ é um **aceitador** se ele retorna VERDADEIRO quando $\mathbf{T}$ é adequado para $\mathbf{P}$ em relação a $\Phi(\mathbf{P})$ e retorna FALSO caso contrário.
- Um procedimento $G(\mathbf{P})$ é um **gerador** se ele produz um conjunto de teste $\mathbf{T}$ que é adequado para $\mathbf{P}$ em relação a $\Phi(\mathbf{P})$.

Budd [5] mostra que sempre que existir um aceitador existe também um gerador e vice-versa. Porém, isso nem sempre acontece. A existência desses dois procedimentos está condicionada à existência de um **Verificador de Equivalência** $S(\mathbf{P}, \mathbf{Q})$ que responde verdadeiro quando os programas $\mathbf{P}$ e $\mathbf{Q}$ são equivalentes e falso caso contrário. Sabe-se que para o caso geral o problema de equivalência é indecidível e portanto não se pode construir aceitadores ou geradores de casos de

teste, baseados no teste de mutação. Por outro lado, o problema de equivalência nesse contexto fica restrito a decidir se $\mathbf{P}$ é ou não equivalente aos demais elementos de $\Phi(\mathbf{P})$, os quais são, em geral, bastante parecidos com $\mathbf{P}$. Isso faz com que algumas heurísticas ou, em última instância, a intervenção do testador possam ser utilizadas para que a equivalência seja decidida, tornando possível a criação dos aceitadores e geradores.

Na prática, dado um programa $\mathbf{P}$ que se deseja testar, são criados vários programas $\mathbf{M}_1, \mathbf{M}_2, \dots, \mathbf{M}_N$ chamados de **mutantes de $\mathbf{P}$** . Esses mutantes são programas semelhantes a $\mathbf{P}$ que possuem apenas pequenas alterações sintáticas em relação a $\mathbf{P}$, que podem ser vistas como a introdução de defeitos simples em $\mathbf{P}$. Por exemplo, se $\mathbf{P}$ contém a expressão “ $a \leq b$ ”, um de seus mutantes terá no lugar dessa, a expressão “ $a < b$ ”, outro terá “ $a > b$ ”, e assim por diante. O programa original é executado com o conjunto de casos de teste $\mathbf{T}$, cuja adequação se deseja avaliar. Caso nenhum erro seja revelado, então os mutantes são executados com $\mathbf{T}$. Se um mutante $\mathbf{M}_i$ apresenta resultado diferente de $\mathbf{P}$ para algum caso de teste t em $\mathbf{T}$ diz-se que esse mutante está morto. Nesse caso, t conseguiu revelar a presença do defeito introduzido em $\mathbf{P}$ para criar o mutante. Por outro lado, se $\mathbf{M}_i$ apresenta resultado idêntico a $\mathbf{P}$, diz-se que ele continua vivo. Isso pode acontecer por dois motivos: ou porque $\mathbf{T}$ não possui um caso de teste capaz de distinguir $\mathbf{M}_i$ de $\mathbf{P}$ ou porque eles são equivalentes. No primeiro caso, novos casos de teste podem ser adicionados a $\mathbf{T}$ para matar o mutante. Já os mutantes equivalentes são as “associações não executáveis” do teste de mutação. Ou seja, além de não se poder distinguir tal mutante, não se consegue, em geral, determinar automaticamente sua equivalência.

O nível de adequação de $\mathbf{T}$ é dado pelo **escore de mutação**, que relaciona o número de mutantes mortos com o número de mutantes não equivalentes:

$$ms(T, P) = \frac{\# \text{ mutantes mortos}}{\# \text{ total mutantes} - \# \text{ mutantes equivalentes}}$$

Os tipos de defeitos introduzidos no programa em teste para criação dos mutantes é determinado por **operadores de mutação**. Esses operadores de mutação dependem da linguagem na qual o programa é desenvolvido e procuram modelar defeitos comuns para a linguagem. Foi definido para a linguagem FORTRAN um conjunto de 22 operadores implementados na ferramenta Mothra [27]. Para a linguagem C, foram definidos 77 operadores de mutação [50] implementados na ferramenta Proteum [29, 30, 31]. Um exemplo do tipo de mutação utilizada está mostrado na Figura 2, extraída do artigo de DeMillo [6]. Nela, cada linha

precedida pelo “ Δ ” representa um mutante da linha correspondente do programa. Note-se que cada um dos quatro mutantes representa um programa diferente.

```
          FUNCTION MAX(M,N)
1         MAX = M
 $\Delta$       MAX = N
 $\Delta$       MAX = ABS(N)
2         IF (N .GT. M) MAX = N
 $\Delta$       IF (N .LT. M) MAX = N
 $\Delta$       IF (N .GE. M) MAX = N
3         RETURN
```

Figura 2: Exemplo de mutações para programas FORTRAN

Um dos problemas para a aplicação do critério é o alto número de mutantes que podem ser gerados, mesmo para programas de tamanho reduzido. Budd [51] estima que – de modo grosseiramente aproximado – o número de mutantes para um determinado programa é proporcional ao produto do número de variáveis pelo número de referências a essas variáveis. Segundo estudos realizadas por Offut et al. [13], também o número de linhas de código e mesmo o número de funções sendo testadas podem ser incorporados na projeção do número de mutantes para um determinado programa. De qualquer modo, o resultado desse cálculo tende a ser um valor elevado. Como cada um dos mutantes tem que ser executado com pelo menos um e potencialmente com todos os elementos no conjunto de teste para que seu comportamento seja comparado com o comportamento do programa original, pode-se ter um custo computacional muito alto para a aplicação do critério.

Diversas soluções têm sido propostas para esse problema. Algumas delas envolvem a utilização de hardware de alto desempenho para a execução dos mutantes [52, 53, 54, 55]. Se por um lado essas abordagens podem reduzir o custo de aplicação em termos de tempo de execução dos mutantes, por outro lado eleva o custo do teste em termos de recursos necessários, principalmente de hardware. Assim, esse tipo de solução só seria justificável para o teste de sistemas com requisitos de teste rigorosíssimos, que justificariam tal investimento na atividade de teste.

As abordagens que têm sido mais exploradas procuram reduzir o número de mutantes a serem executados e analisados. A idéia central é que diversos mutantes podem levar à seleção de um mesmo caso de teste – ou em outras palavras, um caso de teste distingue vários mutantes – caracterizando uma “redundância” de mutantes. Desse modo, pode-se utilizar subconjuntos reduzidos de mutantes que levem à seleção de conjuntos de teste tão efetivos quanto aqueles criados por um conjunto mais completo de mutantes.

Acree [56] e Budd [51] sugerem que essa redução no número de mutantes pode ser alcançada através da **mutação aleatória**, ou seja, através da seleção aleatória de uma pequena porcentagem de mutantes. Por exemplo, experimentos conduzidos por ambos para programas FORTRAN mostraram que conjuntos de teste adequados para um conjunto de 10% dos mutantes gerados pela ferramenta Mothra alcançavam escores de mutação superiores a 99% em relação ao conjunto completo de mutantes. Tais resultados foram confirmados por experimentos realizados por Wong para programas FORTRAN [20, 18] e também para programas C [22]. Por outro lado, segundo argumenta Mathur [57], tal abordagem não leva em conta características particulares de cada operador, em termos de efetividade em revelar erros. Por isso, Mathur propõe também o uso de **mutação restrita**. Nesse caso, somente alguns poucos operadores de mutação são utilizados. Os mesmos experimentos de Wong citados acima exploram também essa abordagem e mostram que ela apresenta resultados semelhantes à seleção randômica de mutantes, sendo ambas boas opções para a redução do número de mutantes.

Offut et al. [13] propõem um método chamado de **mutação seletiva** que também utiliza apenas alguns operadores de mutação na aplicação do teste. O critério de seleção de quais operadores devem ser utilizados é o número de mutantes que cada operador gera. Assim, segundo os autores, deve-se priorizar o uso daqueles operadores que geram o menor número de mutantes, evitando aqueles que tendem a criar muitos mutantes. Na verdade, cria-se desta maneira uma hierarquia de “sub-critérios”, chamados de N -seletivos – que aplicam todos os operadores exceto aqueles N que tendem a gerar mais mutantes. Por exemplo, o critério 1-seletivo utiliza todos os operadores exceto aquele que tende a gerar maior número de mutantes, o 2-seletivo despreza os dois operadores que mais geram mutantes, e assim por diante. Segundo avaliação de Offut et al. para um conjunto de programas FORTRAN [13], aplicando-se o critério de mutação 6-seletivo para os operadores da ferramenta Mothra obtém-se uma redução de até 60% no número de mutantes gerados. Os conjuntos de teste 6-seletivo adequados apresentam também alta adequação em relação aos mutantes gerados pelo conjunto completo de operadores de mutação. Na média, os conjuntos 6-seletivo adequados atingiram 99.71% de adequação ao critério não-seletivo (completo).

No critério de **mutação fraca** [7] considera-se o programa **P** formado por componentes que correspondem a estruturas simples como referência a uma variável, expressão aritmética, etc. Dado tal componente **C**, cria-se a partir de uma mutação em **C**, um mutante que contém o componente **C'**. O critério requer algum caso de teste que faça com que o “resultado” de **C** e **C'** difiram. Em outras

palavras, o estado interno de $\mathbf{P}$ e do mutante, imediatamente após a execução do componente, deve diferir. A vantagem dessa abordagem é que os mutantes não precisam ser executados por completo para que sejam distinguidos de $\mathbf{P}$, uma vez que resultados intermediários da execução do programa são utilizados para essa decisão. Além disso, em muitos casos torna-se fácil determinar um caso de teste para distinguir determinado mutante sem executá-lo. Por exemplo, dado o componente “ $(a + b)$ ” que é alterado para “ $(c + b)$ ”, nota-se que qualquer caso de teste onde o valor da variável a difere do valor da variável c distingue o mutante. A desvantagem em relação à mutação forte é que conjuntos de teste adequados em relação à mutação fraca podem não sê-lo em relação à mutação forte, visto que tais conjuntos garantem que o estado intermediário de cada mutante difere em relação ao programa original mas não garantem que essa diferença se propague até a saída do programa e, por isso, não necessariamente distinguem os “mutantes fortes”.

2.3 Avaliação e Comparação entre Critérios

Um dos pontos importantes quando se fala sobre critérios de adequação é determinar o quão eficaz um determinado critério é. Várias formas têm sido propostas na literatura. Algumas delas são revistas nesta seção.

Segundo Wong e Mathur [18], três fatores podem ser utilizados na avaliação de um critério de teste. O primeiro é o **custo**, que determina o “trabalho” requerido para que o critério seja satisfeito para determinado programa $\mathbf{P}$. O segundo é a **efetividade em revelar erros**, ou seja, a probabilidade de, usando-se determinado critério, revelarem-se defeitos no programa em teste. O terceiro é a dificuldade de satisfação ou também chamado de **força** do critério que, dados critérios $\mathbf{C}$ e $\mathbf{C}'$, estabelece formas de comparação entre eles. Para avaliar esses fatores pode-se ter uma abordagem teórica ou uma abordagem empírica. As seções seguintes descrevem estudos em cada uma dessas duas categorias.

2.3.1 Estudos Teóricos

Na abordagem teórica, os fatores de custo, efetividade e força são avaliados analisando-se características dos requisitos de teste exigidos pelo critério. Para avaliar-se analiticamente o custo de um critério tem-se utilizado uma medida chamada de **complexidade** do critério. A complexidade de um critério $\mathbf{C}$ é dada pelo número de casos de teste requeridos para satisfazer o critério, no pior caso.

Weyuker [58], por exemplo, mostrou que a complexidade dos critérios da Família de Fluxo de Dados é polinomial no número de comandos de decisão do programa. Esse resultado foi contestado por Maldonado [4] que mostra que no pior caso, o número de casos de teste requeridos é exponencial no número de comandos de decisão, tanto para os critérios Fluxo de Dados como para os critérios Potenciais Usos.

No caso do teste de mutação tem-se utilizado o número de mutantes gerados para estimar a complexidade do critério. Obviamente, esse número depende do conjunto de operadores de mutação utilizado. Estudos mostraram que operadores de mutação que são aplicados sobre ocorrências de variáveis no programa são responsáveis pela maior parte dos mutantes. Por isso, Budd [51] determinou que o número esperado de mutantes é proporcional ao número de variáveis do programa vezes o número de referências às variáveis. Essa aproximação, reconhecidamente grosseira, tem sido adotada por outros pesquisadores e pode ser usada como medida de custo (complexidade) para o critério.

Para determinar-se a força de um critério de teste tem-se utilizado a relação de inclusão para compará-lo com outros critérios. Dados os critérios C e C' , diz-se que C inclui C' se para qualquer programa P , todo conjunto de teste C -adequado é também C' -adequado. O critério C inclui estritamente C' , denotado por $C \Rightarrow C'$, se C inclui C' e C' não inclui C . Se nem $C \Rightarrow C'$ nem $C' \Rightarrow C$, diz-se que C e C' são incomparáveis.

Rapps e Weyker [3] mostraram que seus critérios de fluxo de dados formam uma hierarquia de critérios, de acordo com a relação de inclusão. O mesmo foi feito por Maldonado [4], incluindo nessa hierarquia a família de critérios Potenciais Usos. A Figura 3 mostra a hierarquia formada pelos critérios de Rapps e Weyuker e a Figura 4 mostra a hierarquia com a presença dos critérios Potenciais Usos.

A comparação baseada na relação de inclusão entre critérios que contêm bases comuns, como critérios de fluxo de controle e fluxo de dados, em geral pode ser feita de maneira simples, como aponta Zhu [12]. Por exemplo, dados os critérios todos-p-usos e todas-arestas, pode-se mostrar que todos-p-usos $\Rightarrow$ todas-arestas da seguinte maneira:

1. admite-se que nenhuma variável do programa é usada sem ter sido definida;
2. admite-se que toda condição que determina um desvio no programa possui pelo menos um uso de variável;

Figura 3: Hierarquia dos critérios de Rapps e Weyuker de acordo com a relação de inclusão

Figura 4: Hierarquia dos critérios de Rapps e Weyuker e Potenciais Usos de acordo com a relação de inclusão

3. para cada desvio do programa tem-se a situação mostrada na Figura 5;
4. o critério todos-p-usos requer que um caminho do nó i à aresta (j, k) e um caminho do nó i à aresta (j, l) sejam executados;
5. como isso acontece para todos os desvio no programa, tem-se a cobertura de todos os desvios.

Figura 5: Exemplo de como o critério todos-p-usos inclui o critério todas-arestas

Note-se que algumas restrições são impostas para que se possa mostrar a inclusão. Nem sempre essas restrições correspondem à realidade. Por exemplo, Horgan e London [28] mostram que se o programa em teste não atende à restrição de que cada comando de decisão faz referência a pelo menos uma variável – o que é comum, por exemplo, na linguagem C – então a situação mostrada na Figura 5 nem sempre é verdadeira e o critério todos-p-usos não inclui todas-arestas. Nesse caso, a hierarquia de critérios fica alterada, conforme mostrado na Figura 6.

Por outro lado, a comparação baseada na relação de inclusão entre critérios de naturezas diferentes, em geral, não é feita com tanta facilidade. É o que mostra Wong [59] quando demonstra analiticamente que o critério Análise de Mutantes é incomparável com os critérios de fluxo de dados todos-usos, todos-p-usos e todos-c-usos. O problema nesse caso é que, ao contrário dos critérios de fluxo de dados, o teste de mutação não requer explicitamente a execução de sub-caminhos do programa, tornando difícil a comparação entre esses critérios.

A relação de inclusão não é a única relação utilizada para a comparação de critérios. Na verdade, segundo Frankl e Weyuker [9], essa não é a melhor relação para se usar em tais comparações pois o fato de um critério C incluir outro critério C' não indica necessariamente que C seja melhor que C' . Mais precisamente, a inclusão não indica que a efetividade em revelar erros de C seja superior à de C' . Frankl e Weyuker [9] propõem algumas outras relações para a comparação entre critérios e analisam quais delas implicam, de acordo com medidas probabilísticas, numa maior efetividade em revelar erros.

Figura 6: Hierarquia dos critérios de Rapps e Weyuker e Potenciais Usos quando não se tem a restrição de que cada comando de decisão possui pelo menos uma referência a variável

Dados o programa $\mathbf{P}$ e a correspondente especificação $\mathbf{S}$, um critério de teste determina uma divisão do domínio de entrada em k subdomínios $\mathbf{D}_i$, em geral não disjuntos. Por exemplo, uma associação requerida por um critério de fluxo de dados determina um subconjunto dos dados de entrada que executam essa associação. No teste de mutação, um mutante determina um subdomínio que distingue esse mutante. Seja $d_i = |\mathbf{D}_i|$ e $m_i =$ número de casos de teste em $\mathbf{D}_i$ que fazem $\mathbf{P}$ falhar de acordo com $\mathbf{S}$. Frankl e Weyuker definem três medidas probabilísticas para o critério $\mathbf{C}$:

$$M_1(\mathbf{C}, P, S) = \max_{1 \leq i \leq k} \left(\frac{m_i}{d_i} \right)$$

é o limite inferior da probabilidade que um conjunto $\mathbf{C}$ -adequado revele um defeito:

$$M_2(\mathbf{C}, P, S) = 1 - \prod_{i=1}^k \left(1 - \frac{m_i}{d_i} \right)$$

é a probabilidade de que um conjunto $\mathbf{C}$ -adequado revele pelo menos um defeito, quando a seleção de casos de teste é feita escolhendo-se exatamente 1 caso de teste em cada subdomínio $\mathbf{D}_i$:

$$M_3(\mathbf{C}, P, S, n) = 1 - \prod_{i=1}^k \left(1 - \frac{m_i}{d_i} \right)^n$$

é uma generalização de M_2 selecionando-se n casos de teste em cada subdomínio.

Essas medidas consideram que, uma vez determinados os subdomínios, casos de teste são selecionados aleatória e uniformemente nesses domínios. Elas procuram de certa forma determinar de maneira analítica a efetividade em revelar erros de um critério, de acordo com o tipo de divisão que este cria no domínio de entrada.

Frankl e Weyuker definem cinco novas relações entre critérios e analisam se o fato de que $R(\mathbf{C}, \mathbf{C}')$ (a relação R se aplica de $\mathbf{C}$ sobre $\mathbf{C}'$) implica ou não que $\mathbf{C}$ é superior a $\mathbf{C}'$, de acordo com as medidas probabilísticas propostas. As relações propostas e as conclusões das autoras estão sumarizadas na Tabela 1.

Como exemplo, a relação *restringe* é definida da seguinte maneira:

Sejam $SD_{\mathbf{C}_1}(\mathbf{P}, \mathbf{S})$ e $SD_{\mathbf{C}_2}(\mathbf{P}, \mathbf{S})$ os multiconjuntos formados pelos subdomínios que os critérios $\mathbf{C}_1$ e $\mathbf{C}_2$ respectivamente determinam para $\mathbf{P}$ e $\mathbf{S}$. $\mathbf{C}_1$ *restringe* $\mathbf{C}_2$ para $\mathbf{P}$ e $\mathbf{S}$, se para todo subdomínio $\mathbf{D} \in SD_{\mathbf{C}_2}$ existe um subdomínio $\mathbf{D}' \in SD_{\mathbf{C}_1}$ tal que $\mathbf{D} \subseteq \mathbf{D}'$. $\mathbf{C}_1$ *universalmente restringe* $\mathbf{C}_2$ se, para qualquer $\mathbf{P}$, $\mathbf{C}_1$ *restringe* $\mathbf{C}_2$.

Tabela 1: Relações definidas por Frankl e Weyuker e suas implicações quanto à efetividade em revelar erros

Relação	$M_1(\mathbf{C}) \geq M_1(\mathbf{C}')?$	$M_2(\mathbf{C}) \geq M_2(\mathbf{C}')?$	$M_3(\mathbf{C}) \geq M_3(\mathbf{C}')?$
$\mathbf{C}$ <i>restringe</i> $\mathbf{C}'$	nem sempre	nem sempre	nem sempre
$\mathbf{C}$ <i>cobre</i> $\mathbf{C}'$	nem sempre	nem sempre	nem sempre
$\mathbf{C}$ <i>particiona</i> $\mathbf{C}'$	sempre	nem sempre	nem sempre
$\mathbf{C}$ <i>cobre propriamente</i> $\mathbf{C}'$	nem sempre	sempre	nem sempre
$\mathbf{C}$ <i>particiona propriamente</i> $\mathbf{C}'$	sempre	sempre	nem sempre

Em artigo posterior, Frankl e Weyuker [60] utilizam as relações para comparar alguns dos critérios mais populares. Embora tais estudos dêem pistas de como se comportam critérios de teste e mesmo de quais diretivas devem ser seguidas ao se definirem critérios de teste, na prática torna-se difícil a aplicação dessas relações para a avaliação teórica de critérios mais elaborados. Como apontam as próprias autoras, mais trabalho analítico e experimental deve ser realizado até que se possam transpor para o mundo real os resultados até agora obtidos.

2.3.2 Estudos Empíricos

Outra maneira – mais pragmática – para avaliar e comparar critérios é através de estudos empíricos que utilizam estudos de casos onde os critérios são aplicados e

avaliados. Nesta seção são comentados alguns desses trabalhos.

Para verificar empiricamente a estimativa de Budd [51] sobre a complexidade do teste de mutação, Offut et al. [13] geraram mutantes para 28 programas Fortran utilizando todos os 22 operadores de mutação implementados na ferramenta de teste Mothra [27]. Os autores utilizaram diversos modelos de regressão linear utilizando como variáveis: a) número de linhas de código do programa; b) número de variáveis; c) número de referências às variáveis; d) número de desvios; e e) métrica de McCabe [1]. Como conclusão, obteve-se que a melhor equação para estimar o número de mutantes gerados é:

$$mutants = \beta_0 + \beta_1 \times Vars + \beta_2 \times Varrefs + \beta_3 \times Vars \times Varrefs$$

onde *Vars* é o número de variáveis e *Varrefs* é o número de referências às variáveis. Esse resultado, de certa forma, confirma a estimativa de Budd, que aponta o número de mutantes como sendo proporcional ao número de variáveis vezes o número de referências às variáveis.

Em outro experimento, Wong e Mathur [17] avaliaram experimentalmente a efetividade de alguns critérios de teste. Para isso, selecionaram cinco programas com defeitos conhecidos e criaram 30 conjuntos de teste, adequados para cada um dos critérios: análise de mutantes, mutação restrita, mutação aleatória com apenas 10% dos mutantes e critério todos-usos. A efetividade em revelar erros de cada critério pode ser medida da seguinte forma:

$$\frac{\text{número de conjuntos de teste } C\text{-adequados que revelam um defeito}}{\text{número total de conjuntos de teste } C\text{-adequados}} \times 100\%$$

Assim, Wong e Mathur avaliaram a capacidade de cada critério em criar conjuntos de teste que revelam um defeito. Os resultados mostraram que o critério análise de mutantes foi o mais efetivo com 96.33% dos conjuntos de teste revelando um defeito. Em seguida vêm mutação restrita com 87.67%, todos-usos com 83% e mutação aleatória-10% com 80.96%.

Além de efetividade em revelar erros, Mathur e Wong [19] mostram que o custo e a força dos critérios também podem ser comparados experimentalmente. Nesse experimento os autores selecionaram alguns programas e criaram para cada um deles 30 conjuntos de teste adequados ao teste de mutação (TM-adequados) e 30 todos-usos-adequados. Depois, mediram a adequação dos conjuntos TM-adequados em relação ao critério todos-usos e vice-versa. Ou seja, verificou-se

através desse experimento como, na prática, a relação de inclusão se aplica para esses dois critérios. Os resultados mostraram que: 1) nenhum conjunto todos-usos adequado era TM-adequado; e 2) na maioria dos casos os conjuntos TM-adequados eram todos-usos adequados. Esses dados mostram que, na prática, o teste de mutação inclui o critério todos-usos. O custo de cada critério foi medido pelo número de casos de teste requerido por cada critério. Nesse caso, o resultado favorece o critério de fluxo de dados pois o número de casos de teste requerido para satisfazer o teste de mutação foi entre 2 a 3,5 vezes maior que para o critério todos-usos, dependendo do programa testado. Experimento com os mesmos objetivos foi conduzido por Souza [38] comparando o teste de mutação e o critério todos-potencias-usos de Maldonado [4]. Neste caso, chegou-se à conclusão que mesmo empiricamente esses dois critérios são incomparáveis.

Experimento semelhante foi conduzido por Wong et al. [20, 22] comparando o teste de mutação com os critérios de mutação alternativos. Nesse caso, mostrou-se que reduzindo-se o número de mutantes para os programas em teste através da seleção randômica de 10% deles ou utilizando-se conjuntos bastante restritos de operadores de mutação, obtiveram-se conjuntos de teste que são quase TM-adequados, ou seja, que obtêm um escore de mutação próximo de 100% quando são avaliados utilizando-se o conjunto completo de mutantes. Esse é um resultado importante pois o custo do teste de mutação não pode ser medido apenas pelo número de testes requeridos – que também diminuiu usando os critérios alternativos. Deve-se contabilizar também o número de mutantes a serem executados e analisados. Resultado semelhante foi obtido por Offut et al. [13], porém utilizando uma outra forma de mutação alternativa. Foram utilizados critérios N-seletivos que, como já citado, não utilizam os N operadores de mutação que tendem a gerar mais mutantes. Obteve-se, como resultado, que conjuntos de teste 6-seletivos obtiveram em média 99,71% de adequação em relação à mutação não seletiva (utilizando-se todos os operadores de mutação). O número de mutantes foi reduzido, em média, 60,56%.

Além desses – que são os trabalhos mais relevantes no contexto desta tese – diversos outros experimentos empíricos têm sido realizados com o objetivo de avaliar critérios de teste. Dentre eles, podem ser citados os trabalhos de Offut e Lee [15], Offut et al. [13], Weyuker et al. [61], Maldonado [4], Girgis e Woodward [62].

2.4 Ferramentas de Teste

Um ponto fundamental no uso de critérios de teste é a utilização de ferramentas que automatizem sua aplicação. Sem a utilização de ferramentas que suportem a aplicação de critérios, a atividade de teste tende a ser – além de extremamente trabalhosa – propensa a erros. Isso é verdade, principalmente se considerarmos a crescente complexidade dos sistemas de software.

Além de melhora na produtividade e qualidade do teste, ferramentas podem ainda ser úteis como suporte para os futuros testes de regressão. A informação produzida por tais ferramentas podem ser utilizadas para a revalidação do software a cada manutenção. Tal informação pode auxiliar a verificar se a funcionalidade foi alterada ou não: a reduzir o custo de condução do teste de regressão; e a comparar os resultados obtidos no teste de regressão e no teste original.

Diversas ferramentas têm sido desenvolvidas, desde a década de 70, para apoiar a aplicação dos diversos critérios propostos. Em seguida, são brevemente descritas algumas dessas ferramentas.

A ferramenta Asset (A System to Select and Evaluate Tests) apóia a aplicação de critérios baseados em fluxo de controle e fluxo de dados de Rapps e Weyuker para programas Pascal. Foi desenvolvida na Universidade de Nova York em 1985 [63, 64].

Também a ferramenta Atac (Automatic Test Analysis for C) apóia os critérios propostos por Rapps e Weyuker, porém para programas escritos na linguagem C. Ela foi desenvolvida por Horgan e London [28] na Bell Communications Research. A ferramenta permite ao testador verificar a adequação de um conjunto de teste, visualizar código não exercitado, além de auxiliar na geração de casos de teste para cobrir as associações requeridas e na redução do conjunto de teste através da eliminação de casos de teste redundantes.

A ferramenta POKE-TOOL apóia a aplicação dos critérios de fluxo de dados Potenciais Usos, propostos por Maldonado [4], bem como de critérios de fluxo de controle, como todos-nós e todos-arcos. Essa é uma ferramenta multilinguagem, atualmente configurada para C [24], Cobol [26] e Fortran [25]. Foi desenvolvida na Faculdade de Engenharia Elétrica da Universidade de Campinas. Basicamente, dado um programa a ser testado e um conjunto de teste a ser avaliado, a ferramenta determina, para cada critério selecionado pelo testador, um conjunto de associações/caminhos requeridos e avalia os casos de teste em relação a esses requisitos. Auxilia também na elaboração de novos casos de teste, indicando quais

requisitos foram e quais não foram cobertos. Além disso, a ferramenta possui facilidades para a construção e visualização do grafo de programa das unidades do programa sendo testado [65].

Para o teste de mutação, duas ferramentas devem ser destacadas: Mothra [27] e Proteum [29, 30, 31, 66]. A primeira foi desenvolvida na Universidade Purdue e no Instituto de Tecnologia da Georgia. Ela possui 22 operadores de mutação para a linguagem FORTRAN. Além de facilidades para geração, execução e análise de mutantes, a ferramenta conta também com recursos para geração automática de casos de teste [49]. Essa geração é baseada na determinação de certas condições que os casos de teste devem possuir e que podem (não garantidamente) fazer com que os mutantes sejam mortos. Essas condições são representadas através de restrições que são associadas aos mutantes. Assim, casos de teste que satisfaçam à condição associada a um mutante têm grandes chances de distinguir esse mutante. A Mothra utiliza a abordagem interpretativa para geração dos mutantes, isso é, o programa original é transformado em uma linguagem intermediária que é então interpretada pela ferramenta. Cada mutação é aplicada nessa representação intermediária criando assim os mutantes que podem então ser “executados” pela ferramenta. A vantagem dessa abordagem é que a criação dos mutantes é feita facilmente, não sendo necessária a criação de um mutante-fonte que é compilado para que esse possa então ser executado. Por outro lado, a execução dos mutantes tende a ser mais lenta de maneira interpretada do que seria criando-se um mutante executável. Além disso, existe o fato de que muitas vezes o interpretador não consegue reproduzir com fidelidade o ambiente para o qual o programa foi desenvolvido, fazendo com que o programa se comporte de maneira incorreta, não por possuir qualquer defeito, mas por problemas de ambiente.

A ferramenta Proteum (Program Testing Using Mutants) apóia a aplicação do teste de mutação para programas C. Foi desenvolvida no Instituto de Ciências Matemáticas da Universidade de São Paulo, Campus de São Carlos. A atividade de teste no ambiente Proteum é conduzida através de sessões de teste. Uma sessão de teste é caracterizada por uma base de dados que contém informação sobre um conjunto de casos de teste e um conjunto de mutantes. A ferramenta é composta por alguns programas (relacionados na Tabela 2) que atuam sobre essa base de dados para realizar as operações inerentes ao teste de mutação. Esses programas são divididos em dois grupos. Os programas básicos são programas de mais baixo nível e atuam diretamente na base de dados que compõe a sessão. Os programas utilitários usam os programas básicos para realizar alguma operação específica sobre a sessão como um todo.

Tabela 2: Programas que compõem a ferramenta Proteum

Programas Básicos	
li	transforma um programa C em uma representação intermediária chamada LI
li2nli	cria os grafos de programa e adiciona informação sobre nós dos grafos ao programa no formato LI
pctest	cria e manipula arquivos de teste, que descrevem as características gerais da sessão de teste
tcas	cria e manipula a base de dados de casos de teste
muta	cria e manipula a base de dados de mutantes
exemuta	constrói e executa os mutantes a partir de descritores de mutação; também usado para selecionar/de-selecionar grupos de mutantes
opmuta	aplica operadores de mutação ao programa original, criando descritores de mutação
report	cria relatórios sobre o andamento da sessão de teste
Programas Utilitários	
test-new	cria uma nova sessão de teste
tcas-add	inclui, interativamente, um novo caso de teste na base de dados
muta-gen	gera descritores de mutação e os inclui na base de dados
muta-view	programa interativo para visualização e análise dos mutantes

As operações disponíveis na ferramenta podem ser executadas pelo testador de duas maneiras. A primeira é chamando diretamente os programas na linha de comando de uma shell ou dentro de um script. A segunda é interativamente através de uma interface gráfica que cuida de chamar os programas que compõem a ferramenta. A primeira, indicada para “testadores experientes” tem a vantagem de não exigir constante interação do testador que pode conduzir uma sessão de teste de modo batch. A segunda, indicada para “novatos”, não requer o esforço adicional de preparação de scripts para execução de teste, sendo assim um ambiente mais propício para o testador pouco familiar com a aplicação do critério.

Em ambas as formas de interface, as operações típicas numa sessão de teste utilizando-se a Proteum são:

- criação de uma sessão de teste
- manipulação de casos de teste
 - inclusão interativa de casos de teste
 - inclusão batch de casos de teste armazenados em arquivos
 - exclusão lógica ou física de casos de teste
- geração de mutantes
- seleção de subconjuntos de mutantes

Tabela 2: Programas que compõem a ferramenta Proteum

Programas Básicos	
li	transforma um programa C em uma representação intermediária chamada LI
li2nli	cria os grafos de programa e adiciona informação sobre nós dos grafos ao programa no formato LI
pctest	cria e manipula arquivos de teste, que descrevem as características gerais da sessão de teste
tcas	cria e manipula a base de dados de casos de teste
muta	cria e manipula a base de dados de mutantes
exmuta	constrói e executa os mutantes a partir de descritores de mutação; também usado para selecionar/de-selecionar grupos de mutantes
opmuta	aplica operadores de mutação ao programa original, criando descritores de mutação
report	cria relatórios sobre o andamento da sessão de teste
Programas Utilitários	
test-new	cria uma nova sessão de teste
tcas-add	inclui, interativamente, um novo caso de teste na base de dados
muta-gen	gera descritores de mutação e os inclui na base de dados
muta-view	programa interativo para visualização e análise dos mutantes

As operações disponíveis na ferramenta podem ser executadas pelo testador de duas maneiras. A primeira é chamando diretamente os programas na linha de comando de uma shell ou dentro de um script. A segunda é interativamente através de uma interface gráfica que cuida de chamar os programas que compõem a ferramenta. A primeira, indicada para “testadores experientes” tem a vantagem de não exigir constante interação do testador que pode conduzir uma sessão de teste de modo batch. A segunda, indicada para “novatos”, não requer o esforço adicional de preparação de scripts para execução de teste, sendo assim um ambiente mais propício para o testador pouco familiar com a aplicação do critério.

Em ambas as formas de interface, as operações típicas numa sessão de teste utilizando-se a Proteum são:

- criação de uma sessão de teste
- manipulação de casos de teste
 - inclusão interativa de casos de teste
 - inclusão batch de casos de teste armazenados em arquivos
 - exclusão lógica ou física de casos de teste
- geração de mutantes
- seleção de subconjuntos de mutantes

- execução de mutantes
- produção de relatórios

Ao contrário da ferramenta Mothra, a Proteum utiliza a abordagem compilada para execução dos mutantes. Quando da execução dos mutantes, a ferramenta cria um programa fonte com o código do mutante, compila-o e então executa-o. Dessa maneira, a execução dos mutantes tende a ser mais rápida que na abordagem interpretada. Por outro lado, esse processo de compilar os mutantes tende a ser demorado, fazendo com que o tempo total para execução dos mutantes seja alto. Para contornar esse problema, um único programa fonte é criado e compilado para diversos mutantes, exigindo assim apenas uma compilação para vários mutantes. Além disso, a estratégia utilizada para gerar e armazenar os mutantes prioriza a velocidade na criação do mutante fonte. Outros detalhes sobre a ferramenta podem ser encontrados nos trabalhos de Delamaro et al. [31, 66].

2.5 Fases da Atividade de Teste

A atividade de teste é, segundo descreve a literatura, dividida em diferentes fases. Basicamente, podem ser destacadas as fases de: 1) teste de unidade; 2) teste de integração; e 3) teste de sistema [1].

O teste de unidade é voltado para a verificação de cada parte que compõe o software, de maneira independente. O objetivo é obter evidências de que cada uma delas desempenha as funções que delas são esperadas, conforme descreve o projeto do software. Após testadas separadamente, as unidades devem ser integradas, ou seja, colocadas para interagirem, de modo a formar a estrutura do software. Embora testadas isoladamente, quando integradas as unidades podem apresentar problemas, principalmente de interface, ou seja, problemas na maneira como elas interagem. O teste de integração é uma técnica sistemática para se construir a estrutura do programa e revelar erros associados com as interfaces das unidades. Teste de sistema corresponde a uma série final de testes realizada quando a estrutura do software está montada e este funciona como um todo [1].

No escopo desta tese concentra-se atenção nas fases de unidade e – principalmente – integração do software. Em particular, critérios de teste para cada uma dessas fases são de interesse para o desenvolvimento deste trabalho. As sessões seguintes discutem alguns trabalhos encontrados na literatura e que tratam do teste de unidade e de integração.

2.5.1 Teste de Unidade

Segundo o padrão IEEE 610.12-1990, uma unidade é um componente de software que não pode ser subdividido. Um módulo é uma parte do programa logicamente separável. O uso desses dois termos muitas vezes é feito de modo intercambiável ou um como sub-elemento do outro, não havendo uma padronização de fato. Nesta tese, unidade é a menor parte funcional de um programa, como uma sub-rotina ou uma função. Um módulo é composto por uma ou mais unidades que trabalham em conjunto para executar uma tarefa.

Usualmente, o teste de unidade é o primeiro passo no teste de um programa. Seu objetivo é obter um certo grau de confiança de que cada unidade realiza a função dela esperada. Uma unidade é, em geral, desenvolvida para ser parte de um programa maior, não constituindo um programa completo por si mesma. Assim, para que a unidade possa ser testada é necessário que se construa um programa que permita a execução e teste da unidade.

Para cada unidade **F** sendo testada, um **driver** e um ou vários **stubs** podem ser necessários. O driver é uma unidade que coordena o teste de **F**. Ele é responsável por pegar dados de teste fornecidos pelo testador, passar esses dados na forma de parâmetros para **F**, coletar os resultados relevantes produzidos por **F**, e apresentá-los para o testador.

Tome-se, por exemplo, uma função parte de um compilador, responsável pela análise léxica. Segundo sua especificação, ela deve ler uma palavra, e retornar um “token” correspondente à classe da palavra lida. Por exemplo, ela retorna 1 para números, 2 para identificadores, 3 para palavras reservadas e assim por diante. A Figura 7 apresenta um pedaço de tal função. Para testar o analisador léxico, sem que o compilador todo esteja pronto, um driver é necessário. A Figura 8 mostra as declarações e a função necessárias para implementar o driver. Ele lê uma palavra, armazena-a na variável “word”, usada como entrada para a função “analex”, chama o analisador léxico e imprime o resultado retornado.

Um stub é uma unidade que substitui, na hora do teste, uma unidade usada (chamada) por **F**. Na maior parte dos casos, um stub é uma unidade que simula o comportamento da unidade chamada por **F** com o mínimo de computação ou manipulação de dados. No exemplo do compilador, o analisador léxico usa a função “readch” que retorna o próximo caracter da cadeia de entrada e a função “unreadch” que devolve um caracter a essa mesma cadeia de entrada. Esse comportamento é simulado pelas funções apresentadas na Figura 9. O stub da função “readch” simplesmente retorna o próximo caracter da variável “word”, apontado

```

1  anallex()
2  {
3      int      ch;
4
5      do {
6          ch = readch();
7      } while (isspace(ch));
8
9      i = 0;
10     simbol[0] = '\0';
11     value = 0;
12
13     if ( isdigit(ch))    /* analyze  an integer number */
14     {
15         while (isdigit(ch))
16         {
17             if (i > 10)
18             {
19                 simbol[0] = '\0';
20                 token = -1;
21                 possibly_error = ANALEX_ERROR + 1;
22                 return;
23             }
24
25             simbol[i++] = ch;
26             value = value * 10 + (ch - '0');
27             readch();
28         }
29         unreadch(ch);
30         simbol[i] = '\0';
31         token = 1;
32         return;
33     }
34     else
35     if (isalpha(ch))
36     {
37         .
38         .
39         .
40 }

```

Figura 7: Corpo do analisador léxico a ser testado

pela variável “pch”, e incrementa “pch”. O stub de “unreadch” decrementa a variável “pch”. O uso conjunto do driver e desses stubs torna possível o teste do analisador léxico de maneira independente, isolada do resto do compilador e interativamente.

O teste funcional pode ser aplicado no teste de unidade, bem como no teste de integração. A definição dos requisitos de teste é baseada na especificação do programa, quer seja ele um programa simples – tipo driver-unidade-stubs –


```

1 #define          ANALEX_ERROR          100
2
3 int      token;
4
5 int      possibly_error;
6 long     value;
7
8
9 main()      /* This is the driver function */
10 {
11
12     do
13     {
14         gets(buf);      /* read a new word */
15         pch = 0;        /* reset character pointer */
16
17         analex();        /* analyze the word */
18         print("\n%s", simbol); /* print the word read */
19
20         switch(token)    /* print the class to which the word belongs */
21         {
22             case 1:      /* it's a integer number */
23                 printf("Integer number - %ld\n", value);
24                 break;
25             case 2:      /* It's an identifier */
26                 printf("Identifier\n");
27                 break;
28             case 3:
29                 .
30                 .
31                 .
32             default:     /* an unknown token was returned */
33                 printf("Invalid word\n");
34                 break;
35         }
36     } while (1);
37 }

```

Figura 8: Driver do analisador léxico

ou um programa composto por várias unidades integradas. Por outro lado, a maioria dos critérios estruturais e baseados em defeitos citados anteriormente são critérios **intraprocedimentais**, ou seja, os elementos requeridos por eles estão restritos a uma única unidade. Por exemplo, os critérios de Rapps e Weyuker que requerem que associações do tipo definição-uso sejam exercitadas, claramente limitam os requisitos de teste a determinados subcaminhos dentro de uma unidade. O mesmo acontece para outros critérios “baseados em caminhos” como o critério de McCabe.

Analogamente, critérios como o teste de mutação e de cobertura de condições

```

1 int readch()
2 {
3     int i;
4     i = (int) word[pch++];
5     return i;
6 }
7
8 void unreadch()
9 {
10     pch--;
11 }

```

Figura 9: Stubs do analisador léxico

exigem que casos de teste que possuam certas características especiais sejam utilizados. Por exemplo, dados os trechos do programa e do mutante abaixo

<pre> while (a >= 0) { S1; a--; } </pre>	<pre> while (a > 0) { S1; a--; } </pre>
---	--

exige-se um caso de teste que irá exercitar os limites do laço em questão mostrando que a sua execução uma vez a menos causa diferenças na saída do programa. No caso geral, tal mutante apenas exige um caso de teste que exercita uma estrutura interna da unidade, no caso o comando *while*.

Dessa maneira, quando se pretende testar uma unidade isoladamente, visando a encontrar erros no seu comportamento, tanto critérios funcionais como estruturais ou baseados em defeitos podem ser utilizados, fornecendo um nível aceitável de “cobertura”. Na verdade, como aponta a literatura [1, 67], esses critérios devem ser utilizados em conjunto, dado o caráter complementar que os caracteriza, em relação ao tipo de erros que cada um revela com maior probabilidade.

2.5.2 Teste de Integração

A ênfase no teste de integração está nas interfaces das unidades e suas interações. Conjuntos de teste devem ser construídos com o objetivo explícito de exercitar essas interações e não a funcionalidade das unidades. Segundo estudo conduzido por Basili e Perricone [68], para determinados casos, até 40% dos erros de software têm sua origem em problemas de integração e interface e são produto de má interpretação da especificação da unidade [69].

Beizer [67] apresenta uma lista de possíveis erros que podem ser caracterizados como erros de integração. São eles:

- erro no projeto de protocolos
- erro no formato de entradas ou saídas
- proteção contra dados corrompidos
- chamada a subprograma incorreto
- parâmetros de chamada de subprograma incorretos
- ponto de entrada ou saída mal interpretado
- valores de parâmetros incorretos

Também Leung e White [70] apresentam um modelo do que consideram erros de integração. Além de identificar tipos de erros, os autores procuram indicar quais deles podem ser detectados no teste de unidade ou somente na fase de integração. Os erros de integração segundo o modelo, são divididos em três tipos.

O primeiro tipo é dos **erros de interpretação**. Eles ocorrem quando a funcionalidade implementada por uma unidade difere da especificação – ou da interpretação dada à especificação pelo desenvolvedor. Esse tipo de erro é dividido, segundo os autores, em três classes, supondo que a unidade A chama a unidade B:

Funcionalidade incorreta: a funcionalidade implementada na unidade B pode não ser a requerida pela especificação de A. O desenvolvedor pode incorretamente assumir que B desempenha as funções dela esperadas, tendo porém a incorreta interpretação do domínio de B e de como A interage com esse domínio. Esse erro deve ser detectado no teste de unidade, embora possa também ser foco de atenção na fase de integração.

Funcionalidade extra: a unidade B contém funcionalidade extra, não requerida pela unidade A. O desenvolvedor é ciente desse fato, porém assume que A não utiliza essa funcionalidade. O problema ocorre quando algum valor passado de A para B faz com que essa funcionalidade seja inesperadamente executada e resultados inesperados sejam produzidos. Esse problema provavelmente não será revelado no teste de unidade.

Funcionalidade ausente: A passa valores para B fora do domínio de B. Novamente, teste de unidade não garante que esse problema seja revelado.

O segundo tipo é dos erros de **chamadas incorretas**, onde o desenvolvedor posiciona uma chamada a uma unidade na posição incorreta dentro da unidade que chama. Tal erro pode se manifestar como três possíveis defeitos:

Chamada extra: a chamada encontra-se num caminho que não deveria tê-la.

Chamada incorretamente posicionada: a chamada encontra-se no local incorreto, num caminho que deveria tê-la.

Chamada ausente: um caminho que deveria ter uma chamada não a tem.

Embora esses erros devessem ser revelados no teste de unidade, a chamada incorretamente posicionada e a chamada ausente podem ser difíceis de detectar e especial atenção deve ser dada a esse tipo de erro no teste de integração.

O terceiro tipo é dos **erros de interface** que ocorrem sempre que o padrão de interação entre duas unidades é violado. Por exemplo, ordem incorreta de parâmetros, tipo de parâmetro incorreto, formato incorreto, regras de passagem de parâmetro que não combinam (passagem por valor/referência), entre outros. Em particular, um sério problema ocorre quando o domínio dos parâmetros formais difere do domínio dos parâmetros reais. A maioria dos erros relacionados por Beizer [67] são desse tipo e claramente não são adequadamente tratados no teste de unidade.

Várias estratégias têm sido propostas para a integração de um software. Estratégias não incrementais, também conhecidas como **Big-bang**, combinam todos os módulos do software para então aplicar o teste. Já estratégias incrementais constroem o software por partes. Cada parte é testada, integrada com outras partes, e novamente testadas, até que o sistema completo seja construído.

As principais estratégias incrementais são a **Top-Down**, a **Bottom-Up** e **Sandwich**. Na primeira, a integração inicia-se com o módulo mais alto na hierarquia de controle do software e módulos vão sendo adicionados “abaixo” da parte integrada, até que sejam integrados aqueles modos que compõem o mais baixo nível na estrutura do software. Essa estratégia exige a utilização de stubs que devem substituir temporariamente os módulos de mais baixo nível enquanto estes não são integrados. A vantagem dessa estratégia é que pontos principais de decisão ou controle, que em geral encontram-se nos módulos de mais alto nível, são testados antes, fazendo com que possíveis problemas sejam revelados mais cedo. A estratégia Bottom-Up, ao contrário, inicia a integração pelos módulos mais baixos, caminhando em direção ao módulo mais alto na estrutura do programa.

Nesse caso, drivers são necessários para coordenar o teste de cada subsistema construído. A estratégia Sandwich combina as duas anteriores. O programa é integrado de baixo para cima e de cima para baixo e eventualmente a integração “se encontra” em algum ponto no meio da estrutura [1].

Pressman [1] aponta que deve-se preferir as estratégias incrementais às não-incrementais. Segundo ele, nas estratégias incrementais os erros são mais fáceis de isolar e corrigir, as interfaces são testadas de maneira mais completa e uma abordagem sistemática pode ser aplicada. Já as estratégias não-incrementais fazem com que uma série de erros sejam encontrados de uma vez, fazendo com que a sua identificação e correção sejam dificultadas. Por outro lado, um experimento realizado por Solheim e Rowland [71] mostra que em relação à capacidade em revelar defeitos, nenhuma das estratégias é claramente superior às outras.

Um dos problemas do teste de integração é a ausência de critérios para avaliação e seleção de conjuntos de teste. Como citado anteriormente, a maioria dos critérios exige a execução de estruturas limitadas ao corpo de uma única unidade, não contemplando estruturas relacionadas à interação entre unidades. Por exemplo, para que o teste baseado no fluxo de dados possa ser aplicado no nível interprocedimental, segundo Harrold e Soffa [32], deve-se testar subcaminhos definidos por dependências de dados que ultrapassam os limites de uma única unidade, estabelecidas através de variáveis globais e passagem de parâmetros.

Alguns trabalhos têm procurado estabelecer critérios que caminhem para o nível interprocedimental, de maneira a testar com maior eficiência a interação entre unidades. Um trabalho pioneiro nesse sentido é o de Haley e Zweben [40]. Os autores classificam os erros de integração em dois tipos: erros de integração de domínio e erros de integração computacionais. O primeiro tipo é caracterizado por defeitos que causam um erro no caminho que é executado pelo programa, ou seja, um defeito que faz com que um caminho Y' seja executado quando o correto seria a execução do caminho Y . No erro de integração computacional, o caminho executado é o correto porém algum valor é calculado incorretamente. Essas definições são as mesmas feitas por Howden [72] mas incluem o fato de que tais defeitos fazem com que valores incorretos sejam trocados entre unidades que se relacionam. A proposta de Haley e Zweben é que sejam “re-testados” (supondo-se que cada unidade foi testada separadamente) no teste de integração, caminhos de uma unidade relacionados com a sua interface. O número de tais caminhos é proporcional ao número de parâmetros de entrada para que sejam revelados erros de domínio e proporcional ao número de parâmetros de entrada ou saída para erros computacionais.

Linnenkurgel e Müllerbug definem critérios baseados no fluxo de controle e no fluxo de dados interprocedimentais de um programa. O primeiro grupo de critérios – baseado no fluxo de controle – proposto pelas autoras utiliza um Grafo de Chamadas para representar o programa e identificar as estruturas a serem exercitadas. Um Grafo de Chamada é um multigrafo direcionado onde cada nó representa uma unidade e cada aresta representa a chamada de uma unidade a outra. Como uma unidade A pode chamar várias vezes a unidade B , mais do que uma aresta pode ligar dois nós. Alguns dos critérios desse tipo definidos por Linnenkurgel e Müllerbug são:

Todos-módulos: requer que cada unidade (as autoras utilizam o termo “módulo”) seja chamada pelo menos uma vez.

Todas-relações: requer que pelo menos uma aresta entre dois módulos seja executada pelo menos uma vez.

Todas-relações-múltiplas: requer que todas as arestas entre dois módulos sejam executadas pelo menos uma vez.

Os outros critérios definidos por Linnenkurgel e Müllerbug definem associações definição-uso interprocedimentais a serem cobertas pelos casos de teste. Por exemplo, dada uma “variável de comunicação” v de uma função $\mathbf{F}$, ou seja, um parâmetro formal ou uma variável global, alguns dos critérios definidos são:

INT-todas-defs: requer, para cada chamada, a execução de um caminho livre de definição com respeito a v , de cada definição de v antes da chamada de $\mathbf{F}$ até algum ponto onde v é usada em $\mathbf{F}$ e de cada definição de v dentro de $\mathbf{F}$ até algum uso após o retorno de $\mathbf{F}$.

INT-todos-usos: requer, para cada chamada, a execução de um caminho livre de definição com respeito a v , de cada definição de v antes da chamada de $\mathbf{F}$ até todos os pontos onde v é usada em $\mathbf{F}$ e de cada definição de v dentro de $\mathbf{F}$ até todos os usos após o retorno de $\mathbf{F}$.

Outro importante trabalho nessa linha foi apresentado por Harrold e Soffa [32] que também propuseram uma forma de estender os critérios de fluxo de dados para o nível interprocedimental através da definição de associações definição-uso que se estabelecem além do limite de unidades, de maneira bastante semelhante ao que foi proposto por Linnenkurgel e Müllerbug. Harrold e Soffa, porém, foram mais adiante e propuseram meios que permitem a automatização do critério.

apresentando soluções para alguns problemas de implementação para os critérios como, por exemplo, calcular as associações interprocedimentais. Em seguida, são sumarizadas as principais abordagens utilizadas pelas autoras para esse problema.

O primeiro passo para a aplicação do método proposto por Harrold e Soffa é a representação das unidades através de um grafo chamado de **Grafo de Fluxo Interprocedimental** que, juntamente com os grafos de fluxo de controle de cada unidade, sumariza as informações necessárias para a análise de fluxo de dados interprocedimental. A utilização desse grafo é exemplificada através do programa das Figuras 10 e 11.

```

1 program Main(input,output)
2 var S: array[1..10]of integer;
3     I, MAX, N: integer

4 procedure PairMax(var I, J, K: integer)
5 begin
6     if I>J then K:=I
7     else K:=J
8 end

9 procedure GetMax(F,L: integer; var MX: integer)
10 var M1, M2, MD: integer;
11 begin
12     if L<=F+1 then PairMax(S[F], S[L], MX)
13     else begin
14         MD:=(F+L) DIV 2;
15         GetMax(F, MD, M1);
16         GetMax(MD+1, L, M2);
17         PairMax(M1, M2, MX);
18     end;
19 end

20 begin
21     readln(N);
22     for I:=1 to N do readln(S[I]);
23     GetMax(1, N, MAX);
24     writeln(MAX);
25 end.
```

Figura 10: Programa *GetMax*

Os grafos de fluxo interprocedimentais abstraem informação sobre o fluxo de controle e de dados para cada procedimento. Os nós representam partes do código que são de interesse interprocedimental como pontos de entrada, pontos de saída, chamadas e retornos de procedimentos. As arestas (chamadas de arestas de alcance) representam informação sobre o fluxo de controle do procedimento como por exemplo, se o valor de um determinado parâmetro pode ou não permanecer

Figura 11: Grafo de fluxo dos procedimentos *GetMax* e *PairMax*

inalterado até a saída do procedimento. Na Figura 12 é apresentado o grafo da função *PairMax* e parte do grafo da função *GetMax*, correspondente às chamadas dos nós B5, B6 e B7 (linhas 15, 16 e 17). Círculos representam chamadas e retornos de procedimentos, círculos duplos representam pontos de entrada e pontos de saída do procedimento e as linhas pontilhadas representam as arestas de alcance. Assim, os nós 3, 5, 7, 9 e 11 são nós de chamadas. O nó 3 corresponde a M1 sendo passado como argumento na chamada recursiva a *GetMax* no bloco B5. O nó 5 corresponde a M2 sendo passado como argumento na chamada do bloco B6. Os nós 7, 9 e 11 representam M1, M2 e MX sendo passados como argumentos para *PairMax* no bloco B7. Os nós 4, 6, 8, 10 e 12 são os nós que representam o retorno dos nós 3, 5, 7, 9 e 11 respectivamente. Os pares {1,2}, {13,14}, {15,16} e {17,18} são os pontos de entrada e saída para cada um dos parâmetros formais dos procedimentos. Deve-se notar que apenas argumentos passados por referência são considerados na construção do grafo interprocedimental.

A aresta (1,11) indica que o valor de MX que entra no procedimento *GetMax* pode alcançar a chamada de *PairMax* sem ser alterado, e a aresta (12,2) indica que o valor MX retornado por essa mesma chamada pode chegar à saída do procedimento *GetMax*. As arestas (4,7) e (6,9) indicam que os valores de M1 e M2 retornados das chamadas a *GetMax* (blocos B5 e B6) podem alcançar a chamada

a PairMax (bloco B7) sem sofrer alteração. No procedimento PairMax, as arestas indicam que os valores dos parâmetros formais I e J que atingem a entrada do procedimento podem permanecer sem alteração até a saída do procedimento. O mesmo não acontece para o parâmetro K pois não existe um caminho completo em PairMax onde K não seja definido.

Figura 12: Passo inicial na construção do grafo de fluxo interprocedimental para os procedimentos GetMax e PairMax

Cada nó de entrada e cada nó de retorno α possui associado a si um conjunto chamado de **conjunto de usos**, que contém o conjunto de nós do grafo de fluxo (intraprocedimental) do procedimento que possuem usos que possam ser alcançados pelo parâmetro associado a α sem que este seja redefinido. Em PairMax, o parâmetro I tem um uso predicativo nas aresta (B10.B11) e (B10,B12) e um uso computacional em B11. De maneira análoga J tem usos em (B10,B11), (B10,B12) e B12. E em GetMax, no pedaço do grafo mostrado, Apenas MX tem um uso após retornar da chamada de PairMax, no nó B8.

Os subgrafos de cada procedimento são então combinados. São adicionadas arestas que representam o acoplamento de parâmetros reais e formais, formando o grafo interprocedimental. Na Figura 13 pode-se ver como fica o grafo GetMax-PairMax com a inclusão dessas arestas. As arestas (3,1), (5,1), (7,13), (9,15) e (11,17) associam nós de chamadas aos nós de entrada do procedimento chamado

e as arestas (2,4), (2,6), (14,8), (16,10) e (18,12) associam os nós de saída do procedimento chamado com os nós de retorno no procedimento que faz a chamada. Além dessas arestas, são também adicionadas arestas de inter-alcance que indicam que o valor de uma determinada variável usada como argumento de uma chamada pode ter seu valor preservado através da chamada. É o caso das arestas (7,8) e (9,10) que indicam que as variáveis M1 e M2, respectivamente, podem não ser alteradas na chamada de PairMax.

Figura 13: Grafo de fluxo interprocedimental para os procedimentos GetMax e PairMax com indicação dos acoplamentos de parâmetros formais e reais

Em seguida, são associados aos nós de chamadas de procedimento e de saídas de procedimento conjuntos que representam os usos de parâmetros formais e de parâmetros reais que podem ser alcançados através dos limites dos procedimentos. Primeiro são propagados os pontos de usos de parâmetros que podem ser alcançados através de uma chamada de procedimento. Depois são propagados os pontos de usos que podem ser alcançados através de pontos de saída. Como resultado obtém-se para cada nó de chamada e de saída de procedimento, um conjunto de usos alcançáveis. Na Figura 14, os conjuntos RU1 a RU4 são associados aos nós de chamada 3, 5, 7, 9 e 11 e aos nós de saída 18, 16, 14 e 2. Por exemplo, o conjunto RU3 é associado ao nó 18 e representa os pontos onde o valor do parâmetro K retornado por PairMax pode ser usado, ou em outras

palavras, os pontos alcançáveis por K, sem redefinição. Para calcular-se esse conjunto deve-se percorrer o grafo partindo do nó 18 até chegar-se a um nó sem uma aresta saindo dele. Deve-se unir todos os conjuntos U associados aos nós no caminho percorrido, construindo assim o conjunto RU. No caso do nó 18, por exemplo, o caminho percorrido seria: 18, 12, 2, 4, 7, 13, 14, 8, 6, 9, 15, 16 e 10. O conjunto RU3 seria $U3 \cup U1 \cup U2$.

Figura 14: Grafo de fluxo interprocedimental para os procedimentos GetMax e PairMax com os conjuntos de usos alcançáveis

Finalmente, são calculadas as associações definição-uso interprocedimentais, utilizando-se tanto informação local a cada procedimento como informação do grafo interprocedimental. Para cada ponto de definição de um parâmetro formal ou real são associados os usos interprocedimentais alcançáveis ¹. Nesse caso, tem-se a variável K definida em PairMax nos nós B11 e B12. Como ambas as definições alcançam o nó de saída, ambas terão como usos alcançáveis, o conjunto RU3, associado ao nó 18 do grafo interprocedimental. As associações definição-uso são sumarizadas na Tabela 3.

Além da definição de critérios de fluxo de dados, Harrold e Soffa apresentam também um algoritmo para avaliar se um determinado caminho executado

¹Note-se mais uma vez que somente os parametros definidos ou passados como referencia são considerados.

Tabela 3: Associações definição-uso interprocedimentais entre PairMax e GetMax

Variável	Definição	Uso
K no procedimento PairMax	B11	B8, B11, B12, (B10,B11), (B10,B12)
	B12	B8, B11, B12, (B10,B11), (B10,B12)

pelo programa cobre ou não uma associação, solucionando diversos problemas, como a troca de nomes de variáveis quando procedimentos são chamados ou quando retorna-se de uma chamada de procedimento. Talvez seja esta a maior contribuição desse trabalho pois a maior dificuldade em estabelecerem-se critérios interprocedimentais está em definirem-se meios para que eles possam ser implementados e, assim, utilizados na prática.

2.6 Considerações Finais

Neste capítulo discutiram-se os principais trabalhos relacionados com esta tese, destacando-se a importância de critérios de adequação de conjuntos de teste nas diversas fases dessa atividade.

Nota-se que no nível de teste de unidade diversos critérios têm sido propostos e estudados, com o objetivo de se determinar uma maneira sistemática e rigorosa para o projeto de conjuntos de teste que sejam eficazes em detectar erros no software a um baixo custo. Nessa fase da atividade de teste, tanto critérios funcionais quanto estruturais e baseados em defeitos têm sido largamente utilizados.

Além de critérios de teste para o nível de unidade, dispõe-se também de diversas ferramentas de teste que apoiam a aplicação dos critérios. Desta maneira, é possível automatizar sua aplicação, diminuindo a interferência humana e aumentando a qualidade e produtividade do teste. Além disso, ferramentas de teste contribuem também para a própria pesquisa nessa área, permitindo que estudos empíricos possam ser conduzidos empregando os critérios propostos.

Vários autores têm se ocupado de, além de definir critérios, avaliá-los e compará-los com outros critérios. Esses estudos se dão basicamente em duas frentes. A primeira compreende estudos que analisam características dos critérios como custo e efetividade de maneira puramente teórica. A segunda utiliza experimentos para avaliar essas características de maneira empírica. Todos esses estudos têm como objetivo determinar características importantes que podem auxiliar na

definição de qual critério usar ou estabelecer estratégias de teste que agreguem um ou mais desses critérios.

Por outro lado, no nível de integração nota-se uma ausência de critérios e ferramentas que auxiliem na avaliação e na seleção de casos de teste de modo a exercitar de forma efetiva a interação entre unidades. Alguns poucos trabalhos nessa direção têm sido conduzidos, mas percebe-se que muito ainda precisa ser feito.

Por isso, considera-se que a proposta desta tese, de definir um critério interprocedimental baseado em mutações que possa ser utilizado no teste de integração, de implementar uma ferramenta que dê suporte à aplicação desse critério, e de conduzir experimentos empíricos para avaliar a sua efetividade e o seu custo, vem a contribuir para suprir as necessidades nesta importante área dentro da Engenharia de Software.

Capítulo 3

Mutação de Interfaces

Neste capítulo é apresentada a proposta principal desta tese, ou seja, o critério de adequação de casos de teste interprocedimental baseado em mutações chamado Mutação de Interface.

Inicialmente, discute-se um modelo de falhas que caracterizam os erros de integração. Como esses erros manifestam-se nas conexões entre duas unidades, através de suas interfaces, construiu-se um critério que, através da introdução de defeitos simples, cria mutantes procurando reproduzir esses erros. Dessa maneira, o critério exige casos de teste que revelem esses erros e – assumindo que o efeito de acoplamento vale também para erros de integração – exercitem efetivamente as conexões entre as unidades do programa.

Apresenta-se também um conjunto de operadores de mutação de interfaces. Operadores de mutação são dependentes da linguagem de programação para a qual são projetados. Os operadores apresentados aqui foram projetados para a linguagem C mas podem ser facilmente adaptados para qualquer linguagem que possua características semelhantes em termos de interação entre unidades (chamadas de função e passagem de parâmetros).

Considerando esse conjunto de operadores, são feitas algumas análises teóricas sobre o critério. A complexidade de sua aplicação é avaliada e faz-se também uma comparação entre o critério e os critérios de fluxo de dados interprocedimentais de Harrold e Soffa [32].

Discute-se ainda a necessidade de definirem-se critérios alternativos para a

Mutação de Interface com o objetivo de permitir sua aplicação de forma flexível, reduzindo o número de mutantes e ajustando-a a restrições de custo.

3.1 Modelo de Falhas Causadas por Erros de Integração

É importante que se estabeleça os tipos de erros que se deseja revelar para que se possa determinar uma estratégia para selecionar casos de teste. Nesta sessão são determinados tipos de falhas que caracterizam erros de integração. Com isso, define-se posteriormente o critério Mutação de Interface cujo objetivo é revelar esse tipo de erro.

Haley e Zweben [40] definem erro de integração computacional e erro de integração de domínio. Dadas as unidades F e G , sendo que F chama G , um erro de integração computacional ocorre quando um defeito em G faz com que um valor incorreto seja calculado, levando F a produzir uma saída incorreta. Um erro de integração de domínio acontece quando um caminho incorreto é executado em G , levando F a produzir uma saída incorreta. Em ambos os casos existe algum valor incorreto sendo passado entre duas unidades, o que leva a um comportamento incorreto do programa, ou seja, uma falha.

Da mesma maneira, os erros apontados por Leung e White [70], 1) erros de interpretação; 2) chamadas incorretas; e 3) erros de interface, têm em comum o fato de que dadas duas unidades, pode-se identificar erros na interação entre elas, em geral manifestados através de suas interfaces.

Considerando-se esses aspectos, foram estabelecidos, como parte deste trabalho de teste, três tipos de falhas que caracterizam um erro de integração. Suponha-se o programa $\mathbf{P}$, o caso de teste t e as unidades F e G tal que F faz chamadas a G . Define-se $S_I(G)$ como o conjunto de valores passados de F para G e $S_O(G)$ como o conjunto de valores retornados de G para F . Ao executar-se $\mathbf{P}$ com o caso de teste t , identifica-se uma falha causada por um erro de integração das unidades F e G em uma das seguintes situações:

- Tipo 1: se na chamada de G , $S_I(G)$ contém algum valor incorreto que leva a uma saída incorreta (uma falha) antes do retorno de G .
- Tipo 2: se na chamada de G , $S_I(G)$ contém algum valor incorreto que faz com que $S_O(G)$ assuma algum valor incorreto levando a uma saída incorreta (uma falha) após o retorno de G .

- Tipo 3: se na chamada de G , todos os valores de $S_I(G)$ são os esperados, porém valores incorretos de $S_O(G)$ são produzidos em G , levando a uma saída incorreta (uma falha) após o retorno de G .

A Figura 15 esquematiza esses três tipos de falhas que caracterizam os erros de integração.

Figura 15: Modelo de falhas causadas por erros de integração: (a) Tipo 1; (b) Tipo 2; e (c) Tipo 3

Note-se que esta é uma definição bastante ampla. Ela simplesmente considera a existência de valores incorretos entrando em ou saindo de uma chamada de subprograma. Ela não determina tipo ou localização dos defeitos que causam esses erros. A definição exclui, por exemplo, o caso em que $S_I(G)$ tem os valores esperados mas um defeito dentro da unidade G produz uma saída incorreta antes do retorno de G . Nesse caso, nenhum erro é propagado através da conexão F - G e supõe-se que esse tipo de erro seja revelado no teste de unidade.

Além disso, segundo essa definição, mais de um erro de integração pode estar associado a – ou ser causado por – um defeito. Por exemplo, um programa com as unidades R , S e T , conforme mostra a Figura 16, onde todas as passagens de parâmetros são feitas por referência. Na unidade S , o comando $y = x * x$ foi incorretamente codificado. O cálculo correto seria $y = x + x$. Dessa forma, o parâmetro formal y e, conseqüentemente, a variável b recebem o valor incorreto, que influencia uma falha após retornar-se de S , caracterizando um erro do tipo 3 na conexão R - S . Além disso, o valor de b é passado para T onde provoca uma saída incorreta caracterizando um erro do tipo 1 na conexão R - T . Finalmente, o valor incorreto de b passado para a unidade T faz com que o valor da variável c (parâmetro formal j) seja retornado incorreto, gerando uma saída incorreta na unidade R , caracterizando um erro do tipo 2 na conexão R - T . Pode-se notar por esse exemplo que: 1) a localização do defeito não determina a conexão onde

identificam-se os erros de integração; 2) um único defeito pode determinar a existência de diversos erros; e 3) numa conexão podem ser identificados mais de um erro simultaneamente.

Figura 16: Exemplos de erros dos tipos 1, 2 e 3

Deve-se notar que a definição dos conjuntos $S_I(G)$ e $S_O(G)$ depende, em parte, da linguagem em que o programa é escrito. Por exemplo, para a linguagem C – e outras com características de acoplamento semelhantes – esses conjuntos podem ser definidos das seguintes maneiras:

- $S_I(G)$: o conjunto dos valores de entrada em uma chamada à função G é determinado por:
 - parâmetros formais de entrada da função G ;
 - variáveis globais usadas em G ;
- $S_O(G)$: o conjunto dos valores de saída em uma chamada à função G é determinado por:
 - parâmetros formais de saída da função G ;
 - variáveis globais definidas em F ;
 - valores retornados através de comandos *return*.

3.2 O Critério Mutação de Interface

A idéia básica do teste de mutação é avaliar um conjunto de teste baseado na sua habilidade em revelar defeitos simples, injetados no programa sendo testado. Tais defeitos procuram modelar casos comuns de erros. Um conjunto que atenda a esse requisito deve ser também eficaz em revelar outros erros, causados por defeitos mais complexos. É o que se chama de **efeito de acoplamento**, hipótese verificada experimentalmente por Offut [73].

Experimentos têm também mostrado que o teste de mutação é um critério bastante efetivo para avaliação e seleção de casos de teste [59]. Por outro lado, o custo de sua aplicação, em termos do número de mutantes a serem executados, é ainda uma barreira para a sua aplicação prática. Para o teste de unidade, algumas abordagens podem ser utilizadas para reduzir esse custo, por exemplo, aplicando-se critérios de mutação alternativos [20, 39]. Porém, quando se trata de sistemas maiores que algumas poucas unidades, somente essas abordagens podem não ser suficientes, inviabilizando a aplicação do teste de mutação.

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

No nível de integração é necessário testar não a unidade em si, mas principalmente as interações com as outras unidades do programa. Esse é o objetivo da Mutação de Interface. De acordo com o modelo de falhas apresentado, erros de integração são caracterizados por valores incorretos trocados através das conexões entre as unidades. Assim, a Mutação de Interface procura, através da injeção de defeitos simples, introduzir perturbações somente em objetos relacionados com a interação entre as unidades.

Em outras palavras, os mutantes a serem distinguidos pelos casos de teste no nível de integração modelam “erros simples de integração”. Mais precisamente, como define Offut [14], um mutante deve ser um programa cuja diferença semântica em relação ao programa em teste seja pequena. No caso da Mutação de Interface, essa diferença deve se manifestar nas conexões entre as unidades

do programa. Dessa forma, conjuntos adequados à Mutao de Interface (MI-adequados) devem ser – de acordo com o efeito de acoplamento – efetivos para revelar a presena de outros tipos de defeitos que se manifestem atravs de erros nas interaes entre as unidades.

Em linguagens convencionais como C, Pascal, Fortran, etc, unidades so conectadas atravs de chamadas de subprogramas (funes, procedimentos ou sub-rotinas). Numa chamada da unidade F unidade G , existem quatro maneiras, no mutuamente exclusivas, de trocarem-se dados entre as unidades:

- Dados so passados para G atravs de parmetros de entrada (passagem por parmetro).
- Dados so passados para G e/ou retornados para F atravs de parmetros de entrada-sida (passagem por referncia).
- Dados so passados para G e/ou retornados para F atravs de variveis globais.
- Dados so retornados para F atravs de comandos de retorno (como *return* em C).

Uma das vantagens da Mutao de Interface que ela tende, por princpio, a reduzir o nmero de mutantes a serem executados e analisados pois a aplicao dos operadores de mutao se restringe a esses pontos relacionados s conexes entre as unidades. Alm disso, o critrio pode ser aplicado de maneira incremental, propiciando a utilizao de estratgias incrementais de teste. possvel para o testador escolher uma conexo, gerar mutantes relacionados com essa conexo e selecionar casos de teste que distingam esses mutantes, exercitando assim aquela determinada conexo. O teste pode at ser dividido entre diversos testadores, cada um responsvel pelo teste de determinadas conexes.

Na Figura 17 v-se um exemplo de um programa, composto por algumas conexes, sobre o qual aplica-se a Mutao de Interface. O testador pode, por exemplo, escolher testar a conexo $A-B$ e nesse caso um operador de mutao OP aplicado aos pontos relacionadas apenas a essa conexo. Em outras palavras, o critrio sempre aplicado de uma forma “ponto-a-ponto”. limitando os requisitos de teste a uma determinada conexo.

Para reproduzir erros do tipo 1 e tipo 2, o operador OP , aplicado conexo $A-B$, deve fazer com que um objeto entrando na unidade B assuma um valor incorreto. Esse objeto usado em B produzindo uma sada incorreta ou fazendo

Figura 17: Estrutura de programa, utilizada para exemplificar algumas características da Mutação de Interface

com que um valor incorreto seja retornado à unidade *A*. A maneira mais simples de se obter esse resultado é fazendo com que *OP* seja aplicado nos parâmetros reais da chamada de *B*. Outra opção é aplicar o operador em cada ponto dentro da unidade *B* onde os parâmetros formais são usados. Tomando como exemplo o programa da Figura 18, nota-se que, aplicando-se a mutação ao parâmetro real *a* da chamada a *B*, aplicar-se-ia o operador *OP* apenas uma vez. Aplicando-se *OP* a cada uso do parâmetro formal *x*, aplicar-se-ia o operador três vezes, nas linhas 13, 15 e 17 do programa. Dessa maneira, a segunda abordagem tende a executar de forma mais completa os usos das variáveis que fazem parte da interface da unidade *B*, inclusive quando variáveis globais fazem parte dessa interface. Por outro lado, a aplicação dos operadores de mutação dentro da unidade chamada tende a criar um número mais alto de mutantes e, possivelmente, exigir um número mais alto de casos de teste para satisfazer o critério.

Erros do tipo 3 podem ser produzidos introduzindo-se perturbações em valores retornados da unidade *B* para a unidade *A*. Isso pode ser feito através de mutações em cada ponto dentro de *B* onde um valor que vai ser devolvido à unidade *A* é definido. É o caso, por exemplo, dos comandos de retorno nas linha 13, 15, 17 e 19 do programa da Figura 18.

Assim, para testar uma conexão A-B, introduzindo defeitos simples que levem a erros dos três tipos acima, a Mutação de Interface aplica mutações em: 1) pontos onde a unidade *A* chama a unidade *B*, portanto antes da execução de *B*; e 2) pontos dentro de *B* relacionados com sua interface.

Um problema sério no teste de integração é o acoplamento de unidades através de variáveis globais. Embora o critério de Mutação de Interface seja aplicado

```

1  A()
2  {
3  int a, b;
4      b = B(a);
5      printf("%d", b);
6  }
7
8  int B(int x)
9  {
10     switch(x)
11     {
12         case 1:
13             return x*2;
14         case 2:
15             return x;
16         case 3:
17             return x+1;
18     }
19     return -1;
20 }

```

Figura 18: Programa utilizado para exemplificar a aplicação dos operadores de mutação no ponto de chamada e dentro da unidade chamada

de maneira “ponto-a-ponto” – já que ele considera somente uma conexão por vez para derivar os requisitos de teste – outros tipos de acoplamento podem ser exercitados, além daqueles determinados diretamente pelas unidades conectadas. Por exemplo, na Figura 17, suponha-se que a unidade E define o valor de uma variável global v . Ao aplicar-se a Mutação de Interface à conexão C - E , uma mutação $OP(v)$ é aplicada nos pontos de E onde v é definida. Para que tal mutante seja distinguido é necessário um caso de teste que faça com que um uso de v leve a uma saída incorreta. Tal uso pode estar, por exemplo, na unidade D , o que significa que uma mutação dentro de E pode, em muitos casos, exercitar um acoplamento indireto entre as unidades D e E . Por outro lado, se a unidade A também possui um uso de v , é possível que tal mutante seja distinguido através do uso de v ou em A ou em D , ou seja, não se garante que um acoplamento através de variável global seja sempre exercitado.

Um ponto importante para que se aplique Mutação de Interface é que os operadores de mutação devem contemplar a idéia de que uma conexão é testada e não uma unidade. Por exemplo, se deseja-se testar a conexão C - D , um mutante cuja mutação $OP(v)$ seja efetuada dentro da unidade D pode ser distinguido através de uma chamada de D feita em B . Nesse caso, a conexão desejada não está sendo exercitada. Assim, a aplicação de OP , quando feita dentro da unidade chamada

deve levar em consideração o ponto de onde foi feita a chamada. No exemplo, a mutação só pode ser “ativada” se a unidade D for chamada por C . Quando chamada por B , D deve se comportar exatamente como no programa original, fazendo com que a conexão $C-D$ tenha que ser exercitada para que o mutante possa ser distinguido. Para algumas linguagens – como C por exemplo – isso significa que a decisão de aplicar-se ou não a mutação tem que ser feita em tempo de execução do mutante. Para mutações feitas no ponto onde C chama D esse problema não existe pois, nesse caso, o ponto de mutação garante que a conexão desejada está sendo exercitada.

Segundo DeMillo e Offut [49], existem três condições que devem ser satisfeitas para que um caso de teste t distinga um mutante $\mathbf{M}$; 1) alcançabilidade: a execução de $\mathbf{M}$ – e consequentemente do programa original $\mathbf{P}$ – com t deve passar pelo ponto onde a mutação foi feita; 2) necessidade: os estados de $\mathbf{P}$ e do mutante $\mathbf{M}$ devem diferir imediatamente após alguma execução do comando onde a mutação foi aplicada; e 3) suficiência: essa diferença no estado interno de $\mathbf{P}$ e $\mathbf{M}$ deve ser propagada até um ponto de maneira que $\mathbf{M}$ produza uma saída diferente de $\mathbf{P}$, já que no teste de mutação (forte) apenas as saídas são consideradas para distinguir os mutantes. No contexto da Mutação de Interface, quando o operador é aplicado dentro da unidade que é chamada, pode-se afirmar que a condição de alcançabilidade foi alterada, ou seja, para distinguir um mutante $\mathbf{M}$ o ponto de mutação tem que ser alcançado por um caminho que inclua a conexão que se deseja testar. Mais precisamente, redefine-se a condição de alcançabilidade para esses casos como:

Condição de alcançabilidade para a Mutação de Interface: A execução de um mutante $\mathbf{M}$, gerado para a conexão $A-B$, com um caso de teste t deve fazer com que o ponto onde a mutação foi introduzida seja alcançado através de uma chamada de A para B .

Os operadores de Mutação de Interface apresentados na próxima seção definem mais precisamente o critério. Eles são aplicados nos pontos de chamadas das unidades bem como no corpo da unidade chamada, conforme caracterizado nesta seção.

3.3 Operadores de Mutação de Interface para C

Operadores de mutação definem alterações sintáticas a serem realizadas num programa $\mathbf{P}$, com base nas quais são criados os seus mutantes. Num nível de abs-

traç ao mais alto é possível estabelecerem-se operadores de mutação de maneira genérica e que modelem tipos de defeitos comuns para qualquer linguagem de programação, ou pelo menos, para linguagens de programação com características semelhantes.

Por outro lado, no nível de implementação é necessário que os operadores levem em conta particularidades de uma determinada linguagem para a qual foram projetados. Por isso, nesse nível, operadores de mutação são dependentes de uma linguagem de programação para a qual são projetados. Os operadores de Mutação de Interface apresentados nesta seção foram projetados especificamente para a linguagem C, tendo em mente as características de interação entre unidades – basicamente definidas por regras de chamada de função e passagem de parâmetros – dessa linguagem. Porém, nota-se que tal conjunto pode facilmente ser mapeado para outras linguagens procedimentais com características semelhantes às da linguagem C.

A definição de operadores de mutação é fundamental para a aplicação de critérios baseados em mutação. Em última análise, são esses operadores que caracterizam o critério, estabelecendo os requisitos de teste – os mutantes – a serem satisfeitos. Um conjunto muito restrito de operadores pode estabelecer requisitos fracos, levando a seleção de conjuntos de teste pobres. Além disso, deve-se evitar conjuntos de operadores de mutação que levem a um custo, em termos do número de mutantes produzidos, excessivamente alto. No nível de unidade, a “arte” de projetar operadores de mutação vem sendo praticada por muitos anos, podendo-se citar o caso dos operadores para a linguagem FORTRAN, implementados na ferramenta Mothra, que representam mais de uma década de evolução no projeto de operadores [27]. Ainda assim, trabalhos mostram que mesmo esse conjunto pode ser estendido, dependendo dos programas a serem testados e das características que se desejam dos casos de teste [74].

O conjunto de operadores de Mutação de Interface apresentado aqui busca ser o mais completo possível, no sentido de poder exercitar a maioria das interações entre duas unidades. Por outro lado, deve ser também minimal, no sentido de que as mutações sejam restritas àquelas essenciais, ou seja, aplicadas apenas em pontos relacionados com a interação entre as unidades. Além disso, os operadores devem ser adequáveis a restrições de custo. É importante reconhecer que cada programa tem diferentes características e pode requerer teste com diferentes requisitos. Dessa forma, é necessário permitir ao testador ajustar, em termos de custo e requisitos de teste, o conjunto de mutantes a ser utilizado na aplicação do critério. Na Seção 3.5 são apresentadas algumas formas de parametrizar a

aplicação da Mutação de Interface.

Para o teste de unidade, operadores de mutação buscam modelar os defeitos simples mais comuns para uma determinada linguagem. Por exemplo, trocando-se um operador relacional por outros operadores relacionais força-se a escolha de casos de teste nos limites das condições de decisão do programa. Existem também os operadores chamados “instrumentados” [50]. Eles não modelam defeitos comuns mas procuram assegurar aos casos de teste características mínimas desejadas. Tome-se como exemplo um operador de mutação que troca cada comando da função, um de cada vez, pela chamada de uma função *TRAP()*, cuja execução faz com que o mutante seja automaticamente distinguido. Casos de teste que distingam todos mutantes instrumentados dessa forma fazem também com que cada comando da função original seja executado pelo menos uma vez.

Operadores convencionais e de Mutação de Interface possuem semelhanças e diferenças. A idéia por trás de ambos é a mesma, ou seja, produzir pequenas discrepâncias entre o estado do programa original e do programa mutante. Ou, como descrevem Offut e Hayes [14], produzir mutantes que sejam semanticamente parecidos com o programa em teste, através da introdução de pequenas diferenças sintáticas.

Por outro lado, operadores de Mutação de Interface são relacionados a uma conexão entre duas unidades e cada mutante é relacionado com uma chamada de função. Por exemplo, num programa que possua uma função f que faça duas chamadas a uma função g , como na Figura 19, a cada chamada de g corresponde um conjunto de mutantes que só podem ser distinguidos através da execução da chamada correspondente. Se chamarmos g' à função criada ao aplicar-se um operador OP em um ponto no interior de g , deve-se considerar dois mutantes distintos relacionados com o teste da conexão $f-g$. O primeiro é aquele onde a primeira chamada de g é substituída pela chamada a g' e o segundo é aquele onde a segunda chamada é substituída, como exemplificado na Figura 20.

Por isso, como citado anteriormente, uma mutação aplicada dentro da função chamada só deve ser efetivamente aplicada se a função foi chamada do ponto cuja conexão deseja-se testar. No exemplo da Figura 20. no primeiro mutante, que foi gerado para a primeira chamada a g , deve-se substituir g pela chamada a g' mas a segunda chamada deve permanecer inalterada, ou seja, nessa segunda chamada a função g deve comportar-se exatamente como no programa original, de modo que M_1 só possa ser distinguido através da primeira chamada. Somente dessa maneira pode-se exercitar de maneira mais completa a conexão, que na verdade é composta por mais de uma chamada de função.

Figura 19: Indicação dos conjuntos de mutantes requeridos para o teste da conexão $f-g$

Figura 20: Mutantes associados a diferentes chamadas da conexão $f-g$

Um ponto essencial ao se pensar no teste de mutação, e em particular na Mutação de Interface, é o custo do teste, em termos de mutantes a serem criados. É preciso que o número de mutantes seja mantido baixo para que o teste possa ser executado num período de tempo razoável. Os operadores descritos aqui buscam esse objetivo através da restrição dos pontos do programa onde são aplicados. Somente pontos relacionados com as conexões sendo testadas são sujeitos à aplicação dos operadores.

Dois grupos de operadores foram definidos e são apresentados em seguida. O primeiro é composto por operadores que, ao testar-se a conexão $f-g$, são aplicados dentro da função g . O segundo grupo é de operadores aplicados nos pontos onde a função f chama a função g .

Na definição dos operadores são utilizados os seguintes conjuntos, supondo-se que a Mutação de Interface está sendo aplicada na conexão entre as funções f e g :

$P(g)$: é o conjunto dos parâmetros formais de g . Esse conjunto inclui também derreferências a parâmetros do tipo ponteiro ou vetor. Por exemplo, se um parâmetro formal v é definido como “int *v”, v e “*v” pertencem a esse conjunto¹.

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

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

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

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

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

Em adição, mais um conjunto é definido. O conjunto R de “constantes requeridas” contém valores especiais, relevantes para alguns tipos primitivos de dados da linguagem C e operações associadas a esses tipos. A tabela 4 sumariza essas constantes. A Tabela 5 apresenta um resumo com todos os operadores de mutação definidos.

¹Como a linguagem C faz passagem de parâmetros sempre por valor e a passagem por referência é simulada passando-se como valor o endereço da variável, incluiu-se no conjunto $P(g)$ a derreferenciação dos parâmetros ponteiros ou vetores de forma que esse tipo de interface seja exercitada de maneira completa.

Tabela 4: Conjunto de constantes requeridas

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

3.3.1 Grupo I - Mutações Efetuadas na Função Chamada

Os operadores desse grupo aplicam mutações dentro do corpo da função chamada. Eles perturbam valores que entram e/ou saem da função como já descrito anteriormente.

Para esses operadores faz-se necessário um mecanismo para que seja identificado o ponto de chamada da função, possibilitando que a mutação seja ou não habilitada. Na implementação desses operadores na ferramenta *PROTEUM/IM* descrita com mais detalhes no Capítulo 4 – optou-se por incluir um mecanismo que imediatamente antes da chamada da função, no ponto desejado, habilita a mutação e desabilita-a no retorno.

Cada operador de mutação tem um certo escopo de aplicação. Por exemplo, alguns podem ser aplicados em referências a variáveis, outros em expressões e outros até em comandos inteiros. Assim, para uma dada conexão, cada operador tem um conjunto de pontos de aplicação próprio. Para todos os operadores deste grupo, quando uma variável é um ponto de mutação, considera-se que derivadas ou indexações àquela variável também o são. Assim, por exemplo, no programa

```
int g(int v[], int i)
{
  int i;
  *v = 0;
  v[i]++;
}
```

Tabela 5: Operadores de Mutação de Interface

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

ao dizer-se que a variável v é um ponto de mutação, subentende-se que “ $*v$ ” e “ $v[i]$ ” também são pontos de mutação e devem também ser alterados.

3.3.1.1 Operadores de Troca de Variáveis de Interface

A primeira e mais direta maneira de perturbar um valor que entra ou que sai da função chamada é diretamente trocar as variáveis de interface por outras variáveis e constantes. Os operadores desse tipo trocam ocorrências de elementos dos conjuntos P e G por diferentes grupos de variáveis e de constantes. São os operadores cujos nomes contêm o prefixo “DirVarRep”, mostrados na Tabela 5.

Por exemplo, para o programa da Figura 21, são indicados na Tabela 6 os

mutantes criados por cada um desses operadores quando aplicados à conexão $r-s$. Note-se que somente as trocas entre variáveis e constantes de tipos compatíveis são feitas. Além disso, alguns tipos de mutação não fazem sentido, como a substituição de uma variável no lado esquerdo de uma atribuição por uma constante.

```

int a, b[];

int r()
{
  int c, d;

  d = 10
  c = s(d, b);
}

int s(int i, int vet[])
{
  int j;

  a = 0;
  for (j = 0; j < i; j += 1)
    a += vet[j];
  return a;
}

```

$P(s)= \{i, \text{vet. } *vet\}$
 $G(s)= \{a\}$
 $L(s)= \{j\}$
 $E(s)= \{b\}$
 $C(s)= \{1, 0\}$

Figura 21: Programa exemplo para aplicação dos operadores de mutação

Tabela 6: Mutações relativas ao programa da Figura 21, para operadores de troca de variáveis de interface

Operador	Mutante	Operador	Mutante
DirVarRepPar	i = 0	DirVarRepGlob	j < a
	*vet = 0		a += a
DirVarRepConst	j < *vet	DirVarRepExt	a += b[j]
	a += i	DirVarRepLoc	j = 0
	i += vet[j]		j < j
	*vet += vet[j]		a += j
	return i		j += vet[j]
	return *vet		return j
	j < 0	DirVarRepReq	j < -1
	j < 1		j < MININT
	a += 0		j < MAXINT
	a += 1		a += -1
	return 0		a += MININT
	return 1		a += MAXINT
			return -1
			return MININT
			return MAXINT

3.3.1.2 Operadores de Troca de Variáveis Não de Interface

Outra maneira de modificar um valor que entra ou que sai da função é alterando (trocando) o valor de uma variável ou constante que, apesar de não ser parte da interface da função, pode influenciar algum valor da interface. Os operadores desse tipo são aqueles cujo nome contém o prefixo “IndVarRep” na Tabela 5.

Eles são aplicados a locais onde elementos dos conjuntos L e C são usados. Porém, como o objetivo é alterar uma variável de interface, ou pelo menos algum valor calculado a partir de uma variável de interface, e por uma questão de economia, todos operadores cujos pontos de aplicação sejam variáveis não de interface ou constantes são aplicados somente naqueles pontos que são relacionados com variáveis de interface. Mais precisamente, eles são aplicados a variáveis não de interface e constantes que:

1. sejam utilizadas numa expressão que também utiliza variáveis de interface
2. sejam utilizadas em comandos *return*

Assim, se x é uma variável de interface e i é não de interface, i será ponto de mutação nos seguintes casos:

- $\text{if } (x < i) \dots$
- $j = x[i];$
- $x \ll i;$
- $\text{return } i;$

mas não o será nos seguintes casos:

- $i += 10;$
- $\text{printf}("%d", i);$

Considerando o programa da Figura 21, as mutações introduzidas por esses operadores estão mostrados na Tabela 7.

3.3.1.3 Operadores de Incremento e Decremento de Variáveis

Nessa classe existem dois operadores. O primeiro, “DirVarIncDec” coloca um operador de decremento ($--$) e de incremento ($++$) em cada referência a uma

Tabela 7: Mutações relativas ao programa da Figura 21, para operadores de troca de variáveis não de interface

Operador	Mutante	Operador	Mutante
IndVarRepPar	a = i a = *vet i < i *vet < i a += vet[i] a += vet[*vet]	IndVarRepReq	a = -1 a = MININT a = MAXINT -1 < i MININT < i MAXINT < i
IndVarRepConst	a = 1 0 < i 1 < i a += vet[0] a += vet[1]	IndVarRepGlob	a = a a < i
IndVarRepLoc	a = j		a += vet[a]

variável de interface. O operador “IndVarIncDec” funciona da mesma maneira, porém para variáveis não de interface, ou seja, do conjunto L . Ele também adiciona e subtrai 1 de cada uso de constantes dentro da função chamada. As mutações para o exemplo da Figura 21 são mostrados na Tabela 8.

Tabela 8: Mutações relativas ao programa da Figura 21, para operadores de incremento e decremento de variáveis e constantes

Operador	Mutante	Operador	Mutante
DirVarIncDec	j < ++i j < --i a += ++(vet[j]) a += --(vet[j]) return ++a return --a	IndVarIncDec	a = (0 + 1) a = (0 - 1) ++j < i --j < i a += vet[++j] a += vet[--j]

3.3.1.4 Operadores de Inclusão de Operadores Unários

Esses operadores de mutação incluem operadores unários lógico, aritmético e de bit em cada uso de variável ou constante. O operador “DirVarAriNeg” inclui o operador de negação aritmético em cada referência a variáveis de interface. Analogamente, o operador de mutação “DirVarLogNeg” inclui o operador de negação lógica (!) e o operador “DirVarBitNeg” inclui o operador de negação bit a bit (~).

Existem também os correspondentes que atuam sobre referências a variáveis

não de interface e sobre constantes. São eles, “IndVarAriNeg”, “IndVarLogNeg” e “IndVarBitNeg”. As mutações relativas a esses seis operadores para o programa da Figura 21 são mostrados na Tabela 9.

Tabela 9: Mutações relativas ao programa da Figura 21, para operadores de inclusão de operadores unários

Operador	Mutante	Operador	Mutante
DirVarAriNeg	j < -i a += -vet[j] return -a	IndVarAriNeg	a = -0 -j < i a += vet[-j]
DirVarLogNeg	j < ! i a += ! vet[j] return ! a	IndVarLogNeg	a = ! 0 ! j < i a += vet[! j]
DirVarBitNeg	j < ~ i a += ~ vet[j] return ~ a	IndVarBitNeg	a = ~ 0 ~ j < i a += vet[~ j]

3.3.1.5 Operadores de Comandos de Retorno

Os operadores dessa classe têm como objetivo alterar os valores retornados através do comando *return* que, assim como as variáveis de interface, podem ser utilizados para que a função chamada devolva valores à função que a chamou.

O primeiro operador, “RetStaDel”, exclui os comandos *return* da função chamada, um de cada vez. Isso faz com que a função continue executando em vez de parar e retornar um valor, fazendo com que algum resultado diferente seja computado e retornado em algum outro ponto da função.

O outro operador dessa classe, “RetStaRep”, troca a expressão de cada comando *return* por cada uma das expressões dos outros comandos *return*. Dessa forma, para um determinado caso de teste, a função alterada computa os mesmos valores da função original, porém somente o valor retornado é diferente. No exemplo da Figura 21, o operador “RetStaDel” cria um mutante onde o comando *return a* é eliminado, fazendo com que o valor retornado pela função seja indeterminado. O operador “RetStaRep” não gera mutantes para essa conexão.

3.3.1.6 Operadores de Cobertura de Código

Esses operadores são do tipo instrumentado, conforme definido por Agrawal et al. [50]. Eles não modelam erros de integração mas procuram fazer com que

os casos de teste selecionados possuam características mínimas de cobertura em termos de fluxo de controle.

O primeiro deles, “CovAllNode”, é o que garante a cobertura de todos os nós da função, para cada ponto onde ela é chamada. Isso é conseguido incluindo-se no início de cada bloco da função uma chamada à função TRAP_ON_EXECUTION. A execução dessa função faz com que o mutante seja automaticamente distinguido. Assim, ao distinguir-se todos os mutantes desse operador, são executadas todas as funções TRAP_ON_EXECUTION e, conseqüentemente, todos os nós da função.

O segundo, “CovAllEdge”, garante a cobertura de todos os desvios. A instrumentação nesse caso é ligeiramente mais complexa. Ela é feita em cada expressão que determina o desvio do fluxo de controle, como em comandos *if* e comandos repetitivos. Cada uma dessas expressões *expr*, é substituída por TRAP_ON_TRUE(*expr*) e TRAP_ON_FALSE(*expr*). A primeira força o mutante a ser distinguido quando *expr* tem o valor diferente de zero (verdadeiro) e a segunda quando *expr* assume o valor zero (falso). Assim, cada ramo do comando é executado ao menos uma vez. Além disso, cada ramo em um comando *switch* deve também ser executado. Isso pode ser feito utilizando-se a mesma função TRAP_ON_EXECUTION para forçar a execução de um ramo. Por exemplo, no mutante abaixo mostra-se como forçar a execução do ramo “*case 2* :”.

```
switch (a)
{
    case 1:
        s1;
        goto __label__;      /* codigo incluido na geracao do mutante */
    case 2:
        TRAP_ON_EXECUTION(); /* codigo incluido na geracao do mutante */
        __label__: ;         /* codigo incluido na geracao do mutante */
        S2;
    case 3:
        S3;
}
```

Note-se que como a mutação só é habilitada se a chamada da função foi feita através do ponto desejado, como descrito anteriormente, os mutantes estão associados a uma chamada, fazendo com que a cobertura de todos os nós/arcos seja alcançada para cada chamada.

Para o programa da Figura 21, as mutações introduzidas são os mostrados na Tabela 10.

Tabela 10: Mutações relativas ao programa da Figura 21, para operadores de cobertura

Operador	Comando original	Mutante
CovAllNode	a = 0	TRAP_ON_EXECUTION()
	j < i	TRAP_ON_EXECUTION()
	j += 1	TRAP_ON_EXECUTION()
	a += vet[j]	TRAP_ON_EXECUTION()
	return a	TRAP_ON_EXECUTION()
CovAllEdge	j < i	TRAP_ON_TRUE(j < i)
	j < i	TRAP_ON_FALSE(j < i)

3.3.2 Grupo II - Mutações Efetuadas nos Pontos de Chamada

Os operadores desse grupo são aplicados nos pontos onde a função f faz chamadas a g , quando testa-se a conexão $f.g$. Eles são aplicados principalmente aos argumentos das chamadas mas também à chamada como um todo. Quando um argumento é um ponto de mutação e esse argumento é uma expressão, então o operador é aplicado à expressão como um todo e não a partes da expressão como variáveis ou constantes.

3.3.2.1 Operador de Substituição de Argumentos por Constantes

Esse operador substitui cada argumento por constantes – de tipos compatíveis – do conjunto R . O nome desse operador é “ArgRepReq”. As mutações a ele associadas, para o programa da Figura 21, são mostradas na Tabela 11.

3.3.2.2 Operador de Incremento e Decremento de Argumentos

Esse operador “ArgIncDec” incrementa e decrementa cada um dos argumentos da chamada. Como os argumentos são uma expressão qualquer, não necessariamente a referência a uma variável, isso é feito adicionando-se e subtraindo-se o valor 1 em cada argumento.

3.3.2.3 Operadores de Troca de Argumentos

O primeiro operador nessa classe é “ArgSwiAli”, que troca a ordem que os argumentos aparecem na chamada. Troca-se de lugar cada par de argumentos de

Tabela 11: Mutações relativas ao programa da Figura 21, para operadores do Grupo II

Operador	Mutante
ArgRepReq	$c = s(0, b)$
	$c = s(1, b)$
	$c = s(-1, b)$
	$c = s(\text{MININT}, b)$
	$c = s(\text{MAXINT}, b)$
ArgIncDec	$c = s(d+1, b)$
	$c = s(d, b+1)$
ArgSwiDif	$c = s(b,d)$
ArgDel	$c = s(d)$
	$c = s(b)$
ArgAriNeg	$c = s(-d, b)$
ArgLogNeg	$c = s(!d, b)$
	$c = s(d, !b)$
ArgBitNeg	$c = s(\sim d, b)$
FuncCalDel	$c = 0$
	$c = -1$
	$c = 1$
	$c = \text{MININT}$
	$c = \text{MAXINT}$

tipos compatíveis. Outro operador similar é “ArgSwiDif” que faz o mesmo tipo de troca, porém entre argumentos de tipos diferentes. Em C, esse tipo de engano pode ser facilmente percebido em tempo de compilação pois haveria argumentos de tipos incorretos, diferentes daqueles esperados pela função chamada. Porém, isso só acontece se a função chamada for previamente definida ou tiver um protótipo previamente definido. Caso contrário, o erro pode passar despercebido, tornando-se interessante procurar revelá-lo durante o teste. É uma decisão do testador aplicar ou não esse operador, dependendo se ele pretende ou não, durante o teste, atacar esse tipo de erro que é estaticamente detectável.

O operador “ArgSwiAli” não gera mutantes para o programa da Figura 21 pois os argumentos d e b da chamada à função s não podem ser trocados pois possuem tipos incompatíveis. Já o operador “ArgSwiDif” substitui a chamada pelo comando “ $c = s(b, d)$ ”. Se a função s fosse definida antes da função r ou se um protótipo com os tipos dos parâmetros de s fosse definido, então esse operador também não geraria mutantes pois isso causaria um erro sintático no mutante.

3.3.2.4 Operador de Eliminação de Argumento

Esse operador, “ArgDel”, elimina cada um dos argumentos da chamada. Sua aplicação está sujeita às mesmas restrições feitas ao operador “ArgSwiDif” pois pode gerar os mesmos tipos de problemas estaticamente detectáveis. As mutações provocadas por esse operador para o programa da Figura 21 seriam “ $c = s(d)$ ” e “ $c = s(b)$ ”.

3.3.2.5 Operadores de Inclusão de Operadores Unários

Esse operadores são semelhantes àqueles definidos para o Grupo I, todavia aplicados aos argumentos da chamada. O operador “ArgAriNeg” acrescenta o operador de negação aritmético antes de cada argumento. Analogamente, o operador de mutação “ArgLogNeg” acrescenta o operador de negação lógica (!) e o operador “ArgBitNeg” acrescenta o operador de negação bit a bit (~). Na Tabela 11 são mostradas as mutações correspondentes a cada um desses operadores para o exemplo da Figura 21.

3.3.2.6 Operador de Remoção da Chamada

Esse operador, “FuncCalDel”, não é aplicado aos argumentos da chamada da função mas sim à chamada como um todo, que é simplesmente removida. Se o tipo da função chamada for “void”, então basta simplesmente eliminar a chamada da função. No entanto, se a função retorna algum valor, ela pode ser usada como parte de uma expressão. Nesse caso, a chamada é trocada por constantes do conjunto R , conforme o seu tipo. A Tabela 11 dá exemplos de mutações desse tipo.

3.4 Análise Teórica do Critério Mutação de Interface

Nesta Seção são feitas duas análises teóricas sobre propriedades do conjunto de operadores de Mutação de Interface proposto. A primeira delas é sobre a complexidade do critério. Mostra-se que o número de mutantes gerados pelos operadores propostos é determinado, principalmente, pelo número de variáveis vezes o número de referências a variáveis – como também acontece com o teste de mutação no nível de unidades – e pelo número de parâmetros formais da função chamada.

A segunda análise faz uma comparação entre o critério Mutação de Interface e o critério de fluxo de dados interprocedimental de Harrold e Soffa [32], mostrando que esses critérios são incomparáveis.

3.4.1 Análise de Complexidade

A complexidade de critérios baseados em mutação tem sido determinada através da estimativa do número de mutantes gerados. No teste de mutação ao nível de unidade, Budd [51] mostra que o número de mutantes gerados é proporcional ao número de variáveis do programa vezes o número de referências a essas variáveis. Esse resultado foi confirmado empiricamente por Offut et al. [13]. Deve-se notar que o número de mutantes depende do conjunto de operadores de mutação utilizado. No caso desses dois trabalhos, considerou-se como base o conjunto de operadores de mutação para a linguagem Fortran, implementado na ferramenta Mothra [27].

Para analisar a complexidade do critério Mutação de Interface com o conjunto de operadores proposto neste capítulo, serão analisados cada um dos operadores. Supõe-se que o critério esteja sendo aplicado numa conexão **F-G** em que **F** faz uma única chamada a **G**. Denota-se por $mutantes(op)$ o número de mutantes gerados por um determinado operador ou um conjunto de operadores “op” para essa conexão.

- O operador “DirVarRepPar” troca cada ocorrência de uma variável de interface por cada um dos elementos do conjunto $P(G)$. Assim, tem-se:

$$mutantes(DirVarRepPar) = Intfrefs \times |P|$$

$Intfrefs$ é o número de ocorrências de parâmetros formais de G , e indexações e derreferências a esses parâmetros. Como cada ocorrência é substituída por cada elemento de $P(G)$, multiplica-se $Intfrefs$ pela cardinalidade do conjunto $P(G)$.

- Analogamente, para o operador “DirVarRepGlob” tem-se

$$mutantes(DirVarRepGlob) = Intfrefs \times |G|$$

- Portanto, para todo o grupo “DirVarRep” tem-se o número de mutantes representado por

$$mutantes(DirVar) = Intfrefs \times (|P| + |G| + |L| + |E| + |C| + |R|)$$

Note-se que para trocas das variáveis por constantes existem alguns casos que não fazem sentido, quando a variável aparece do lado esquerdo de uma atribuição. Portanto, o número de mutantes para “DirVarRepConst” e “DirVarRepReq” deveria levar esse fato em consideração. Porém, como o interesse aqui é no pior caso, pode-se considerar que esse caso não ocorre dentro de G e assim, toda a ocorrência de uma variável de interface é trocada pela constante.

- Similarmente, chamando de *Nintrefs* o número de ocorrências de variáveis locais (não de interface) ou constantes, obtém-se para o grupo de operadores “IndVarRep”, a seguinte equação:

$$mutantes(IndVarRep) = Nintrefs \times (|P| + |G| + |L| + |E| + |C| + |R|)$$

É importante observar que variáveis não de interface e constantes só são ponto de mutação quando estão num comando que possui também ocorrência de variáveis de interface. Como essas equações representam o pior caso, considera-se que uma ocorrência de variáveis não de interface e de constantes sempre está junto com uma variável de interface e portanto, sempre é ponto de mutação.

- Para o conjunto de todos os operadores de troca de variáveis tem-se

$$mutantes(VarRep) = (Intrefs + Nintrefs) \times (|P| + |G| + |L| + |E| + |C| + |R|) \quad (1)$$

- Para os operadores de incremento e decremento e para os de inclusão de negação tem-se:

$$mutantes(VarIncDec) = (Intrefs + Nintrefs) \times 2 \quad (2)$$

$$mutantes(VarNeg) = (Intrefs + Nintrefs) \times 3 \quad (3)$$

- Para operadores de comando de retorno;

$$mutantes(RetSta) = Rets + (Rets \times (Rets - 1)) = Rets^2 \quad (4)$$

onde *Rets* é o número de comandos de retorno de G.

- Para operadores de cobertura:

$$mutantes(CovAll) = Nos + Arestas \quad (5)$$

onde *Nos* dá o número de nós de G e *Arestas* dá o número de arestas saindo de nós com comandos de decisão como *if* e *switch*.

- Para os operadores do Grupo II, tem-se as seguintes equações:

$$\begin{aligned}
mutantes(ArgRepReq) &= |R| \times Params \\
mutantes(ArgIncDec) &= 2 \times Params \\
mutantes(ArgSwi) &= \binom{Params}{2} \\
mutantes(ArgDel) &= Params \\
mutantes(ArgNeg) &= 3 \times Params \\
mutantes(FunCalDel) &= |R|
\end{aligned}$$

$Params$ representa o número de parâmetros formais de G ². O total então para o Grupo 2 pode ser dado por:

$$mutantes(Grupo II) = (|R| + 6) \times Params + \binom{Params}{2} + |R| \quad (6)$$

Assim, somando-se as equações 2 a 6, obtém-se um limite superior para o número de mutantes da conexão F-G. Percebe-se que o número de mutantes para os operadores do Grupo I, assim como acontece com o teste de mutação convencional, é influenciado, principalmente, pelo número de variáveis (globais e locais) vezes o número de referências a essas variáveis. O número de mutantes para operadores do Grupo II é proporcional ao número de parâmetros da unidade G .

3.4.2 Análise de Inclusão

Wong [59] mostrou analiticamente que no nível de unidade o teste de mutação e o critério todos-usos de Rapps e Weyuker [3] são incomparáveis segundo a relação de inclusão. Mostrou, também, através de estudos empíricos que, na prática, o teste de mutação tende a incluir os critérios de fluxo de dados. Souza [38] mostra, também através de estudos empíricos, que o teste de mutação e os critérios Potencias Usos [4] são incomparáveis.

Nesta seção mostra-se que no nível interprocedimental o mesmo ocorre com o critério de Mutação de Interface utilizando o conjunto de operadores de mutação aqui proposto e o critério todos-usos interprocedimental proposto por Harrold e Soffa [32], ou seja, mostra-se que eles são incomparáveis.

O critério de Harrold e Soffa (critério H&S) basicamente exige que, dado o parâmetro formal v da unidade g , seja exercitado pelo menos um caminho livre de

²Note-se que esse valor é diferente de $|P(G)|$ pois P inclui derreferências a parâmetros do tipo ponteiro ou vetor.

definição que vai de cada ponto onde o valor de v é definido antes da chamada de g - através de uma definição do argumento associado a v na chamada - até cada ponto dentro de g onde v é usada. Exige também que seja exercitado pelo menos um caminho livre de definição que vai de cada ponto em g onde v é definida até cada ponto após o retorno da função onde existe um uso desse valor através do uso do argumento associado a v . Além disso, se uma variável u , passada como argumento para g , associada ao parâmetro formal v , é definida antes da chamada de g e usada após o seu retorno, o critério exige que, entre esses dois pontos, pelo menos um caminho livre de definição em relação a u e a v que “atravesse” g seja exercitado, caso existam tais caminhos. Por exemplo, dado o programa da Figura 22, definem-se as seguintes associações definição-uso: $\langle f_1, (g_1, g_2) \rangle$, $\langle f_1, (g_1, g_3) \rangle$, $\langle f_1, g_2 \rangle$, $\langle f_1, f_3 \rangle$ e $\langle g_2, f_3 \rangle$.

Figura 22: Programa para exemplificar associações requeridas pelo critério de Harrold e Soffa

Assim, pode-se estabelecer:

Teorema 1: Os critérios Mutação de Interface e o critério de H&S são incomparáveis.

Para demonstrar esse teorema deve-se mostrar que 1) o critério Mutação de Interface (MI) não inclui o critério H&S; e 2) o critério H&S não inclui o critério MI. Para tal, considera-se o programa **P**, do qual fazem parte as unidades f e g , de forma que f faz chamadas a g e toda passagem de parâmetros entre elas é feita por referência³.

Inicialmente mostra-se que o critério Mutação de Interface (MI) não inclui o critério H&S. Para isso, basta que seja analisado o caso em que a função g possui

³Essa restrição faz-se necessária pois o critério definido por Harrold e Soffa somente considera relações definição-uso entre unidades quando a interação entre elas se dá através de parâmetros passados por referência ou por variáveis globais.

um parâmetro formal v . A única operação realizada em g é a atribuição de um valor (uma definição) a v . O critério de H&S exige que pelo menos um caminho livre de definição que vai dessa definição até cada ponto $f_{i1}, \dots, f_{in}$ após o retorno de g onde esse valor é usado através da variável u associada a v na chamada, seja exercitado. Uma mutação aplicada no comando que define o valor de v exige, para ser distinguido, um caso de teste que: 1) faça ser executado o comando onde v foi definida (condição de alcançabilidade); 2) provoque uma discrepância no valor atribuído a v (condição de necessidade); e 3) faça com que essa diferença seja propagada até um ponto onde o mutante produza uma saída diferente do programa original (condição de suficiência). Essa última condição exige um uso de v . Como v não é usada dentro de g , esse uso se dá depois do retorno de g , através de um uso de u . Podem existir diversos usos de u após o retorno de G mas todos os mutantes criados dessa maneira podem ser distinguidos através de um único uso de u . Em outras palavras, pode-se distinguir esses mutantes cobrindo-se uma única associação definição-uso requerida pelo critério de H&S, deixando outras não cobertas, e isso acontece para todos os operadores de Mutação de Interface. Assim, mostrou-se que o critério MI não inclui o critério de H&S.

A segunda parte da demonstração corresponde a mostrar que o critério de H&S não inclui o critério MI. Isso pode ser feito através da análise do programa da Figura 23. Para ele são definidas as seguintes associações definição-uso a serem cobertas: $\langle f_1, (g_1, g_2) \rangle$, $\langle f_1, (g_1, g_3) \rangle$, $\langle f_1, g_2 \rangle$, $\langle f_1, g_3 \rangle$, $\langle g_5, f_3 \rangle$, $\langle g_6, f_3 \rangle$. Para satisfazer o critério de H&S, os seguintes casos de teste podem ser usados: (1,0) e (4,4). Porém esses casos de teste não matam o mutante produzido pelo operador “DirVarRepPar” que substitui o comando “ $x = j$ ” no nó g_2 por “ $x = i$ ”. Portanto, H&S não inclui MI.

Pode-se mostrar que para alguns casos especiais, com classes restritas de programas, ou mais precisamente, com tipos restritos de interações entre as unidades, o critério Mutação de Interface inclui o critério H&S. Para mostrar isso, assume-se que **P** não possui anomalias de fluxo de dados, em particular, nenhuma variável é usada no programa sem que seu valor tenha sido previamente definido.

Para o caso em que a conexão F - G tem as seguintes características:

1. g possui apenas parâmetros de entrada, ou mais precisamente, os valores passados como parâmetros são apenas usados dentro de g e não são redefinidos;
2. os valores definidos para as variáveis passadas como argumentos são usados somente dentro de g ; e

```

procedure g(var i, j, y:integer)
var x: integer;
begin
  if i > j then
    x = i;
  else
    x = j;
  if x < 4 then
    y = 0;
  else
    y = 1;
end;

```

```

procedure f;
var ab, b, c:integer;
begin
  read(a,b);
  g(a,b,c);
  write(c);
end

```

Figura 23: Programa que mostra que H&S não inclui MI

3. cada argumento é definido em apenas um ponto antes da chamada de g ;

pode-se estabelecer o seguinte teorema:

Teorema 2: Para conexões que atendam a essas restrições. MI inclui o critério H&S.

Para demonstrar o teorema 2, seja v um parâmetro formal de g . Supondo-se que f_i é o ponto onde o valor de v é definido antes da chamada de G através da definição do argumento u associado a v e que os nós $g_{j1}, \dots, g_{jn}$ são os pontos onde ocorrem c-usos de v e $(g_{k1}, g_{l1}), \dots, (g_{km}, g_{lm})$ são as arestas onde ocorrem p-usos de v , então ficam definidas $n + m$ associações definição-uso interprocedimentais para a variável v que são $\langle f_i, g_{j1} \rangle, \dots, \langle f_i, g_{jn} \rangle$ e $\langle f_i, (g_{k1}, g_{l1}) \rangle, \dots, \langle f_i, (g_{kn}, g_{ln}) \rangle$. Um operador de mutação aplicado no nó g_{jk} que possui um c-uso de v requer um caso de teste que faça com que esse nó seja executado para que tal mutante seja distinguido (condição de alcançabilidade). Se esse nó é o ponto inicial de um p-uso de v , a aplicação do operador “CovAllEdge” faz com que todas as arestas saindo desse nó sejam executadas, cobrindo dessa maneira também esse p-uso. Considerando que existe somente um ponto onde u é definida e que não existe uso de variável sem prévia definição, conclui-se que todas as associações definição-uso são cobertas por um conjunto de casos de teste MI-adequado.

Se existe mais de um ponto antes da chamada de g onde u é definida, então não se garante que todos os usos de cada definição são cobertos. Podem existir

definições que tenham todos os usos cobertos, outras que tenham alguns usos cobertos e outras, nenhum uso.

O segundo caso ocorre quando a conexão obedece as seguintes restrições:

1. g possui apenas parâmetros de saída, ou seja, os parâmetros formais são apenas definidos dentro da unidade e servem para retornar um valor à unidade que chamou;
2. não existe um caminho completo de g livre de definição em relação aos seus parâmetros; e
3. existe, para cada parâmetro de g , somente um ponto após o retorno no qual há um uso (um c-uso) do valor retornado.

Define-se então:

Teorema 3: Para conexões que atendam a essas restrições, MI inclui o critério H&S.

Para demonstrar o teorema, seja v um parâmetro formal de g . Supondo-se que os nós $g_{j1}, g_{j2}, \dots, g_{jn}$ são os pontos onde ocorrem definições de v , e que após o retorno de g existe um c-uso de v no nó f_i , então ficam definidas n associações definição-uso que são $\langle g_{j1}, f_i \rangle, \dots, \langle g_{jn}, f_i \rangle$. Ao aplicar-se um operador em g_{jk} , cria-se um mutante que para ser distinguido exige um caso de teste que execute o nó g_{jk} (condição de alcançabilidade), interfira no valor de v (condição de necessidade) e alcance um ponto onde o mutante produz um resultado diverso do programa original (condição de suficiência), o que por sua vez exige que um uso de v seja alcançado (através do argumento associado a v na chamada de g). Como só existe um uso (em f_i) do valor retornado através de v , garante-se que toda associação $\langle g_{jk}, f_i \rangle$ $k = 1, 2, \dots, n$ é coberta.

Se existe mais de um uso do valor definido através de v após o retorno de g , então não se garante que o critério H&S será incluído pelo critério de Mutação de Interface. Nesse caso porém, pode-se dizer que a Mutação de Interface garante a inclusão do correspondente, no nível interprocedimental, do critério “todas-definições” que exige que cada definição seja exercitada por pelo menos um uso. Isso é assegurado pelo fato mostrado acima de que uma mutação que altera v exige um uso posterior dessa variável para que o mutante seja distinguido.

Note-se que esses casos especiais apresentam restrições que provavelmente não são satisfeitas em casos reais. Por outro lado, é possível que determinadas

variáveis dentro de uma unidade, sigam os padrões estabelecidos nesses dois casos, como por exemplo, uma única definição de uma variável usada posteriormente como argumento de entrada de uma chamada de função.

O Teorema 1 mostrou a incomparabilidade entre os critérios MI e H&S. Esse resultado ressalta a relevância do presente trabalho na medida que evidencia os aspectos complementares desses critérios no nível do teste de integração e motiva a realização de estudos empíricos na perspectiva de estabelecer uma estratégia de teste que agregue esses critérios.

3.5 Critério de Mutação de Interface Alternativos

O conjunto de operadores apresentados neste capítulo é experimental e foi criado com base, principalmente, na experiência anterior do uso do teste de mutação. Ele busca ser um conjunto completo, no sentido de poder exercitar a maioria das interações entre duas unidades. Por outro lado, tenta também ser minimal, no sentido de que as mutações sejam restritas àquelas essenciais, mantendo baixo o custo de aplicação do critério. Mesmo assim, como mostrado na Seção 3.4.1 que analisa a complexidade do critério, pode-se esperar que o número de mutantes gerados pelo conjunto completo de operadores de Mutação de Interface seja bastante alto, mesma para conexões entre unidades simples.

O conjunto de operadores de Mutação de Interface é projetado para modelar os erros de integração a partir da inclusão dos defeitos simples mais comuns para a linguagem C. Por outro lado, é importante reconhecer que cada programa tem diferentes características e pode requerer teste com diferentes requisitos. Dessa forma, é necessário permitir ao testador ajustar, em termos de custo e requisitos de teste, o conjunto de mutantes utilizado na aplicação do critério. Mathur e Wong [39, 22] já o fizeram para o teste de mutação convencional através de critérios alternativos de mutação, definidos de duas maneiras: 1) selecionando aleatoriamente uma pequena porcentagem dos mutantes; ou 2) aplicando apenas um conjunto bastante restrito de operadores de mutação. O mesmo fez Offut et al. [13] com critérios N-seletivos, que deixam de utilizar os N operadores de mutação que mais geram mutantes [13]. Experimentos mostraram que aplicando essas abordagens consegue-se uma significativa redução no custo de aplicação do teste de mutação sem que se perca significativamente na capacidade de revelar erros dos conjuntos de casos de teste selecionados.

Também no nível interprocedimental pode-se utilizar estratégias de restrição

do número de mutantes. Definem-se assim, os **Critérios de Mutação de Interface Alternativos** (MI-alternativos). Através de parametrizações dos operadores de mutação que permitam selecionar qual porcentagem de mutantes gerar e quais operadores utilizar, pode-se estabelecer os critérios que correspondem, no nível interprocedimental, aos critérios de Mathur e Wong e de Offut et al:

Critério MI-aleatório: seleciona-se aleatoriamente para cada operador de Mutação de Interface uma pequena porcentagem dos mutantes.

Critério MI-restrito: utilizam-se apenas os mutantes criados por um pequeno número de operadores de Mutação de Interface.

Critério MI-N-seletivo: não são utilizados os N operadores de Mutação de Interface que mais geram mutantes.

Outro tipo de parametrização desejável que pode ser aplicada aos operadores de Mutação de Interface, e que leva também a uma redução no número de mutantes criados, é permitir que o testador estabeleça, para cada operador, o número máximo de mutantes a serem gerados em cada ponto de mutação. Com essa abordagem tenta-se eliminar mutantes “redundantes” criados por um operador num mesmo ponto do programa e que tendem a se comportar de maneira semelhante, ou seja, tendem a ser distinguidos pelos mesmos casos de teste. Por exemplo, operadores de troca de variáveis por outras variáveis ou constantes tendem a gerar mutantes, no mesmo ponto de aplicação, que se comportam de maneira semelhante [75]. Ainda, outra maneira de parametrizar os operadores de Mutação de Interface é permitir que o próprio testador especifique o conjunto de constantes requeridas, conforme as características do programa em teste. Assim, o conjunto apresentado anteriormente é um conjunto padrão, que pode ser aumentado ou restringido, dependendo da aplicação em teste. Essas idéias complementam as abordagens dos critérios alternativos e podem ser utilizadas de maneira conjunta com eles. Na verdade, mesmo os critérios MI-alternativos podem ser combinados entre si e utilizados em conjunto.

Os critérios MI-alternativos permitem que se estabeleçam estratégias incrementais de teste utilizando a Mutação de Interface. Nessas estratégias, aplica-se uma sequência de critérios MI-alternativos, iniciando-se com um critério de custo reduzido e supostamente menos efetivo, incrementando-o até que o critério de Mutação de Interface completo tenha sido aplicado ou até que restrições de custo estabeleçam o fim do teste. Duas sequências óbvias de critérios seriam: 1) de MI-32-seletivo até MI-0-seletivo; e 2) de MI-aleatório-10% até MI-aleatório-100%. A Figura 24 mostra um exemplo dessa última sequência.

Figura 24: Possível seqüência de aplicação de critérios MI-aleatórios

Essas seqüências são baseadas simplesmente em parâmetros probabilísticos e não levam em conta as características dos operadores de mutação que podem ser importantes para determinar uma seqüência para sua aplicação. Os operadores do Grupo II, por exemplo, modelam alguns dos defeitos mais comuns em termos de chamadas de função e passagem de parâmetros. Na maioria dos casos, esses operadores forçam a seleção de casos de teste que mostrem que cada parâmetro e cada chamada de função desempenham um papel dentro do programa. Por exemplo, se o valor de um argumento não é referenciado ou não influencia qualquer cálculo dentro da função chamada, então um mutante criado através da alteração desse argumento não será distinguido e, provavelmente, o argumento possa ser eliminado da interface da função, uma vez que não é usado. Os operadores do Grupo II tendem a ser mais “leves” em termos do número de mutantes que eles geram. Provavelmente possa ser verificado também que eles são mais fáceis de serem analisados pois pode não ser necessário um completo entendimento da estrutura interna da unidade chamada para que se decida sobre sua equivalência. Esses são portanto, bons candidatos para a geração de um conjunto inicial de mutantes e respectivo conjunto inicial de casos de teste.

Já os mutantes do Grupo I tendem a criar um número maior de mutantes mas exercitam de forma mais completa a interface da função chamada. Os ope-

radores que alteram diretamente as variáveis de interface e comandos de retorno podem formar um segundo nível na geração de mutantes, seguindo-se aqueles que são aplicados em variáveis não de interface e constantes e que podem (ou não) influenciar os valores da interface.

Finalmente, os operadores de cobertura são projetados para aqueles casos em que se deseja assegurar um certo nível de cobertura para os casos de teste. O número de mutantes gerados tende a não ser muito alto e a análise e construção de casos de teste para distinguir tais mutantes tende a ser simples. Por outro lado, é discutível se todos eles são significativos e necessários para que se exercite de maneira adequada a conexão em teste. Assim, esse conjunto de operadores pode ser usado nas primeiras etapas de geração como uma maneira de criar-se um conjunto inicial de casos de teste ou, no extremo oposto, como uma forma de complementar o conjunto de teste obtido, de forma a garantir a cobertura desejada.

3.6 Considerações Finais

Neste capítulo apresentou-se o critério de Mutação de Interface. Esse critério interprocedimental para avaliação e seleção de conjuntos de teste é baseado em mutações e tem como objetivo, exercitar de maneira efetiva as interações entre as unidades de um programa. Para isso, são feitas no programa em teste, pequenas alterações que criam erros de integração. Tais erros caracterizam-se por valores incorretos sendo trocados além dos limites das unidades, através de suas interfaces. Dessa maneira, pode-se restringir as mutações somente a pontos do programa relacionados com as conexões entre as unidades.

Para uma completa caracterização do critério Mutação de Interface faz-se necessária a definição de um conjunto de operadores de Mutação de Interface que estabelecem os tipos de mutantes a serem criados. Definiu-se um conjunto de 33 operadores que procuram ser completos, no sentido de procurar exercitar a maior parte das possíveis interações entre unidades. Esses operadores foram especificamente projetados tendo em mente as características de interação da linguagem C. Entretanto, esse conjunto de operadores pode facilmente ser mapeado para outras linguagens com características semelhantes às da linguagem C.

Duas análises teóricas do critério mostraram que: 1) a exemplo do que ocorre no nível de unidade, critérios de fluxo de dados interprocedimentais [32] e o critério de Mutação de Interface são incomparáveis; e 2) o número de mutantes que devem

ser gerados para uma determinada conexão, pelo conjunto de operadores de Mutação de Interface proposto, é proporcional ao número de variáveis do programa vezes o número de referências a essas variáveis na unidade chamada e ao número de parâmetros formais dessa mesma unidade.

O primeiro resultado é importante porque aponta para os aspectos complementares – já ressaltados no nível de unidade – entre critérios de fluxo de dados e critérios baseados em mutação e sugere a utilização conjunta desses critérios na atividade de teste tanto no nível de unidade quanto no nível de integração. Assim, a elaboração de uma abordagem sistemática para utilização desses critérios é importante e permite que se defina uma estratégia de teste desde o nível de unidade até o nível de integração, utilizando critérios de fluxo de dados e baseados em mutação. A segunda análise mostra que, embora os pontos de aplicação dos operadores de Mutação de Interface sejam limitados a pontos relacionados com uma conexão, o número de mutantes pode ser elevado, o que revela a necessidade de adotarem-se abordagens que restrinjam o número de mutantes para que a Mutação de Interface possa ser aplicada dentro de parâmetros aceitáveis de custo. Para isso foram definidos critérios de Mutação de Interface alternativos que resgatam, no nível interprocedimental, abordagens de redução dos mutantes já utilizadas com sucesso no nível de unidade. Uma avaliação dessas abordagens é apresentada no Capítulo 5.

Capítulo 4

PROTEUM/IM: Definição e Implementação

A aplicação de um critério de teste sem o suporte de uma ferramenta é uma atividade improdutiva e propensa a erros. A análise do programa em teste, a geração de requisitos a serem satisfeitos e a avaliação de conjuntos de teste de acordo com esses requisitos são, em geral, tarefas demasiado complexas para serem realizadas manualmente.

Este capítulo descreve a ferramenta de teste *PROTEUM/IM*, que suporta a aplicação do critério Mutação de Interface. Nela foram implementados os operadores de mutação descritos no Capítulo 3. No contexto deste trabalho tal ferramenta representa contribuição importante, pois sem ela seria impossível a realização de experimentos empíricos para avaliar a aplicação do critério proposto, como os experimentos descritos no Capítulo 5. Atualmente, a ferramenta apóia o teste de programas escritos na linguagem C. Pode ser executada sobre os sistemas operacionais SunOs 4 ou 5 e sobre Linux. A interface gráfica disponível para ela foi implementada utilizando-se os programas Tcl e Tk [76], exigindo assim um ambiente gráfico que os suporte.

Os requisitos fundamentais de qualquer ferramenta que apóie a aplicação de um critério de teste são que, dado um programa **P** a ser testado e um conjunto de teste **T**, essa ferramenta determine automaticamente os requisitos de teste a serem satisfeitos para **P** e avalie a adequação de **T** frente a esses requisitos.

No caso do teste de mutação, essas duas tarefas correspondem à geração dos mutantes de **P** e à execução de cada mutante e comparação dos resultados com o do programa original. Adicionalmente, outras características são desejadas de uma ferramenta desse tipo. Entre elas, pode-se destacar: 1) facilidades para armazenar e manipular casos de teste; e 2) análises e relatórios sobre o comportamento dos casos de teste frente aos requisitos. No caso específico do teste baseado em mutações, é também de suma importância que a ferramenta de teste viabilize a aplicação de critérios alternativos, permitindo que a geração de mutantes seja feita de maneira flexível, de acordo com uma estratégia determinada pelo testador.

Inicialmente, são apresentadas as principais características funcionais da ferramenta. A *PROTEUM/IM* é composta por diversos programas que, executados em conjunto, permitem ao testador aplicar Mutação de Interface no teste de um programa. Isso permite que a atividade de teste possa ser desenvolvida de maneira “batch” através da execução de “scripts” que descrevem os passos a serem seguidos. Além dessa forma de utilização, a ferramenta possui uma interface interativa, gráfica, que o testador pode também utilizar para conduzir o teste. As duas formas de interface são descritas neste capítulo. A segunda parte do capítulo apresenta algumas considerações importantes sobre a implementação da ferramenta. É dada particular ênfase às abordagens utilizadas na aplicação dos operadores de mutação e construção e execução dos programas mutantes. Descreve-se também como informações simples sobre o fluxo de execução dos casos de teste podem ser utilizadas para acelerar a execução dos mutantes.

4.1 Funcionalidade

Basicamente, a ferramenta *PROTEUM/IM* oferece ao testador recursos para, aplicando-se o critério Mutação de Interface, avaliar-se a adequação de um conjunto de teste **T** para um determinado programa **P**. Baseado no resultado dessa atividade, o testador deve ser capaz de melhorar a qualidade de **T** procurando satisfazer os requisitos não satisfeitos, até obter um conjunto MI-adequado. Assim, a ferramenta pode ser usada como instrumento de avaliação bem como de seleção de casos de teste. Com esse objetivo, as operações mínimas que devem ser suportadas pela *PROTEUM/IM* são:

- manipulação de casos de teste: execução, inclusão e exclusão
- manipulação de mutantes: criação, execução e análise

- análise de adequação: escore de mutação e relatórios estatísticos

A *PROTEUM/IM* suporta essas e outras operações, algumas de maneira completamente automática – como a execução de mutantes – e outras através da intervenção do testador – como a análise de equivalência de mutantes. Diversas características foram também adicionadas à ferramenta de forma a facilitar a execução de experimentos. É o caso, por exemplo, de permitir que cada mutante seja executado com todos os casos de teste, independentemente de estar vivo ou morto. Esse não é o comportamento normal mas permite que dados sejam colhidos, por exemplo, sobre a eficácia dos operadores de mutação e que se determinem estratégias de minimização do conjunto de teste [38].

O teste usando-se a *PROTEUM/IM* é dirigido por **sessões de teste**. O estado de uma sessão de teste é caracterizado por uma base de dados identificada por um nome e composta, basicamente, por dados sobre os casos de teste sendo utilizados, dados sobre os mutantes sendo utilizados e alguns arquivos intermediários que descrevem o programa sendo testado. Uma sessão de teste é constituída de uma seqüência de operações realizadas sobre essa base de dados¹ através de chamadas aos programas que compõem a ferramenta. Dessa forma, pode-se iniciar uma sessão de teste – através da criação da base de teste – executar diversas operações, interromper a sessão e retomá-la posteriormente do ponto onde foi interrompida.

As chamadas aos programas da *PROTEUM/IM* podem ser feitas de duas maneiras distintas. A primeira é diretamente na linha de comando da shell do sistema ou em scripts. A segunda é de maneira transparente para o testador, através de uma interface gráfica que cuida de invocar os programas. Essas duas formas de se conduzir uma sessão de teste são descritas nas seções seguintes.

4.1.1 Interface Gráfica

A interface gráfica da *PROTEUM/IM* permite ao testador sem experiência explorar e aprender os conceitos sobre o teste baseado em mutação. Mutação de Interface e sobre a própria ferramenta. Ela também fornece melhores recursos para se visualizar o conjunto de casos de teste e os mutantes, facilitando dessa maneira algumas tarefas como a identificação de mutantes equivalentes. Uma visão geral das operações que podem ser realizadas através da interface é mostrada na Figura 25.

¹ A partir desse ponto utiliza-se o termo “base de teste” para identificar essa base de dados.

Figura 25: Operações disponíveis na interface da *PROTEUM/IM*

O primeiro passo necessário para que se conduza um sessão de teste é identificar qual base de teste deseja-se manipular. Caso trate-se de uma nova base de teste (que ainda não foi iniciada), deve-se então criá-la. Para isso o testador utiliza a operação *Program Test/New* e deve fornecer algumas informações, entre elas um nome que identificará a base de teste. A Figura 26 mostra quais são as informações requeridas pela ferramenta. Se trata-se de uma base de teste que já existe, o testador deve apenas identificar o nome da base de teste a ser utilizada. Na operação de criação, toda estrutura necessária para a condução da sessão de teste é inicializada, ou seja, criam-se os arquivos intermediários, cria-se uma base de casos de teste vazia e cria-se uma base de mutantes também vazia.

Figura 26: Criação de uma base de teste

Uma vez identificada a base de teste a ser utilizada na sessão, operações podem ser realizadas, por exemplo, para construir o conjunto de teste a ser avaliado. A primeira dessas operações permite que o testador adicione interativamente casos de teste ao conjunto. Com essa funcionalidade, ativada através da operação *Test*

Case/Add, o testador deve fornecer os dados de entrada para o caso de teste. A ferramenta executa o programa em teste com esses dados e exibe os resultados para o testador. O testador então faz o papel de oráculo e decide se o caso de teste deve ser incluído ou não (por exemplo, se um erro foi revelado, o testador deve interromper o teste, corrigir o defeito e reiniciar o teste). A tarefa de fornecer os dados de entrada é dividida em duas etapas. Na primeira, o testador fornece os parâmetros requeridos pelo programa quando invocado na linha de comando (Figura 27a). Na segunda etapa, a *PROTEUM/IM* inicia a execução do programa e o testador interage com ele fornecendo os dados requeridos em tempo de execução (dados passados para o programa através de sua entrada padrão “stdin”). A ferramenta se encarrega de capturar os dados fornecidos pelo testador bem como os resultados produzidos (Figura 27b), caracterizando dessa maneira o caso de teste.

Figura 27: Inclusão de um caso de teste; a) parâmetros de linha de comando; b) dados de tempo de execução

Outra maneira de incluir casos de teste é através de sua importação de arquivos que descrevem esses casos de teste. Existem três tipos de arquivos que podem ser importados: 1) outra base de casos de teste da própria *PROTEUM/IM*; 2) uma base de casos de teste da ferramenta de teste POKE-TOOL [24]; e 3) um par de arquivos ASCII que descrevem os parâmetros de linha de comando e os dados da entrada padrão. Da mesma maneira que faz na inclusão interativa, a ferramenta utiliza os dados nesses arquivos para executar o programa em teste e captura os resultados produzidos para caracterizar o caso de teste. A importação de casos de teste é ativada através da operação *Test Case/Import* da interface.

Casos de teste podem ser retirados da base de dados através da operação *Test Case/Delete*. Cada caso de teste tem um número, de acordo com a ordem em que foi incluído. Para excluir um caso de teste o testador deve simplesmente

indicar o intervalo de casos de teste a serem excluídos, fazendo com que os demais casos de teste sejam reenumerados. Por exemplo, num conjunto com 10 casos de teste, o testador determina que sejam removidos os casos de 3 a 7, restando 4, reenumerados de 1 a 4. A *PROTEUM/IM* também permite a exclusão lógica de casos de teste. Quando isso ocorre o caso de teste não é fisicamente removido da base mas apenas temporariamente desabilitado, o que implica que não será usado nas próximas operações de execução dos mutantes. Mais tarde, esse caso de teste pode ser novamente habilitado, passando a ser utilizado nas próximas execuções. A operação de habilitar/desabilitar um caso de teste não faz com que os demais casos de teste sejam reenumerados.

O testador pode também examinar o conjunto de casos de teste vigente. Isso é feito através da operação *Test Case/View*. Como mostra a Figura 28, o testador pode “navegar” através dos casos de teste. Ele obtém informação sobre os dados de entrada e de saída do caso de teste, parâmetros de entrada, tempo de execução do caso de teste, código de retorno e uma indicação do estado habilitado/desabilitado do caso de teste, que pode ser alterado pelo testador. A informação sobre o tempo de execução do caso de teste com o programa em teste é armazenada pois pode ser utilizada para distinguir mutantes que entram num laço infinito. Se um mutante ultrapassa muitas vezes o tempo de execução do programa original com um determinado caso de teste, ele é abortado pela ferramenta e considerado morto pois, caso contrário, sua execução poderia não terminar nunca. O código de retorno também é um ponto utilizado para comparar o comportamento dos mutantes. Todo caso de teste tem que ter uma terminação normal com o programa original. Caso contrário, a ferramenta considera que todos os mutantes são mortos por esse caso de teste pois este é um caso de teste revelador de defeito. Os códigos de retorno dos mutantes devem também indicar uma terminação normal. Caso contrário, o mutante é considerado morto pois seu modo de terminação difere daquele do programa original.

O ponto central no teste baseado em mutações é a manipulação dos mutantes. Os operadores de mutação implementados na *PROTEUM/IM* são aqueles descritos no Capítulo 3. Inicialmente o testador deve selecionar um conjunto de operadores de mutação que deseja utilizar. Isso é feito através da operação *Mutants/Generate* mostrada na Figura 29. Para cada operador pode ser selecionada uma porcentagem de geração que representa a porcentagem de mutantes que serão efetivamente criados para aquele operador. Assim, permite-se que os critérios MI-aleatório e MI-restrito sejam utilizados. Pode-se determinar também o número máximo de mutantes criados pelo operador em cada ponto de mutação. O processo de geração

Figura 28: Visualização de casos de teste

de mutantes não precisa ser feito de uma só vez. Ele pode ser repetido diversas vezes, permitindo que o testador gere um conjunto de mutantes, crie um conjunto de teste adequado de acordo com esses mutantes e volte a gerar mais mutantes e incrementar o conjunto de teste.

Figura 29: Geração de mutantes

Uma vez gerados os mutantes e criado o conjunto de casos de teste a ser avaliado, resta executar os mutantes com os casos de teste e comparar os seus

resultados com os do programa original. A *PROTEUM/IM* executa um mutante com cada caso de teste até que um deles mate o mutante ou até que todos os casos de teste sejam usados com esse mutante. Passa-se então para o próximo mutante. Como citado anteriormente, esse comportamento pode ser alterado escolhendo-se, na criação da base de teste a opção “Research” para o tipo da base de teste. Nesse caso, cada mutante é sempre executado com todos os casos de teste.

Como resultado da execução, mutantes são marcados como vivos ou mortos. Além disso, alguns podem ser considerados “anômalos”. Isso acontece quando a *PROTEUM/IM* não consegue compilar o programa fonte correspondente ao mutante. Essa situação pode ocorrer pois o testador tem a liberdade de determinar, na criação da base de teste, qual compilador e quais opções de compilação devem ser usados na compilação dos mutantes. A mudança aplicada no programa original pode criar uma construção aceita por determinado compilador mas não por outro. Os mutantes anômalos são simplesmente ignorados, não sendo considerados no cálculo do escore de mutação.

Após a execução dos mutantes, o testador pode analisar os mutantes vivos e deve decidir quais são equivalentes e quais não são e, nesse último caso, pode ainda selecionar novos casos de teste para matar tais mutantes. Assim, como para casos de teste, a *PROTEUM/IM* permite que o testador “navegue” através do conjunto de mutantes, como mostrado na Figura 30. Pode-se selecionar os tipos de mutantes que se deseja visualizar, ou seja, os vivos, mortos, equivalentes, anômalos ou qualquer combinação destes. Caso decida-se que um mutante é equivalente, basta marcar a caixa com rótulo “Equivalent” e este passa a ter tal status.

Da mesma forma como casos de teste são habilitados/desabilitados, conjuntos de mutantes podem ser ativados/desativados. Essa característica permite que diferentes conjuntos de mutantes possam ser utilizados numa mesma sessão de teste. Essa é uma característica importante para a condução de estudos empíricos para que se possa, por exemplo, comparar custo e eficiência de diferentes conjuntos de operadores de mutação, como no experimento descrito no Capítulo 5. A seleção de mutantes, executada a partir da operação *Mutants/Select*, é feita também determinando-se porcentagens por operadores de mutação. A diferença é que essas porcentagens referem-se apenas aos mutantes previamente gerados. O resultado é que mutantes desativados não são considerados em operações de execução dos mutantes e nem nos cálculos de adequação.

O estado corrente de uma sessão pode ser obtido através de relatórios fornecidos pela ferramenta. O relatório resumo é exibido através da operação *Status*. Esse relatório, mostrado na Figura 31, traz as principais informações sobre a

Figura 30: Visualização de mutantes

sessão, incluindo o escore de mutação obtido. Também existe um relatório com informações sobre o comportamento dos casos de teste em relação aos mutantes. O testador pode escolher quais informações devem ser exibidas nesse relatório. A Figura 32a mostra as opções de configuração e a Figura 32b mostra um exemplo de relatório.

A operação *Properties* permite ao testador configurar algumas características da ferramenta. Basicamente, pode-se configurar o diretório corrente, onde deve estar o programa a ser testado e onde é criada a base de teste, e o valor de “timeout” para que um mutante seja distinguido através de seu tempo de execução.

Finalmente, a operação *Quit* faz com que o ambiente da interface gráfica seja abandonado.

4.1.2 Scripts de Teste

Executar uma sessão de teste através da interface gráfica é, provavelmente, mais fácil, porém menos flexível que através da chamada direta dos programas da *PROTEUM/IM*. A interface depende de constante interação do testador, pelo menos para passar-se de uma certa operação para a próxima. O uso de scripts de teste que chamam diretamente os programas da *PROTEUM/IM* pode evitar tal problema mas exige um esforço de programação e completo domínio, tanto dos

Status

Directory: /home/elisa/teste

Program Test Name: myprog-1

Source Program: myprog

Executable Program: myprog

Compilation Command: gcc myprog.c -o myprog

Connections:

Type: Test

Test Cases: 1

Total Mutants: 48

Executed Mutants: 9

Active Mutants: 48

Live Mutants: 40

Equivalent Mutants: 0

Anomalous Mutants: 1

MUTATION SCORE: 0.148936

OK

Figura 31: Relatório resumo sobre o estado atual da sessão

Test Case Report

Header:

☐ Operator's Percentage

Body:

☐ Effective Test Cases Only

☐ Number of Not Executed Mutants

☐ Number of Alive Mutants

☐ Number of Mutants by each "Causa Mortis"

☐ Total Number of Dead Mutants

☐ Enabled/disabled

☐ Parameters, Return Code, Execution Time

☐ Input

☐ Output

ConfirmCancel

/home/elisa/teste/myprog-1.lst

PROGRAM TESTE: myprog-1

SOURCE FILE: myprog

TOTAL MUTANTS: 48

ANOMALOUS MUTANTS: 1

ACTIVE MUTANTS: 48

ALIVE MUTANTS: 40

EQUIVALENT MUTANTS: 0

MUTATION SCORE: 0.148936

OPERATORS:

I-CovAllEdg 2

I-CovAllNod 2

I-DivVarAr1Neg 2

I-DivVarBitNeg 1

I-DivVarIncDec 3

I-DivVarLogNeg 4

I-DivVarRepCon 6

I-DivVarRepReq 6

I-IndVarAr1Neg 1

I-IndVarIncDec 3

I-IndVarLogNeg 3

I-IndVarRepCon 5

I-IndVarRepPar 1

I-IndVarRepReq 4

I-RetStaRep 2

Teste Case # 1

Number of Not Executed Mutants: 0

Number of Alive Mutants: 40

Number of Mutants Dead by Stdout: 5

Number of Mutants Dead by Retcode: 0

Number of Mutants Dead by Timeout: 0

Number of Mutants Dead by Trap: 2

Number of Mutants Avoided: 0

Number of Dead Mutants: 7

OK

(a)

(b)

Figura 32: Relatório de casos de teste; a) configuração do relatório; b) exemplo de relatório

conceitos sobre o teste baseado em mutação e o critério Mutação de Interface, quanto dos próprios programas que compõem a ferramenta. Dessa forma, tal tipo de interface deve ser utilizada pelo testador mais experiente. É também essencial para a execução de experimentos onde uma mesma seqüência de passos deve ser efetuada diversas vezes para que se obtenha uma significância estatística dos

resultados.

Os programas da *PROTEUM/IM* são divididos em dois grupos. Os programas básicos e os programas utilitários. Os primeiros são programas de nível mais baixo e atuam diretamente sobre os arquivos ou as bases de dados que compõem uma base de teste. Os utilitários utilizam-se dos básicos para realizar alguma operação na base de teste. Por exemplo, existe um programa básico que cria uma base de dados de casos de teste, outro que cria uma base de dados de mutantes e existe um programa utilitário que cria uma base de teste e para isso utiliza os dois primeiros.

As mesmas operações efetuadas através da interface gráfica podem ser efetuadas através da chamada direta dos programas. Na verdade, esta última maneira é mais flexível uma vez que se dominem os programas e os diversos parâmetros que eles aceitam. Uma completa descrição dos programas e todos os parâmetros pode ser encontrada no manual do usuário da *PROTEUM/IM*[77]. A lista de programas é apresentada na Tabela 12.

Tabela 12: Programas que compõem a ferramenta *PROTEUM/IM*

Programas básicos	
li	transforma um programa C numa representação intermediária
li2nli	cria os grafos de programa e adiciona informação sobre nós dos grafos na representação intermediária
instrum	cria programa instrumentado que produz como saída a lista de nós percorridos por um caso de teste
pctest	cria e manipula o arquivo de teste, que dá características gerais da base de teste
tcas	cria e manipula a base de dados dos casos de teste
muta	cria e manipula a base de dados dos mutantes
opmuta	aplica os operadores de mutação ao programa original, criando descritores de mutação
exemuta	constrói os fontes dos mutantes e executa-os; também usado para ativar/desativar mutantes
report	cria relatório sobre o comportamento dos casos de teste
Programas Utilitários	
test-new	cria uma nova base de teste
tcas-add	inclui um novo caso de teste, interativamente
muta-gen	gera descritores de mutação e os inclui na base de dados dos mutantes
muta-view	exibe mutantes para visualização e análise

A criação de uma base de teste pode ser feita utilizando-se o programa *test-*

new. As mesmas informações que devem ser fornecidas à interface gráfica devem ser passadas como parâmetros para esse programa. Por exemplo, para criar-se no diretório corrente uma base de teste chamada *myprog-1*, cujo programa fonte a ser testado é *myprog.c*, cujo executável é *myprog* e cujo comando de compilação é “*gcc myprog.c -o myprog -w*”, pode-se utilizar o programa *test-new* da seguinte maneira:

```
test-new -D . -S myprog -E myprog -C "myprog.c -o myprog -w" myprog-1
```

Como resultado desse comando, são criados os arquivos intermediários e as bases de dados de casos de teste e de mutantes, ou seja, todos os objetos que formam a base de teste.

Para incluir casos de teste interativamente pode-se usar o programa utilitário *tcase-add*. Os parâmetros de linha de comando, se existirem, são passados como parâmetros para o *tcase-add* e, assim como na interface gráfica, inicia-se a execução do programa em teste para que o testador forneça os dados lidos através da entrada padrão. Para adicionar um caso de teste na base de teste *myprog-1* criada acima, pode-se utilizar o comando

```
tcase-add -p "-l -g 10 file1" myprog-1
```

Nesse caso, a opção “-p” foi utilizada para passar os parâmetros “-l -g 10 file1” para o programa *myprog* que é então executado. A saída é capturada pelo *tcase-add* e juntamente com a entrada forma o caso de teste. No programa *tcase-add* assim como no programa básico *tcase* usado para importar casos de teste, pode-se utilizar a opção “-trac” para que seja incluída na base de casos de teste a lista de nós do programa em teste executados pelo caso de teste. Essa informação pode ser usada para acelerar a execução dos mutantes, conforme descrito adiante. Para utilizar a opção “-trac” o testador deve antes criar uma versão do programa em teste que produza a lista de nós percorridos. Por exemplo,

```
instrum -EE myprog-1 myprog
```

O programa *instrum* cria dois arquivos fontes que o testador deve então compilar juntos criando o programa instrumentado. Esse programa instrumentado, cada vez que executado, produz um arquivo *myprog-1.log*, com a lista de nós percorridos na execução. Essa lista é usada quando um caso de teste é incluído interativamente ou importado, caracterizando para cada caso de teste, os nós por ele percorridos.

Para a geração de mutantes há um outro programa utilitário. Na verdade, ele não só aplica os operadores de mutação, criando os descritores dos mutantes, mas também se encarrega de incluir esses descritores na base de mutantes. Os parâmetros para esse programa são os nomes dos operadores de mutação para os quais deseja-se gerar mutantes e a porcentagem e número máximo de mutantes por ponto de mutação.

Alguns dos nomes dos operadores na ferramenta foram ligeiramente alterados para permitir maior flexibilidade por parte do testador no teste via script. Os operadores do grupo I têm o nome iniciado por “I-” e os do segundo grupo por “II-”. Dessa forma, comandos que utilizam os nomes dos operadores podem utilizar referências ambíguas para referenciar múltiplos operadores. Por exemplo, o testador pode utilizar o nome “I-” ou “II-” para gerar mutantes somente para operadores do grupo I ou do grupo II respectivamente. Além disso, cada nome de operador é composto por partes que procuram identificar o tipo de alteração que ele efetua. Por exemplo, “I-IndVarRepReq” indica que a alteração é aplicada em variáveis (Var) não de interface (Ind), trocando-a (Rep de “Replacement”) pelo conjunto de constantes requeridas (Req). Na implementação da *PROTEUM/IM* cada uma dessas partes é composta por exatamente três letras, também com o objetivo de facilitar a referência ambígua aos operadores. Todo comando que referencie um nome de operador pode utilizar a letra ‘x’ ou ‘X’ para casar com qualquer letra do nome do operador. Por exemplo, a referência “I-xxxVarxxxNeg” diz respeito a todos os operadores de inclusão de operador unário, aplicados em variáveis de interface ou não. Por isso faz-se necessária a padronização do número de letras de cada parte do nome. Por exemplo:

```
muta-gen -I-DirVarRepPar 10 2 myprog-1
muta-gen -I-xxxVarRepGlo 100 1 -II 100 1 myprog-1
```

O primeiro comando irá criar mutantes para o operador “I-DirVarRepPar”. A geração para esse operador fica limitada a, no máximo, 2 mutantes em cada ponto de mutação e além disso, entre esses mutantes selecionam-se apenas 10% que serão efetivamente gerados. O segundo comando mostra que mais de um operador pode ser utilizado na geração de uma só vez e que pode-se utilizar nomes ambíguos para representar mais de um operador. O nome “I-xxxVarRepGlo” refere-se a qualquer operador cujo nome inicie com “I-”, tenha 3 letras quaisquer em seguida e termine com “VarRepGlo”. No caso, os operadores “I-DirVarRepGlo” e “I-IndVarRepGlo” se encaixam nessas condições. O mesmo acontece com a opção “-II 100 1”. Ela refere-se a qualquer operador que inicie com “II”, no caso todos os operadores do grupo II.

A execução dos mutantes é feita através do programa *exemuta*. Basta fornecer o nome da base de teste e o programa inicia a execução dos mutantes marcando-os como vivos, mortos ou anômalos. Na execução dos mutantes pode-se utilizar a informação dos nós percorridos pelos casos de teste para evitar a execução de certos mutantes. Se o caso de teste não exercita os pontos do programa onde foram feitas as alterações, isso significa que o resultado do mutante deve coincidir com o do programa original, pois a parte executada do mutante é exatamente igual à do programa original.

Uma característica interessante desse programa é que pode-se determinar um intervalo de mutantes que se deseja executar. Por exemplo, os comandos abaixo executam inicialmente os mutantes de 101 a 200, depois de 201 até o último mutante e depois de 0 a 100.

```
exemuta -exec -f 101 -t 200 myprog-1
exemuta -exec -f 201 myprog-1
exemuta -exec -t 100 myprog-1
```

A vantagem de tal característica é a possibilidade de se distribuir a execução dos mutantes em diversas máquinas em uma rede local de maneira simples e sem nenhum esforço extra de programação. O programa *exemuta* é também usado para ativar/desativar mutantes. Nesse caso, o testador fornece como parâmetros, quais são os operadores de mutação cujos mutantes devem permanecer ativos e a porcentagem de mutantes a serem mantidos ativos. Por exemplo, o comando abaixo desativa todos os mutantes (opção “-all 0”) menos o operador *I-RetStaDel* que tem 50% dos mutantes gerados mantidos ativos e os outros desativados.

```
exemuta -select -all 0 -I-RetStaDel 50 myprog-1
```

Existe um programa simples para visualização de mutantes. Na verdade, o programa *mula-view* exibe apenas os pontos do programa que foram alterados para geração do mutante. A Figura 33 mostra um exemplo de exibição de um mutante. São mostradas as características principais do mutante como: status, operador que o gerou e conexão na qual ele foi aplicado. Em seguida, são mostrados os pontos do programa original onde foram feitas alterações. No caso de operadores do grupo I, supondo-se que a mutação foi feita na conexão A-B, esses pontos correspondem ao comando onde a função A chama a função B e o ponto dentro da função B que foi alterado. Em seguida, são apresentados os pontos do mutante correspondentes aos comandos que sofreram alteração. Com esse programa o testador pode alterar o estado dos mutantes, marcando-os como equivalentes, vivos ou anômalos.

Figura 33: Programa para visualização e análise dos mutantes

O programa *report* cria o mesmo relatório mostrado na sessão sobre a interface gráfica. A escolha das informações que devem aparecer no relatório é feita através de opções passadas para o programa. O relatório é armazenado num arquivo ao invés de ser mostrado ao testador, como na interface gráfica.

Para exemplificar essas idéias, a Figura 34 mostra um script de teste bastante simples. Nesse script são avaliados 10 diferentes conjuntos de teste em relação ao programa *search* e a um conjunto de 10% de cada operador de mutação. Os casos de teste são importados de arquivos ASCII armazenados em subdiretórios chamados I, II, ..., X. O resultado da avaliação, ou seja, os relatórios para cada conjunto, são armazenados nos mesmos subdiretórios.

4.2 Aspectos de Implementação

Um dos aspectos mais críticos numa ferramenta que apóia o teste baseado em mutação é o gerenciamento dos mutantes. A ferramenta deve analisar o programa fonte, aplicar os operadores de mutação e de alguma forma criar uma série de programas alternativos que devem ser executados e seus resultados comparados com os do programa original. Considerando-se que um dos maiores problemas com o teste baseado em mutação é o custo, em termos do tempo requerido para a execução dos mutantes, deve-se encontrar uma forma eficiente para efetuar-se tal tarefa.

```

#!/bin/csh #####
#
# This script uses Proteum/IM to evaluate 10 test case sets to test #
# the connections in program "search.c". The test case sets are in #
# ASCII files in subdirectories I, II, ..., X. #
# At the end of each evaluation, the report file is placed in the #
# same subdirectory from where test set was imported. #
#####

# create a test session
test-new -C "gcc search.c -o search -w -lm" search

foreach tset (I II III IV V VI VII VIII IX X)
    ### generate 10% of the mutants for all operators, for all connections
    muta-gen -all 10 0 search

    ### import test cases from ASCII files input1, ..., input100
    tcase -ascii -D $tset -I input -f 1 -t 100 search

    ### execute mutants
    exemuta -exec search

    ### produce report and copy it to the subdirectory
    report -tcase search
    mv search.lst $tset

    ### reset test case set and mutant set for the next interaction
    tcase -create search
    muta -create search
end

```

Figura 34: Exemplo de um script de teste que utiliza os programas da ferramenta *PROTEUM/IM*

A maioria das ferramentas de mutação descritas na literatura – entre elas a ferramenta Mothra [27] – usa uma abordagem interpretativa para a execução dos mutantes. Nessa abordagem o programa original é traduzido para uma representação intermediária. Nessa representação intermediária os operadores de mutação são aplicados e então o programa original e os mutantes são interpretados e seus resultados comparados. As principais dificuldades nessa abordagem são: 1) os programas interpretados não executam na mesma velocidade de um programa compilado, fazendo cair a velocidade de execução dos mutantes; e 2) o interpretador pode não reproduzir fielmente o ambiente no qual o programa vai realmente executar, podendo provocar distorções no seu comportamento.

Outra abordagem, usada pela *PROTEUM/IM*, é aplicar os operadores de mutação no programa fonte original, criando mutantes-fontes, compilá-los e executá-los. Dessa maneira, a velocidade de execução dos mutantes é melhorada e tanto

o programa original quanto os mutantes podem ser executados no ambiente real para o qual foram projetados. Por outro lado, isso requer um tempo extra para a compilação dos mutantes-fontes. As duas próximas sessões mostram como é feito o gerenciamento dos mutantes na *PROTEUM/IM*. A primeira mostra como é a aplicação dos operadores de mutação. A seguinte mostra como são executados os mutantes e quais as soluções adotadas para acelerar a execução dos mutantes, mesmo tendo que compilá-los antes da execução.

4.2.1 Aplicação dos Operadores de Mutação

Como citado anteriormente, os operadores do grupo I exigem que a mutação seja habilitada no ponto de chamada antes de ser aplicada. Por exemplo, suponha-se que o operador “I-DirVarIncDec” esteja sendo aplicado na conexão entre as funções f e g , sendo que f chama g . É necessário que no ponto em que f chama g exista uma “notificação” de que g está sendo chamada por este ponto e não por outro ponto do programa. Dessa maneira é possível, dentro de g , verificar se o ponto de chamada é o desejado e como consequência, se a mutação deve ou não ser aplicada. Ao visualizar-se um mutante criado por tal operador observa-se um código como o do exemplo abaixo.

<pre>int g(int a) { return a + 1; } int f() { int i; i = g(0); }</pre>	<pre>int g(int a) { return IF_MUTA(++a + 1, a + 1); } int f() { int i; i = PREPARE_MUTA(g(0)); }</pre>
--	---

A macro *PREPARE_MUTA* indica, antes da chamada de g , que a mutação está habilitada. A macro *IF_MUTA* cuida de verificar se a mutação está habilitada. Em caso afirmativo, o comando *return ++a + 1* é executado e em caso negativo, o comando original *return a + 1* é executado. A mutação é desabilitada no retorno de g para f . A descrição de como isso é feito é apresentada na próxima seção, que trata da execução dos mutantes.

A aplicação dos operadores de mutação é feita em duas etapas. Na primeira, realizada quando é criada a base de teste, cria-se uma representação intermediária do programa fonte sendo testado, chamada de Linguagem Intermediária ou LI [78].

Na segunda, através da análise da LI e da identificação das estruturas que devem sofrer a ação de um operador de mutação, são criados descritores de mutação, incorporados à base de dados dos mutantes. A Figura 35 resume esse processo.

Figura 35: Processo de aplicação dos operadores de mutação

A Linguagem Intermediária foi definida por Carnassale [78] com o intuito de servir de instrumento para a geração do grafo de fluxo de controle de um programa. Além disso, ela identifica estruturas do programa como declarações, comandos de repetição, comandos de seleção, etc. Na Figura 36 vê-se um trecho de um programa C e sua correspondente representação LI.

Cada “comando” da LI identifica uma estrutura no programa fonte. A primeira coluna identifica o tipo de estrutura, como “\$DCL” para declaração, “\$FOR” para comando *for* e “\$Sn” para o n-ésimo comando simples. A segunda coluna dá o número do bloco do grafo de fluxo de controle a que a estrutura pertence, a terceira dá o seu deslocamento no programa fonte, a quarta dá o tamanho da estrutura em termos de caracteres que ela ocupa no programa fonte e a última dá a linha do programa fonte onde a estrutura se localiza. Desta maneira, dado um comando da LI torna-se fácil localizar o texto correspondente no programa fonte e a sua posição no grafo de programa.

A aplicação propriamente dita dos operadores de mutação consiste em analisar o programa LI e, ao identificar-se uma estrutura onde deve-se aplicar um operador, buscar no programa fonte tal estrutura e nela aplicar a mutação. Assim, o programa que aplica os operadores é um parser que ao identificar determinadas construções da linguagem toma certas ações, ou seja, a aplicação dos operado-

	\$DCL	1	1	6	1
	@main	1	9	4	3
	\$DCL	1	9	6	3
int a;	{	1	16	1	4
	\$S01	1	21	8	5
	}	1	30	1	6
main()	@g	1	37	1	8
{	\$DCL	1	33	7	8
a = g();	{	1	41	1	9
}	\$DCL	1	43	13	10
	\$FOR	1	61	3	12
int g()	\$S02	1	66	6	12
{	\$C(01)01	2	73	7	12
int i, x[10];	\$S03	6	81	3	12
	{	3	89	1	13
for (i = 0; i < 10; i++)	\$S04	3	92	16	14
{	\$IF	3	110	2	15
scanf("%d", &a);	\$C(01)02	3	113	7	15
if (a < 0)	{	4	0	0	0
x[i] = 0;	\$S05	4	125	9	16
else	}	4	0	0	0
x[i] = a;	\$ELSE	5	136	4	17
}	{	5	0	0	0
return 0;	\$S06	5	145	9	18
}	}	5	0	0	0
	}	6	158	1	19
	\$RETURN	7	163	9	20
	}	8	173	1	21

Figura 36: Tradução de programa C para Linguagem Intermediária

res. Por exemplo, ao encontrar um comando “\$RETURN”, o parser reconhece que nesse comando deve-se aplicar o operador “I-RetStaDel” que elimina o comando do programa original. Por sua vez, a aplicação consiste em retirar do programa fonte o texto correspondente ao comando *return* e incluir no seu lugar o comando vazio, representado pela cadeia “:”. No caso dos operadores do grupo I, é necessário identificar também os pontos onde deve-se habilitar a mutação, de acordo com as conexões sendo testadas. Para isso é feito um passo extra em que o programa LI é analisado para identificarem-se esses pontos.

Como resultado da análise da LI, são gerados descritores de mutação que descrevem as alterações a serem feitas no programa original de forma a criarem-se os mutantes-fontes. Esses descritores – um para cada mutante – são compostos por uma sequência de alterações do tipo: <início da alteração, tamanho do texto a ser retirado, cadeia a substituir o texto retirado >. Por exemplo, a aplicação do operador “I-RetStaDel” na conexão *main-g* da Figura 36 produziria um descritor do seguinte tipo:

Função que chama: *main*

Função chamada: g

Início da alteração: 21 (deslocamento do comando “ $a = g();$ ”)

Tamanho do texto a ser excluído: 8 (tamanho do comando “ $a = g();$ ”)

Cadeia a ser incluída: “ $a = \text{PREPARE_MUTA}(g());$ ”

Início da alteração: 163 (deslocamento do comando “ $\text{return } 0;$ ”)

Tamanho do texto a ser excluído: 9 (tamanho do comando “ $\text{return } 0;$ ”)

Cadeia a ser incluída: “ $\text{IF_MUTA}(\text{ ; }, \text{return } 0;)$ ”

Essa forma de descrever os mutantes concentra todo trabalho em identificar os pontos de mutação e toda consideração sobre a semântica dos operadores de mutação para essa fase de geração dos descritores. Quando os mutantes forem executados, os descritores são “aplicados” sobre o programa original numa simples substituição de texto, o que torna a criação do mutante-fonte uma tarefa bastante rápida, privilegiando assim a velocidade na execução dos mutantes. Além disso, ela faz com que as demais etapas, em particular a execução dos mutantes, sejam independentes da linguagem do programa em teste.

4.2.2 Execução dos Mutantes

A operação de execução dos mutantes pode ser considerada o “gargalo” do teste baseado em mutação. Por isso, a abordagem utilizada na *PROTEUM/IM* privilegia essa operação procurando acelerar o processo de duas maneiras: 1) criando programas executáveis para os mutantes, ao invés de interpretá-los; e 2) fazendo com que o processo de criar-se o mutante-fonte seja o mais simples e rápido possível. A Figura 37 sumariza o processo de execução dos mutantes.

O problema desse processo é que para cada mutante existe um atraso, devido ao tempo de construção do mutante-fonte – que embora pequeno não pode ser negligenciado – e de sua compilação. Para reduzir esse atraso, a *PROTEUM/IM* constrói mutantes-fontes que na verdade correspondem a mais de um mutante, isso é, um mutante-fonte contém o código de diversos mutantes e não de apenas um deles, fazendo com que o processo de construção/compilação seja realizado apenas uma vez para um grupo de mutantes. Tome-se, por exemplo, o programa da Figura 36 e dois mutantes gerados pelo operador *IndVarRepReq*. O primeiro tem o comando “ $\text{return } 0$ ” substituído por “ $\text{return } 1$ ” e o segundo tem o mesmo

Figura 37: Processo de execução dos mutantes

comando substituído por “*return - 1*”. O mutante-fonte que contém o código para esses dois mutantes é mostrado na Figura 38.

Como se vê no exemplo, a função *g* é substituída por duas outras funções *__1g* e *__2g*, cada uma contendo o código correspondente a um mutante. A nova função *g* no mutante-fonte faz chamadas a uma dessas funções, dependendo de qual mutante deve ser executado. O programa *exemuta*, antes de iniciar a execução de um mutante assinala à variável de ambiente *MUTANTTOEXEC* o valor do mutante a ser executado, “avisando” assim qual função deve ser chamada. Dessa maneira, os dois mutantes podem estar no mesmo programa, exigindo apenas uma compilação. Apesar de o código da função que sofre a mutação ser repetida para cada mutante que compartilha o mutante-fonte, obtém-se uma diminuição bastante acentuada no tempo de compilação pois, em geral, o trecho duplicado corresponde a uma pequena parte do programa, principalmente tratando-se do teste no nível de integração quando os programas a serem testados tendem a ser maiores que simples unidades.

Outro ponto a ser notado na Figura 38 é o controle de quando aplicar ou não aplicar a mutação, dependendo do ponto onde *g* é chamada. Dentro da função *main*, imediatamente antes da chamada de *g*, assinala-se à variável global *__flg_muta__*, o valor 1 (verdadeiro)². Dentro das funções *__1g* e *__2g* essa variável é testada e se seu valor for verdadeiro executa-se o código com a respectiva mutação

²No código da Figura 38 são apresentadas as expansões das macros *PREPARE_MUTA* e *IF_MUTA* citadas anteriormente.

```

int __flg_muta__ = 0;
int a;

main()
{
    a = (__flg_muta__ = 1) ? g(): g();
}

int __1g()
{
    int i, x[10];
    for (i = 0; i < 10; i++)
    {
        scanf("%d", &a);
        if (a < 0)
            x[i] = 0;
        else
            x[i] = a;
    }
    return (__flg_muta__) ? 1 : 0;
}

int __2g()
{
    int i, x[10];
    for (i = 0; i < 10; i++)
    {
        scanf("%d", &a);
        if (a < 0)
            x[i] = 0;
        else
            x[i] = a;
    }
    return (__flg_muta__) ? -1 : 0;
}

int g()
{
    int __LOCAL_VAR__;
    static int __LOCAL_COUNTER__ = 0;
    if ( __flg_muta__ ) __LOCAL_COUNTER__++;
    switch ( atoi(getenv("MUTANTTOEXEC")) )
    {
        case 1: __LOCAL_VAR__ = __1g();
                break;
        case 2: __LOCAL_VAR__ = __2g();
                break;
    }
    if ( __flg_muta__ ) __LOCAL_COUNTER__--;
    if ( __LOCAL_COUNTER__ == 0 ) __flg_muta__ = 0;
    return __LOCAL_VAR__;
}

```

Figura 38: Exemplo de mutante-fonte com o código correspondente a dois mutantes distintos

e caso contrário executa-se o código original, sem mutação. Para operadores de mutação do grupo II isso não é necessário pois somente o ponto de chamada é alterado. Na função g , responsável pela chamada a uma das funções duplicadas, a variável `--flag_muta--` recebe o valor 0 (falso) ao retornar-se a execução à função *main*. A variável local `--LOCAL_COUNTER--` controla o aninhamento da chamada de g , de maneira que se g chama a si mesma recursivamente, de maneira direta ou indireta, todas chamadas “abaixo” da chamada feita pela função *main* não devem, ao retornar, atribuir o valor 0 à variável `--flag_muta--`. Além disso, todas elas fazem com que a função alterada execute o código do mutante e não do programa original pois a variável `--flag_muta--` continua sempre com o valor 1.

Outro ponto importante a destacar-se na execução dos mutantes, e que colabora para acelerar a operação, é a utilização de informação de fluxo de controle para evitar a execução de certos mutantes. Como foi descrito na sessão que trata da funcionalidade da ferramenta, o testador pode, ao adicionar interativamente ou importar casos de teste, colher informação sobre o caminho que o caso de teste executa no programa original. Considerando-se a condição de alcançabilidade para que um mutante seja distinguido, pode-se deixar de executar um mutante com um caso de teste cuja execução não passe pelos pontos em que a mutação foi realizada. Por razão de economia no armazenamento e facilidade de processamento, a *PROTEUM/IM* não armazena a seqüência de execução dos nós executados pelos casos de teste mas apenas quais são esses nós. De qualquer forma, essa informação é suficiente para eliminar um grande número de execuções. haja visto que um mutante relacionado com a conexão $f-g$ só deve ser executado com um caso de teste t se: 1) ele foi gerado pela aplicação de um operador do grupo II num determinado nó k do programa original e a execução de t exercita esse nó; ou 2) ele foi gerado pela aplicação de um operador do grupo I no ponto de chamada da função g , localizada no nó k , e dentro da função g , no nó l , e a execução de t passa por esses dois pontos. Nesse segundo caso, se a execução de t passa apenas pelo nó k , o mutante não será distinguido pois a alteração nesse ponto não altera o comportamento do programa. Se a execução passa somente pelo nó l isso significa que a mutação não foi nunca habilitada e da mesma forma o mutante não pode ser distinguido. Por outro lado, é possível que k e l sejam executados por t sem que a conexão $f-g$ seja exercitada. Porém, como não se conhece a ordem de execução dos nós, pode-se decidir executar o mutante com tal caso de teste. Nesse caso, pode-se executar um mutante desnecessariamente mas isso não altera o resultado da avaliação dos casos de teste pois, de qualquer maneira, ele não será distinguido pela execução.

4.3 Considerações Finais

Este capítulo discutiu a ferramenta de teste *PROTEUM/IM* que apóia a aplicação do critério Mutação de Interface para programas C. Foram descritas as principais funcionalidades da ferramenta através de duas possíveis interfaces. A primeira, uma interface gráfica permite ao testador menos experiente familiarizar-se com os conceitos sobre o critério, e sobre o teste baseado em mutações de forma geral, sendo assim importante instrumento para que se possa aplicar a ferramenta, por exemplo, em atividades de ensino. A segunda maneira de utilizar-se a *PROTEUM/IM* é através da elaboração de scripts de teste. Esses scripts descrevem seqüências de comandos da ferramenta que caracterizam uma sessão de teste. Essa forma de interface propicia ao testador mais experiente uma forma de conduzir uma sessão de teste sem a necessidade de constante interação com a ferramenta. Scripts de teste têm se mostrado de grande valor na condução de estudos empíricos que requerem a repetição de sessões de teste para que se obtenha uma significância estatística dos resultados. Essa tarefa tende a ser demorada tornando os scripts um instrumento indispensável para sua execução.

Apresentaram-se também alguns aspectos da implementação da ferramenta. Utiliza-se para a execução de mutantes uma abordagem compilativa, ou seja, criam-se mutantes-fontes a partir do programa original e esses mutantes-fontes são compilados e então executados. Essa abordagem, em contraste com a abordagem interpretativa, tem a vantagem de que a execução dos mutantes é acelerada e que os mutantes podem ser executados no ambiente real para o qual o programa em teste foi desenvolvido.

A aplicação dos operadores de mutação no programa original é baseada na análise de uma Linguagem Intermediária que permite a identificação de estruturas do programa onde os operadores devem ser aplicados, como declarações, comandos e expressões. Com a aplicação dos operadores são criados descritores de mutação que determinam como o programa original deve ser modificado – através de simples substituições de cadeias – para que sejam gerados os mutantes. A cada mutante corresponde um descritor de mutação.

Finalmente, mostraram-se as abordagens utilizadas para acelerar a execução dos mutantes. A primeira delas, através da criação de mutantes-fonte que contêm o código relativo a mais de um mutante, diminuindo o número de compilações necessárias. A segunda utiliza informação do fluxo de controle dos casos de teste para acelerar a execução de mutantes. Basicamente, mutantes não precisam ser executados por casos de teste cuja execução não atinja os pontos onde foram

aplicadas as mutações. Isso faz com que um grande número de execuções seja evitado, acelerando o processo.

A definição e implementação da *PROTEUM/IM* é de fundamental importância pois permite que estudos empíricos envolvendo o critério Mutação de Interface possam ser conduzidos. Dentro do escopo desta tese, a ferramenta foi utilizada para conduzir experimentos de avaliação do custo e efetividade do critério. Esses experimentos estão descritos no Capítulo 5.

Capítulo 5

Avaliação Empírica do Custo e Eficácia do Critério Mutação de Interface

Neste capítulo são apresentados dois estudos de caso envolvendo a utilização da Mutação de Interface, com o objetivo de medir o custo de sua aplicação, bem como a sua efetividade em revelar erros. No primeiro experimento, um programa de ordenação UNIX foi semeado com alguns defeitos e então aplicou-se a Mutação de Interface com o objetivo de avaliar-se o custo de sua aplicação e sua efetividade em revelar os defeitos semeados. Critérios alternativos foram também experimentados através da utilização de diversos subconjuntos de operadores de mutação. Além disso, compararam-se as efetividades em revelar erros do critério Mutação de Interface com o do teste aleatório. O objetivo foi procurar evidências de que as medidas de efetividade obtidas não estão ligadas a características secundárias dos conjuntos de casos de teste – como sua cardinalidade – mas sim, ao modo como eles são selecionados pela Mutação de Interface.

No segundo estudo foi utilizado um programa chamado *SPACE*, desenvolvido para a Agência Espacial Européia para o cálculo de antenas. O programa *SPACE* possui um histórico documentado que descreve os defeitos que foram encontrados durante o seu desenvolvimento. Com essa informação procurou-se reproduzir o processo de teste do programa, iniciando-se com uma versão recheada com diversos desses defeitos e, aplicando-se Mutação de Interface, avaliou-se quantos desses

defeitos puderam ser removidos. Avaliou-se também o custo desse processo.

As próximas seções descrevem os programas utilizado, os métodos empregados nos experimentos, os resultados obtidos e a análise desses resultados, além das considerações finais sobre os experimentos.

5.1 Estudo de Caso 1: O Programa *Sort*

Para este experimento utilizou-se um programa extensivamente utilizado na prática. O programa *Sort* foi selecionado para este experimento por tratar-se de um programa não trivial em termos de estrutura e tamanho, como mostram a Figura 39 e a Tabela 13. A Figura 39 mostra as interações entre as unidades do programa através de seu Grafo de Chamadas. A Tabela 13 dá uma indicação do “tamanho” do programa. Além da complexidade e do tamanho, que habilitam o programa para esse tipo de experimento, o programa faz uso das mais diversas construções e estruturas da linguagem C, sendo assim uma boa arena para experimentação dos operadores de Mutação de Interface e para o próprio critério.

Figura 39: Grafo de Chamadas do programa *Sort*

Esse programa foi anteriormente usado por Wong [59] em outros experimentos. Naquele trabalho Wong descreve um conjunto de 25 defeitos implantados no

Tabela 13: Métricas relacionadas com as funções do programa *Sort*

Função	LOC's	McCabe	Conexões
main	112	35	19
sort	67	16	13
merge	92	29	10
disorder	7	2	2
newfile	8	2	4
setfil	10	3	0
oldfile	8	3	3
safeoutfil	14	7	1
cant	4	1	2
diag	6	1	1
term	11	3	4
cmp	98	38	3
cmpa	14	3	0
skip	36	14	1
eol	4	2	0
copyproto	7	2	0
field	50	16	1
number	11	2	0
qsort	65	14	1
TOTAL	624	193	65

programa *Sort*. Daquele conjunto de defeitos foram seleccionados 11 para serem utilizados neste experimento. O critério utilizado para essa seleção foi escolher aqueles defeitos mais difíceis de serem revelados. Para fazer tal cálculo computou-se o **Índice de Dificuldade** (Hardness Index ou HI) para cada um dos 25 defeitos e escolheram-se aqueles cujos índices alcançam, no máximo, a casa dos 25%.

Para o cálculo dos HI's foram criadas 25 versões do *Sort*, cada uma tendo um defeito implantado. Essas versões foram executadas com um conjunto de 500 casos de teste gerados aleatoriamente. Assim, o HI de cada defeito é definido como a percentagem desses casos de teste que revelam o defeito implantado, ou seja, cujo resultado difere daquele apresentado pelo programa *Sort* sem nenhum defeito implantado, usado como oráculo. Dessa maneira, seleccionaram-se 11 defeitos que intuitivamente são os mais difíceis de serem revelados.

Para esses 11 defeitos calcularam-se os respectivos Índices de Dificuldade utilizando-se não um conjunto de 500 casos de teste, mas 30 diferentes conjuntos. Os defeitos, numerados de I até XI e os respectivos índices são mostrados na Tabela 14.

Tabela 14: Defeitos implantados no programa *Sort*

	Função	Comando original/Defeito	HI
I	merge	(*compare)(ip[0]->l,ip[-1]->l)<0 (*compare)(ip[0]->l,ip[-1]->l)<=0	3,99
II	cmp	if (pb<lb && *pb==tabchar) if (pb<lb *pb==tabchar)	4,63
III	skip cmp	c == ' ' c == '\t' c == ' ' && c == '\t'	25,20
IV	field	p->ignore = dict+128; p->ignore = dict+290;	7, 17
V	main	break; continue;	5,57
VI	qsort	for (k=lp+1; k<=hp;) for (k=lp+1; k<hp;)	13,20
VII	cmp	if (b = *--ipb - *--ipa) if (b = *--ipb - --*ipa)	10,60
VIII	merge	while ((*dp++ = *cp++) != '\n'); while ((*++dp = *cp++) != '\n');	13,77
IX	cmp	if (*--ipa != '0') if (--*ipa != '0')	11,53
X	cmpa	-fields[0].rflg -fields[0].nflg	23,23
XI	merge	while (++k < j) while (++j < k)	2,87

5.1.1 Descrição do Experimento

As versões com defeito do *Sort*, batizadas de versões I, II, XI, foram então “testadas” utilizando-se a Mutação de Interface. Mutantes foram gerados utilizando-se todos os operadores de mutação, exceto os operadores de cobertura de nós e de arestas (CovAllNod e CovAllEdg). Esses operadores foram aplicados a 7 conexões do programa, descritas na Tabela 15. Nessa tabela, a primeira coluna mostra quais são as funções que fazem chamadas às funções da segunda coluna. A terceira coluna fornece o número de vezes que a conexão existe no programa, ou seja o número de chamadas que a função que chama faz à função chamada.

Experiências anteriores [75] mostram que operadores de troca de variáveis e de constantes, quando aplicados diversas vezes no mesmo ponto, tendem a gerar mutantes que se comportam de maneira semelhante. Por exemplo, trocando uma ocorrência de uma variável por todas as outras variáveis compatíveis são gerados *N* mutantes – onde *N* é o número de variáveis compatíveis – que têm a tendência a se comportar de maneira semelhante, ou seja, se um deles é distinguido por um caso de teste, os demais são também distinguidos pelo mesmo caso de teste. Assim,

Tabela 15: Conexões onde foram aplicados os operadores de Mutação de Interface no programa *Sort*

Função que chama	Função chamada	# de chamadas
main	sort	1
merge	qsort	1
sort	qsort	1
qsort	qsort	2
merge	disorder	2
cmp	skip	4
cmp	cmpa	1

apenas um deles seria suficiente para selecionar o mesmo caso de teste. Por isso, nos experimentos, a aplicação dos operadores de troca de variáveis e constantes foi limitada, utilizando as características de parametrização dos operadores de Mutação de Interface. Operadores “DirVarRep” que atuam sobre variáveis de interface foram limitados a, no máximo, 2 mutantes por ocorrência de variável. Os operadores “IndVarRep” foram limitados a, no máximo, 1 mutante por ponto de mutação. Coincidentemente, o número de mutantes gerados para cada uma das versões do *Sort* foi sempre o mesmo. A Tabela 16 mostra o número de mutantes gerados para cada um dos operadores.

Tabela 16: Número de mutantes gerados, por operador, para o programa *Sort*

Operador	Mutantes	Operador	Mutantes
DirVarRepPar	114	DirVarRepGlo	46
DirVarRepLoc	118	DirVarRepExt	232
DirVarRepCon	118	DirVarRepReq	116
IndVarRepPar	98	IndVarRepGlo	83
IndVarRepLoc	107	IndVarRepExt	147
IndVarRepCon	73	IndVarRepReq	67
DirVarIncDec	240	IndVarIncDec	230
DirVarAriNeg	58	IndVarAriNeg	67
DirVarLogNeg	58	IndVarLogNeg	67
DirVarBitNeg	58	IndVarBitNeg	67
RetStaDel	12	RetStaRep	10
ArgRepReq	0	ArgStcAli	6
ArgStcDif	1	ArgDel	8
ArgAriNeg	4	ArgLogNeg	4
ArgBitNeg	4	ArgIncDec	52
FunCalDel	44	TOTAL	2309

Usando o conjunto de operadores de mutação descrito acima, foram construí-

dos 30 conjuntos de teste adequados à Mutação de Interface (MI-adequados). O processo de criação de cada um desses conjuntos é esquematizado na Figura 40. Inicialmente, são marcados os mutantes equivalentes. Depois, um pool de 500 casos de teste é gerado aleatoriamente e a partir desse pool, os casos de teste são retirados e utilizados na execução dos mutantes. Somente os casos de teste que distinguem pelo menos um mutante – ou seja, aqueles “selecionados” pelos mutantes – são mantidos no conjunto MI-adequado, os demais são descartados. Esse processo é repetido até que se obtenha um escore de mutação de 100% ou até que o pool seja esgotado. Nesse último caso, pode-se obter um conjunto não 100% adequado¹.

Figura 40: Processo de geração de um conjunto de teste MI-adequado para o programa *Sort*

O processo de identificação dos mutantes equivalentes também foi automatizado. Antes de iniciar-se a criação dos conjuntos MI-adequados, os mutantes de cada um dos experimentos de I a XI foram executados com um conjunto de 500 casos de teste gerados aleatoriamente. Mutantes não distinguidos nesse processo foram considerados equivalentes nas fases seguintes do experimento. Assim, a operação “Marcar Mutantes Equivalentes” que aparece na Figura 40 consiste simplesmente em marcar como equivalentes os mutantes previamente identifica-

¹Todos os conjuntos gerados durante o experimento atingiram adequação acima de 95%.

dos como tal. Essa certamente não é uma maneira precisa de determinarem-se os mutantes equivalentes porém é uma abordagem conservadora. Ela pode fazer com que mutantes que não são equivalentes sejam marcados como tal. Como consequência, durante a construção dos conjuntos MI-adequados, pode-se deixar de selecionar um caso de teste que mataria aquele mutante. Isso por sua vez só pode fazer com que o conjunto MI-adequado seja menos efetivo em revelar um defeito, nunca mais efetivo. Dessa forma, as medidas de efetividade obtidas são pessimistas.

Uma vez obtido um conjunto MI-adequado, sua efetividade é verificada pela sua capacidade em revelar o defeito semeado na versão do *Sort* para o qual ele foi construído. Para isso, cada caso de teste do conjunto é executado com a versão original do *Sort*, sem defeito, e com a versão incorreta. Se para algum dos casos de teste os resultados obtidos são diferentes, conclui-se que o caso de teste selecionado é capaz de revelar o defeito. Como são gerados 30 conjuntos adequados para cada versão incorreta do *Sort*, pode-se obter uma inferência estatística sobre a efetividade em revelar erros desses conjuntos. Na Tabela 17 são mostrados os resultados para cada um dos experimentos, isto é, a média e o desvio padrão do escore de mutação e do número de casos de teste em cada conjunto adequado e a porcentagem de conjuntos que revelam o defeito.

No teste baseado em mutação no nível de unidade, experimentos têm mostrado que a utilização de um número alto de operadores de mutação pode ser substituída por um subconjunto de operadores que podem alcançar a mesma efetividade em revelar defeitos porém com um custo sensivelmente inferior [18]. Intuitivamente, o mesmo deve ocorrer ao aplicar-se Mutação de Interface Assim, a aplicação do conjunto completo no teste de um programa pode levar a um conjunto de mutantes superdimensionado que poderia ser substituído por outro menor e com a mesma efetividade. Por isso, foram utilizados também nesse experimento os critérios MI-alternativos.

Foram selecionados alguns subconjuntos de mutantes que foram executados com os conjuntos MI-adequados. Após essa execução, somente aqueles casos de teste que distinguem algum mutante desse subconjunto são mantidos, selecionando-se assim um subconjunto do conjunto MI-adequado. A efetividade desses novos conjuntos, adequados aos critérios alternativos, foram então medidas para se avaliar qual seria a efetividade obtida e qual seria o custo do teste se fossem utilizados somente esses critérios. As Tabelas 18 e 19 mostram as medidas de efetividade obtidas por alguns dos critérios alternativos. Os critérios MI-alternativos utilizados foram:

Tabela 17: Resultados de efetividade

EXPERIMENTO	ESCORE DE MUTAÇÃO	NÚMERO DE CASOS DE TESTE	CONJUNTOS REVELADORES (%)
I	0.993883	44,10	100
	0.061714	1,98	
II	0.980643	44,77	100
	0.056256	2,24	
III	0.982352	42,30	100
	0.051813	1,87	
IV	0.944547	45,77	100
	0.057354	2,50	
V	0.969903	45,00	100
	0.051361	1,74	
VI	0.984760	45,13	100
	0.102494	1.97	
VII	0.986067	41,33	100
	0.049830	2,57	
VIII	0.983761	46,70	100
	0.109722	2,48	
IX	0.987760	42,33	100
	0.045523	2,33	
X	0.962617	40,17	100
	0.060709	2,12	
XI	0.999986	17,13	96
	0.004505	2,79	
Média	0.979662	41,34	99.64
Desvio	0.106719	1,99	0.79

- Critério MI-aleatório-10%: 10% dos mutantes selecionados aleatoriamente;
- Critério MI-restrito com operadores do grupo I;
- Critério MI-restrito com operadores do grupo II;
- Critério MI-restrito com operadores aplicados em variáveis de interface;
- Critério MI-restrito com operadores aplicados em variáveis não de interface e constantes;
- Critério MI-restrito com operadores aplicados em comandos de retorno; e
- Critério MI-N-seletivo, com N variando de 1 a 30 ².

²Como a ferramenta *PROTEUM/IM* não possui facilidades para aplicação de critérios MI-N-

Tabela 18: Efetividade dos critérios MI-aleatório e MI-restritos

Experimento	MI-10%	Gr. I	Gr. II	DirVar	IndVar	RetSta
I	96	100	70	100	100	90
II	100	100	86	100	100	60
III	100	100	100	100	100	86
IV	100	100	80	100	100	60
V	100	100	96	100	100	66
VI	100	100	100	100	100	76
VII	100	100	100	100	100	93
VIII	100	100	96	100	100	70
IX	100	100	100	100	100	93
X	100	100	100	100	100	100
XI	96	96	96	96	96	96
Média	99,27	99,64	93,09	99,64	99,64	80,91
Desvio	1,13	0,79	2,93	0,79	0,79	3,90
Mutantes	223	2186	123	1158	1006	22

Tabela 19: Efetividade dos critérios MI-N-seletivos

Experimento	6 Sel	17 Sel	19 Sel	20 Sel	21 Sel	23 Sel	24 Sel	25 Sel
I	100	100	100	96	93	83	20	13
II	96	96	96	86	66	60	33	23
III	100	100	100	100	96	90	83	76
IV	100	96	93	93	66	56	36	30
V	100	100	96	83	66	63	33	30
VI	100	100	100	100	76	63	36	33
VII	100	100	100	100	96	96	56	50
VIII	100	100	96	86	73	66	66	63
IX	100	100	100	100	93	90	56	46
X	100	100	100	100	100	96	70	70
XI	96	96	96	96	96	96	96	96
Média	99,27	98,91	97,91	94,55	83,73	78,09	53,18	48,18
Desvio	1,13	1,35	1,64	2,46	3,93	4,20	4,61	4,69
Mutantes	1224	307	191	139	93	37	27	19

Outro ponto abordado nesse experimento foi a respeito de quais conexões são ou não importantes para o teste do programa. É possível revelar os mesmos defeitos aplicando-se o teste em apenas uma ou um subconjunto das conexões testadas? Procedeu-se então da mesma maneira feita para os critérios alternativos, desta vez selecionando-se os mutantes por conexão. Em outras palavras, considerando-se apenas os mutantes gerados para uma das conexões, foram selecionados subcon-

seletivos, determinou-se o número de mutantes de cada operador e selecionou-se manualmente os conjuntos de mutantes correspondentes a cada critério MI-N-seletivo.

juntos dos conjuntos MI-adequados que dessa forma são adequados ao teste da conexão individualmente. Esses subconjuntos de casos de teste foram então avaliados quanto à sua efetividade em revelar o respectivo defeito. O resultado é mostrado na Tabela 20.

Tabela 20: Efetividade dos conjuntos de teste restritos a cada conexão

Experimento	cmp skip	merge qsort	cmp cmpa	sort qsort	main sort	qsort qsort	merge disorder
I	96	96	93	0	0	33	0
II	96	100	73	100	53	80	0
III	100	100	100	100	93	93	36
IV	96	100	60	96	96	30	0
V	100	70	53	43	36	33	3
VI	100	100	0	100	100	100	0
VII	100	100	100	100	100	100	10
VIII	96	96	73	0	3	16	43
IX	96	100	96	100	100	93	3
X	100	90	100	100	100	100	0
XI	96	96	96	96	96	90	20
Média	97,82	95,27	76,73	75,91	70,64	69,82	10,45
Desvio	1,55	2,29	4,89	6,16	6,32	5,96	3,62
Mutantes	1080	264	250	264	367	20	64

Por último, o experimento procurou mostrar que as características de efetividade dos critério de Mutação de Interface não são somente relacionados com o número de casos de teste que compõem os conjuntos, mas sim com a maneira como os casos de teste são selecionados. Em outras palavras, procurou-se mostrar que o fato de os casos de teste serem responsáveis por distinguir mutantes elevam sua efetividade em revelar defeitos. Para isso, foram gerados aleatoriamente, para cada experimento, 30 conjuntos de casos de teste com as mesmas cardinalidades dos conjuntos MI-adequados e verificou-se sua efetividade em revelar o defeito semeado na correspondente versão do *Sort*. O mesmo foi feito considerando-se as cardinalidades dos conjuntos MI-19-seletivo adequados e MI-aleatório-10% adequados, que são critérios alternativos que apresentam uma alta redução no número de mutantes com uma pequena redução na efetividade em revelar defeitos. Dessa maneira, obtém-se uma comparação entre a efetividade do teste randômico contra a efetividade do critério Mutação de Interface e de dois critérios MI-alternativos.

A Tabela 21 mostra a média da cardinalidade para cada um dos três critérios. Para cada critério existem dois valores. O primeiro é simplesmente a média da cardinalidade de cada conjunto de teste. O segundo é a média considerando os

conjuntos adequados somente até o primeiro caso de teste que revela o defeito (ou o tamanho total, se o conjunto não contém tal caso de teste). Esse segundo valor, que será chamado de “cardinalidade minimizada”, é também importante pois proporciona uma comparação mais real e menos pessimista. Quando aplica-se um critério para selecionar casos de teste, o testador ao deparar-se com um caso de teste que revele um erro, em geral, interrompe o teste para que o defeito seja localizado e corrigido. Assim, se os conjuntos de teste MI-adequados fossem selecionados dessa maneira, sua cardinalidade seria a minimizada. Por isso, esse valor é utilizado também na comparação com os casos de teste gerados aleatoriamente. A Tabela 22 apresenta a porcentagem de conjuntos de casos de teste gerados aleatoriamente que revelam os defeitos dos programas, considerando esses conjuntos com as seguintes cardinalidades: 1) mesma cardinalidade que casos de teste MI-adequados; 2) cardinalidade minimizada dos conjuntos MI-adequados; 3) cardinalidade dos conjuntos MI-aleatório-10% adequados; 4) cardinalidade minimizada dos conjuntos MI-aleatório-10% adequados; 5) cardinalidade dos conjuntos MI-19-seletivo adequados; e 6) cardinalidade minimizada dos conjuntos MI-19-seletivo adequados.

Tabela 21: Cardinalidade dos conjuntos MI-adequados

Experimento	MI		MI-10%		19-Seletivo	
I	44.10	15,13	20,17	11,70	16,73	9,37
II	44,77	13,57	23,30	10,40	16,43	8,97
III	42.30	4,67	18,13	4,50	15,73	4,50
IV	45.77	9.73	22.23	8,17	17,53	7,77
V	45.00	11,87	20.37	8,43	15,73	7,67
VI	45.13	6,33	22,37	5,73	17,37	5,47
VII	41.33	6,90	21,23	5,47	14,70	5,30
VIII	46.70	4,83	22,07	4,03	17,20	4,20
IX	42.33	7,77	21,33	6,07	15,57	5,73
X	40.17	4,00	20.47	3,70	15,23	3,50
XI	17,13	17,13	11,63	11,63	10,27	10,27
Média	41,34	9.27	20,30	7,62	15,68	6,61
Desvio	1,99	2,06	1,33	1,70	1,05	1,51

5.1.2 Análise dos Dados

Uma das difíceis questões colocadas ao se conduzir um experimento como o descrito neste capítulo diz respeito à seleção dos programas e defeitos – artificialmente introduzidos – a serem utilizados. Uma pergunta a ser levantada é: *até onde os*

Tabela 22: Efetividade dos conjuntos randomicamente gerados

Experimento	MI		MI-10%		19-Seletivo	
I	93	50	70	43	60	33
II	86	43	63	30	40	33
III	100	56	100	56	100	56
IV	93	40	73	36	66	33
V	86	50	63	43	53	40
VI	100	60	100	60	96	60
VII	93	50	76	43	76	40
VIII	100	30	93	30	86	30
IX	100	63	100	56	100	53
X	100	53	93	46	80	46
XI	40	40	26	26	13	13
Média	90.09	48.64	77.91	42.64	70.00	39.73
Desvio	3.20	2.80	4.32	2.89	4.80	3.18

defeitos utilizados representam o universo de defeitos reais, de maneira que possamos generalizar os resultados colhidos? A resposta a esta pergunta é bastante difícil. A menos que programas e defeitos reais sejam utilizados, não se pode sequer afirmar que estes são representativos de realidade do teste de software. Mesmo que defeitos reais sejam usados, não se pode ainda com certeza generalizar os resultados para outros softwares e outros tipos de defeitos [14]. Dessa maneira, o objetivo de tais experimentos é produzir um estudo de caso no qual, independentemente de quais defeitos sejam utilizados, pode-se medir custo e efetividade de um certo critério para essa particular arena e, se possível, compará-lo com outros critérios. Bons resultados podem apenas sugerir novos experimentos e encorajar seu uso prático. O problema da generalização tem que ser encarado de maneira indutiva, à medida que novos experimentos são conduzidos.

Este experimento foi iniciado a partir de um conjunto de 25 defeitos dos quais os 11 mais difíceis de serem revelados foram selecionados. Utilizou-se a medida do Índice de Dificuldade para evitar aqueles defeitos que quase certamente seriam revelados por qualquer conjunto de teste. No conjunto final, os defeitos selecionados possuem Índice de Dificuldade variando entre 2.87 e 25.20%.

A efetividade obtida pela aplicação do critério Mutação de Interface completo (com o conjunto de operadores definidos na seção 5.1.1) é quase 100% para todos os experimentos. Somente um dos 330 conjuntos MI-adequados não revelou o correspondente defeito. Alguns dos critérios alternativos apresentados nas Tabelas 18 e 19 também proporcionam medidas de efetividade muito boas.

Em termos de custo, como já era de se esperar, pode-se observar que a aplicação do critério com todos os operadores tende a ser cara. Não em termos absolutos, visto que os 2309 mutantes e 41,34 casos de teste em média, não são números demasiadamente altos. Mas, se comparado com os critérios alternativos, vemos que resultados de efetividade muito próximos aos obtidos pela aplicação do critério completo podem ser obtidos com um custo bastante inferior, tanto em termos do número de mutantes gerados quanto do número de casos de teste requeridos. Por exemplo, utilizando-se o critério MI-aleatório-10% obtém-se, com uma redução de aproximadamente 90% no número de mutantes e 50% no número médio de casos de teste requeridos, uma efetividade de 99,27% em revelar erros, um decréscimo de apenas 0,37%. Utilizando-se o critério MI-19-seletivo, a redução no número de mutantes é de quase 95% e no número de casos de teste é de, aproximadamente, 62% enquanto a efetividade mantém-se em 97.91% dos conjuntos de teste.

É interessante também observar que existem conjuntos de operadores de mutação que são completamente disjuntos mas que apresentam resultados semelhantes em termos de efetividade. É o caso dos conjuntos “DirVarRep” e “IndVarRep”. Os resultados para esse dois conjuntos são exatamente os mesmos, o que sugere que, apesar de disjuntos, existe uma “intersecção” no conjunto de casos de teste que esses mutantes selecionam. Intuitivamente, essa é uma indicação de que apenas um deles seria suficiente para selecionar os mesmo conjuntos de casos de teste, ou pelo menos, conjuntos com características similares.

Essas conclusões porém não significam que o critério Mutação de Interface completo não deva ser utilizado. Elas sugerem que uma estratégia incremental para sua utilização deva ser definida. Suponha-se um software que contenha alguns defeitos. Pode-se iniciar a aplicação da Mutação de Interface com um conjunto bastante restrito de operadores, esperando que algum(ns) erro(s) seja(m) revelado(s). No experimento descrito, por exemplo, utilizando-se apenas 27 mutantes produzidos pelo critério MI-24-seletivo, revelou-se um defeito em mais de 50% dos conjuntos de teste. Se um defeito é revelado, deve-se corrigir o programa e reiniciar o teste. Somente se nenhum defeito é revelado, novos mutantes devem ser gerados e mais casos de teste fornecidos. Assim, conforme mostram os experimentos, Mutação de Interface é um critério interessante (na verdade, um conjunto de critérios) pois desde as primeiras fases, com um conjunto reduzido de mutantes, diversos defeitos são revelados, com uma alta probabilidade. Além disso, pode-se estabelecer um ponto final para a aplicação do critério. o que, dependendo dos requisitos de teste e restrições de custo, pode ser a aplicação

do critério completo. Diversas seqüências diferentes de mutantes podem ser utilizadas nessa abordagem incremental. Duas seqüências óbvias são os critérios MI-N-seletivos e os MI-aleatório-N%.

Analisando as Tabelas 21 e 22 pode-se notar que para o conjunto de defeitos semeados no programa *Sort*, o critério Mutação de Interface completo produz conjuntos de teste tão grandes que conjuntos de mesma cardinalidade, randomicamente gerados, têm quase a mesma efetividade em revelar erros que os conjuntos MI-adequados. O mesmo não acontece com os critérios MI-aleatório-10% e MI-19-seletivo. Nesses casos, os conjuntos adequados são drasticamente reduzidos, de modo que conjuntos aleatórios de mesmo tamanho têm efetividade de apenas 77.91% e 70.00% respectivamente. Além disso, analisando os conjuntos com cardinalidades minimizadas, percebe-se que os critério de Mutação de Interface tendem a selecionar casos de teste que revelam erros muito mais rapidamente que através de geração aleatória. Considerando os cardinalidades minimizadas, a efetividade dos conjuntos aleatórios não atingem 50%. Assim, além da Mutação de Interface proporcionar um modo objetivo e significativo para medir a qualidade de conjuntos de teste, ela claramente ultrapassa o teste randômico em efetividade em revelar defeitos.

Outra importante informação a ser retirada dos experimentos é a respeito de quais conexões colaboram mais para que defeitos sejam revelados. Como mostra a Tabela 20, o teste de algumas conexões pode ser capaz de revelar a presença de defeitos localizados em outras funções, não diretamente relacionadas com a conexão sendo testada. Então, ao testar um programa já integrado, como o caso do *Sort*, quais conexões devem ser testadas antes? A resposta óbvia é que deve-se testar aquelas conexões que apresentem maior probabilidade de revelar a presença de um defeito, mas como identificar tais conexões? Por exemplo, o que faz a conexão *cmp-skip* melhor para iniciar o teste do que a conexão *merge-disorder*?

Quando uma mutação é aplicada a uma conexão *A-B*, um valor relacionado com essa conexão é perturbado. Para que tal mutante seja distinguido, esse valor que foi alterado deve ser usado e deve levar a uma saída diferente da saída produzida pelo programa original. Esse uso pode ocorrer em qualquer lugar do programa. Supondo que esse valor que sofreu a mutação seja usado, além possivelmente de *A* e *B*, numa determinada função *C*, todos os casos de teste selecionados para testar a conexão *A-B* devem também executar *C* pois só assim esses mutantes são distinguidos. Isso pode limitar o conjunto de caminhos percorridos pelos casos de teste selecionados. É o que ocorre, por exemplo, no teste da conexão *merge-disorder*. Os 64 mutantes gerados para essa conexão podem ser distingui-

dos, em média por 2,24 casos de teste que cobrem sempre um pequeno trecho do programa, sendo assim ineficazes para revelar defeitos que poderiam estar localizados em outras partes do código. Por outro lado, conexões como *cmp-skip* e *merge-sort* requerem um número maior de casos de teste que executam um conjunto maior de caminhos e, conseqüentemente, permitem que um número maior de defeitos possa ser executado e revelado. Até agora, parece que a melhor maneira para selecionar quais conexões devem ser testadas é analisando o código e determinando em quais delas são alterados os valores dos objetos (variáveis) principais do programa, permitindo que, ao distinguir os mutantes, possa-se exercitar grandes trechos do programa.

5.2 Estudo de Caso 2: O Programa *SPACE*

O programa *SPACE* [79] foi escolhido para este experimento pois trata-se de um sistema real, que vem sendo utilizado pela Agência Espacial Européia e que possui documentado um histórico de desenvolvimento. Por esses motivos, e por tratar-se de um programa de complexidade considerável, como mostra a Tabela 23, esse programa foi escolhido para este estudo de caso.

Tabela 23: Métricas relacionadas com o programa *SPACE*

	LOC's	Conexões
Média por função	32,95	3,29
Total (135 funções)	4449	444

O programa processa uma linguagem de alto nível que permite ao usuário descrever um vetor de antenas. Seu objetivo é produzir como saída um arquivo de dados com formato e características pré-definidas, a partir dessa descrição. A linguagem desenvolvida para esse fim permite que se descreva o vetor de antenas a partir de alguns poucos comandos, ao invés de descrever-se uma lista completa de posições e estímulos dos elementos. Essa lista é então produzida como saída do programa [79]. A cada execução, o programa solicita o nome de arquivo com extensão *.adl* que contém a descrição a ser processada e gera como saída um arquivo com mesmo nome e extensão *.dat*. Um exemplo de arquivo de entrada é dado na Figura 41. Na Figura 42 é mostrado um trecho do respectivo arquivo de saída.

O programa consiste de três partes principais:


```

GROUP  testgroup

      GRID RECTANGULAR

      ELEMENT
          GEOMETRY RECTANGULAR 20,10 cm

      ADD BLOCK  -1,-1  1,1
      ADD NODE   -2,-2
      ADD NODE   -2,2
      ADD NODE    2,2
      ADD NODE    2,-2

      GROUP_EXCITATION
          AMPLITUDE UNIFORM
          PHASE UNIFORM

END

```

Figura 41: Arquivo de entrada para programa *SPACE*

- Um parser faz a análise sintática da descrição e prepara as estruturas de dados internas descrevendo o vetor.
- Um módulo de cálculos assinala ou calcula parâmetros do vetor que foram omitidos pelo usuário na descrição, verifica a consistência dos parâmetros fornecidos na descrição e calcula a posição e estímulo de cada elemento do vetor.
- Um módulo de formatação prepara a saída no formato esperado.

Durante o processo de teste do *SPACE*, defeitos foram detectados e removidos. O processo continuou após a liberação do software, à medida que falhas foram ocorrendo. Assim, o sistema evoluiu até que uma versão “sem defeitos” foi obtida. Essa versão é utilizada como oráculo no experimento descrito neste capítulo. Durante o processo de remoção de defeitos, tanto na fase de teste como de manutenção, manteve-se um controle da evolução do produto. Foram revelados e registrados 33 defeitos, classificados por Pasquini [79] – de acordo com o padrão IEEE 1044 “Standard Classification for Software Errors, Faults and Failures” – nas categorias da Tabela 24.

A lista dos defeitos está mostrada no Apêndice A. Infelizmente, não existe na documentação fornecida [79] uma classificação por defeito. Dessa maneira, não se conseguiu relacionar precisamente cada defeito com as categorias da Tabela 24. Não é citada também a ordem em que esses defeitos foram encontrados

ARRAY PREPROCESSOR ver. 1.0

Enter source file name (omit .ADL) ?:

```
2  GROUP testgroup
4  GRID RECTANGULAR
6  ELEMENT
7  GEOMETRY RECTANGULAR 20 10 cm
9  ADD BLOCK -1 -1 1 1
10 ADD NODE -2 -2
11 ADD NODE -2 2
12 ADD NODE 2 2
13 ADD NODE 2 -2
15 GROUP_EXCITATION
16 AMPLITUDE UNIFORM
17 PHASE UNIFORM
19 END

** Generating GROUP: testgroup.
** PSTEP,QSTEP set to:200.000000,100.000000
** Grid ANGLE set to:90.000000
** Group ROT_ANGLE set to: 0.000000
```

** Writing output file: gr1.dat

NEL						
NPORTS						
IDUM	REXA AMP	REYA PSH	REZA PSC	THEA PHEPOL	PHEA	PSEA
13						
1						
1	-200.00	-100.00	0.00	0.00	0.00	0.00
	1.00	0.00	0.00	90.00		
2	0.00	-100.00	0.00	0.00	0.00	0.00
	1.00	0.00	0.00	90.00		
3	200.00	-100.00	0.00	0.00	0.00	0.00
	1.00	0.00	0.00	90.00		
	...	...	...	...	...	...
11	-400.00	200.00	0.00	0.00	0.00	0.00
	1.00	0.00	0.00	90.00		
12	400.00	200.00	0.00	0.00	0.00	0.00
	1.00	0.00	0.00	90.00		
13	400.00	-200.00	0.00	0.00	0.00	0.00
	1.00	0.00	0.00	90.00		

** End writing output file: gr1.dat

Figura 42: Arquivo de saída do programa *SPACE*

Tabela 24: Classificação de defeitos do programa *SPACE*

TIPO DE DEFEITO	SUBTIPO	#
Lógica omitida ou incorreta	Casos ou passos esquecidos	2
	Função desnecessária	1
	Condição de teste faltando	4
	Verificação de variável errada	2
Problemas computacionais	Equação insuficiente ou incorreta	10
Interface incorreta ou incompleta	Módulos não combinam	3
Problemas de manipulação de dados	Inicialização incorreta	1
	Acesso ou armazenagem incorreto	10

e removidos.

Além do histórico de defeitos, seus desenvolvedores definiram um perfil operacional para esse programa. Para isso, foram identificadas as possíveis subfunções do programa³. Gerou-se então um grafo capturando a conectividade entre essas subfunções. Cada nó do grafo representa uma subfunção. Os nós *A* e *B* são ligados por uma aresta se o controle de execução do programa pode passar da subfunção *A* para a subfunção *B*. Existe um único nó de início e um único nó de final de execução. Um caminho do nó inicial ao nó final representa uma função executada pelo programa. A cada aresta foi assinalada uma probabilidade de transição. Por exemplo, se o nó *A* é conectado com *B*, *C* e *D* com probabilidades de transição a partir de *A* de 0,3, 0,6 e 0,1 respectivamente, isso significa que após a execução da subfunção *A*, as probabilidades de executarem-se *B*, *C* e *D* são respectivamente 0,3, 0,6 e 0,1. As probabilidades de transição foram determinadas através de entrevistas com os usuários [79].

Existe ainda um programa que gera dados de teste de acordo com o perfil operacional definido. Esse programa toma como parâmetro um número inteiro e cria um dado de teste. Assim, gerando-se aleatoriamente uma sequência de números e usando-os como parâmetros para esse programa consegue-se criar um conjunto de dados de teste que representa o perfil operacional. Esse foi o método utilizado para a geração de conjuntos de dados de teste neste experimento.

³Nesse contexto, “subfunção” e “função” não estão relacionados com “função” que é uma unidade de processamento da linguagem C mas sim, ao comportamento que se espera de um trecho do programa.

5.2.1 Descrição do Experimento

O objetivo deste experimento é avaliar o critério Mutação de Interface quanto ao custo de sua aplicação e sua efetividade em revelar defeitos, tentando reproduzir os procedimentos normalmente seguidos no processo de teste de software. Uma pergunta que reflete esse objetivo é: *Qual seria a probabilidade de revelar os defeitos conhecidos do programa SPACE utilizando Mutação de Interface, e a que custo?*

Para isso, tomou-se a versão “sem defeitos” do *SPACE* – chamada de agora em diante de $SPACE_0$ – e nela foram reimplantados alguns – digamos n – dos defeitos conhecidos, criando-se uma versão $SPACE_n$ do programa. Aplicou-se nessa versão a Mutação de Interface para selecionar casos de teste. Se t , um desses casos de teste selecionados revela um erro no comportamento do programa, ou seja faz com que $SPACE_n$ tenha um comportamento diferente de $SPACE_0$, então escolhem-se e removem-se um ou mais defeitos, de maneira que a nova versão tenha o mesmo comportamento com t que tem $SPACE_0$. Em outras palavras, removem-se m defeitos, $m \geq 1$ de maneira que $SPACE_{n-m}(t) = SPACE_0(t)$. Esse processo é repetido até que não se consiga mais revelar nenhum defeito, ou porque os casos de teste selecionados não conseguem revelar os defeitos existentes ou porque não existem mais defeitos a serem revelados, ou seja, chegou-se à versão $SPACE_0$. A Figura 43 descreve esse processo.

Para selecionar os defeitos a serem implantados, criou-se um subdomínio dos casos de teste, baseado no perfil operacional do programa. Utilizando o programa de geração de casos de teste do *SPACE*, foram gerados aleatoriamente 2000 casos de teste. Com esses 2000 casos de teste foram computados os Índices de Dificuldade de cada defeito e foram selecionados aqueles defeitos com índices menores ou iguais a 10%. Note-se que o processo de “teste” do *SPACE* foi reproduzido 10 vezes para que se obtivesse um significado estatístico dos dados. Para cada uma dessas repetições – que outra vez foram chamadas de “experimento I” a “experimento X” – o subdomínio gerado foi diferente e, portanto, os defeitos selecionados nem sempre são os mesmos (embora obviamente sejam conjuntos bastante parecidos).

A seleção de casos de teste foi limitada ao subdomínio gerado, que de certa forma representa o perfil operacional do programa. A idéia é tentar responder à pergunta: *“Dado este domínio (restrito), qual a chance de, utilizando Mutação de Interface, selecionar casos de teste que revelem defeitos?”*. Note-se que como os Índices de Dificuldade foram também calculados dentro do subdomínio, sabe-

Figura 43: Processo de “teste” do programa *SPACE* utilizando Mutação de Interface

se exatamente quão difícil é selecionar casos de testes que revelem os defeitos. Assim, pode-se colher dados sobre a efetividade do critério Mutação de Interface, que nada mais é do que a probabilidade de, aplicando-se o critério, selecionarem-se esses casos de teste.

O processo de seleção de casos de teste a partir do subdomínio gerado está descrito na Figura 44. Definiu-se uma estratégia incremental de geração de mutantes através de critérios MI-alternativos, descritos adiante. Inicialmente, utiliza-se um critério que gera um número bastante restrito de mutantes. Esses mutantes são executados com os casos de teste, que são escolhidos no subdomínio numa ordem pré determinada. Se um caso de teste distingue algum mutante ele é mantido e, caso contrário, é descartado. Isso é repetido até que algum dos casos de teste que foram mantidos revele a presença de defeitos na versão do *SPACE* sendo testada ou até que se esgote o subdomínio. Note-se que são selecionados somente aqueles casos de teste que matam pelo menos um mutante. Aqueles que não matam mutantes não seriam selecionados pela Mutação de Interface e, por isso, são

descartados. Dessa maneira, podem existir casos de teste que revelem a presença de defeitos mas que são descartados por não matarem nenhum mutante.

Figura 44: Processo de seleção de casos de teste para cada experimento do programa *SPACE*

Se todo o subdomínio é utilizado na execução dos mutantes e nenhum dos casos de teste selecionados revela a presença de defeitos, então são gerados mais mutantes, aplicando-se o próximo critério MI-alternativo. Se todos os critérios dentro da estratégia estabelecida forem aplicados sem que um defeito tenha sido revelado, então considera-se terminado o teste, sem sucesso.

A sequência de critérios MI-alternativos empregada nesse processo utiliza uma combinação de três abordagens para reduzir o número de mutantes. A primeira é a seleção de apenas um pequeno número de operadores de mutação. Os operadores de mutação utilizados para caracterizar a sequência de critérios são os seguintes:

1. Operadores do Grupo II.
2. Operadores de troca de variáveis de interface (operadores do tipo “DirVar-Rep”).

3. Operadores de incremento/decremento de variáveis de interface (operador “DirVarIncDec”).
4. Operadores de inclusão de operadores unários em ocorrências de variáveis de interface (operadores do tipo “DirVarxxxNeg”).
5. Operadores de comandos de retorno (operadores “RetSta”).
6. Operadores de troca de variáveis não de interface (operadores do tipo “IndVarRep”).
7. Operadores de incremento/decremento de variáveis não de interface (operador “IndVarIncDec”).
8. Operadores de inclusão de operadores unários em ocorrências de variáveis de interface (operadores do tipo “IndVarxxxNeg”).

A segunda é a limitação do número máximo de operadores por ponto de mutação. Para os critérios de 1 a 5 estabeleceu-se como 2 o número máximo de mutantes por ponto de mutação. Para os demais, que utilizam operadores que alteram variáveis de interface de maneira indireta, ou seja, são aplicados em variáveis locais e constantes, esse número foi limitado a, no máximo, 1 mutante. E a terceira forma de reduzir o número de mutantes foi a seleção aleatória de apenas 10% dos mutantes. A combinação dessas duas abordagens foi feita primeiro limitando-se o número por ponto de aplicação e dentre esses mutantes selecionando-se 10%.

Além de uma estratégia incremental, algumas conexões não foram testadas. Os operadores de mutação foram aplicados a todas as conexões do programa, exceto às conexões das funções 1) main-glvainit; 2) glvainit-kwdsinit; 3) glvainit-emsginit; 4) glvainit-unaminit; e 5) glvainit-moseinit. Essas conexões são executadas uma única vez em cada execução do programa, sendo que as funções envolvidas são responsáveis unicamente pela inicialização de tabelas estáticas. Por esse motivo decidiu-se não aplicar mutações nessas conexões e supor que defeitos na inicialização dessas tabelas pudessem ser revelados através do teste de outras conexões, relacionadas a possíveis usos dessas tabelas. O número de mutantes relativo a essas conexões e que deixaram de ser gerados é de 77.789 mutantes.

Para as conexões que sobraram, obteve-se uma diminuição de aproximadamente 94% no número de mutantes, através do uso dos critérios MI-alternativos. Para a versão sem defeitos do *SPACE*, o número total de mutantes possíveis de serem gerados para as mesmas conexões com o mesmo conjunto de operadores

é de 91352. Esse número cai para 5129 com a aplicação das restrições impostas pelas parametrizações descritas. O número de mutantes gerados para a versão *SPACE*₀, por operador, é mostrado na Tabela 25.

Tabela 25: Número de mutantes gerados, por operador, para a versão *SPACE*₀

Operador	Mutantes	Operador	Mutantes
I-DirVarAriNeg	93	I-DirVarBitNeg	70
I-DirVarIncDec	700	I-DirVarLogNeg	77
I-DirVarRepCon	217	I-DirVarRepExt	101
I-DirVarRepGlo	4	I-DirVarRepLoc	341
I-DirVarRepPar	240	I-DirVarRepReq	165
I-IndVarAriNeg	248	I-IndVarBitNeg	229
I-IndVarIncDec	484	I-IndVarLogNeg	222
I-IndVarRepCon	278	I-IndVarRepLoc	293
I-IndVarRepPar	484	I-IndVarRepReq	250
I-RetStaDel	91	I-RetStaRep	95
II-ArgAriNeg	22	II-ArgBitNeg	13
II-ArgIncDec	270	II-ArgLogNeg	22
II-ArgRepReq	28	II-ArgStcAli	20
II-FunCalDel	72	TOTAL	5129

A Tabela 26 mostra, em ordem crescente de HI, as características dos defeitos selecionados para o experimento I, que como já citado anteriormente, depende do subdomínio gerado para esse experimento. A Tabela 27 mostra os resultados obtidos com a aplicação da Mutação de Interface no experimento I.

Tabela 26: Defeitos implantados no experimento I

DEFEITOS	33	12	18	22	27	8	17	7	
HI	0,05	0,15	0,15	0,25	0,4	1,4	1,65	1,75	
DEFEITOS	20	21	23	16	24	29	13	11	Média
HI	2,55	2,55	2,55	4,85	5,6	6,25	6,65	9,9	2,92

A Tabela 27 mostra a evolução do teste para o experimento I. A primeira coluna indica o número de defeitos de cada uma das fases do teste. A coluna dois indica o número de mutantes que foram gerados até que algum caso de teste selecionado tenha revelado um defeito ou até que não se tenha conseguido mais revelar defeitos. A terceira coluna indica o número dos defeitos que foram escolhidos para remoção para criar-se a versão do programa para a próxima fase. A quarta coluna indica o número de casos de teste que foram selecionados no

Tabela 27: Resultados da aplicação da Mutação de Interface no experimento I

Número de defeitos	Mutantes gerados	Defeitos removidos	Casos de teste	Escore de mutação
16	431	23, 16, 24	1	0,1787
13	427	13	7	0,2998
12	436	8	8	0,3050
11	436	7, 11	12	0,3899
9	437	29	13	0,4005
8	437	27	15	0,4462
7	437	20, 21	27	0,5515
5	437	12, 18	43	0,6293
3	435	33, 22	50	0,6759
1	2215	17	61	0,4790
0	5129	–	85	0,3931

teste e a última coluna indica o escore de mutação obtido por esses casos de teste em relação aos mutantes gerados. Iniciou-se com 16 defeitos implantados e 431 mutantes. Foi necessário selecionar-se apenas 1 caso de teste para que um defeito fosse revelado. Para que o programa se comportasse de acordo com o oráculo, foi então necessário remover os defeitos números 23, 16 e 24 simultaneamente. O escore de mutação obtido por esse único caso de teste atingiu 0,1787. A segunda linha mostra os números do teste aplicado à nova versão sem os 3 defeitos removidos, ou seja, com 13 defeitos restando. Foram 427 mutantes, 7 casos de teste, escore de 0,2998 e defeito 13 removido.

Nota-se na Tabela 27 que em alguns casos são removidos mais do que um defeito ao final de uma fase. Além disso, existe a possibilidade de que, quando dois ou mais defeitos estão relacionados entre si, a remoção de qualquer um deles faça com que a nova versão se comporte de maneira correta, fazendo com que qualquer um deles seja candidato à remoção. O processo de remoção de defeitos é mais precisamente descrito da seguinte maneira, supondo-se que o caso de teste t causou uma falha na versão $SPACE_n$:

1. Retira-se da versão $SPACE_n$, o defeito que possui o mais alto HI. Se a nova versão não se comporta de acordo com o oráculo $SPACE_0$, reimplanta-se esse defeito e retira-se o próximo, com o segundo maior HI. Esse processo é repetido até que a nova versão $SPACE_{n-1}$ comporte-se de acordo, ou até que todos os defeitos implantados tenham sido tentados. Dessa maneira, a abordagem é tentar eliminar primeiro os defeitos mais “fáceis”, ou seja, com maior HI.

2. Se nenhum defeito pôde ser removido, então são removidos simultaneamente todos aqueles defeitos para o qual t provoca uma falha, ou seja, aqueles defeitos que, se implantados sozinhos no *SPACE*, criam uma versão que falha com t . Essa informação é disponível pois após a geração do subdomínio de teste, todos os casos de teste são executados com cada versão do *SPACE* com um único defeito para medirem-se os HI's de cada defeito.
3. Se essas tentativas de remover defeitos falharem, então deve-se fazê-la manualmente. Em nenhum dos experimentos foi necessária a remoção manual.

A Tabela 28 mostra um resumo dos dados de efetividade e custo de todos os experimentos. Nela são apresentados o número de defeitos inicialmente implantados; os HI's mínimo e máximo e a média dos HI's dos defeitos implantados; quais os defeitos que não foram revelados e a média de mutantes e de casos de teste requeridos nas diversas fases de cada experimento.

Tabela 28: Sumário dos resultados de efetividade e custo

Exp.	Defeitos implantados	Índice de Dificuldade			Def. não revelados	Mutantes	Casos de teste
		Min.	Max.	Média			
I	16	0.05	9,9	2.92	-	1023,36	29.27
II	17	0,1	9,8	3,79	-	1120,73	34.45
III	16	0.05	9,55	2.99	33	974,33	34.17
IV	14	0.1	6,35	2.36	-	1189,60	40.20
V	17	0.15	9.0	2.94	-	800,38	26.0
VI	17	0.05	9.9	3.17	33	934,92	28.85
VII	16	0.05	8,5	2.75	17. 7	916,0	34.33
VIII	15	0.15	9,65	3.19	-	1433.10	40.0
IX	15	0.2	8,9	2.89	-	905,10	30,80
X	16	0.05	9.55	2.76	-	1122.82	30.91

Para que se possa analisar cada experimento com mais detalhes, as Tabelas 29 a 46 mostram os defeitos implantados e os resultados obtidos em cada um dos experimentos, de II a X. Na coluna “Defeitos Removidos”, quando o número do defeito vier entre parêntesis, isso indica o número do defeito que não pôde ser removido no final do experimento.

Tabela 29: Defeitos implantados no experimento II

DEFEITOS	12	18	22	27	8	7	17	20	21
HI	0,1	0,1	0,25	0,25	0,8	1,8	2,0	2,55	2,55
DEFEITOS	23	13	16	29	24	11	31	10	Média
HI	3,00	5,5	5,5	5,5	6,45	8,65	9,55	9,8	3,79

Tabela 30: Resultados da aplicação da Mutação de Interface no experimento II

Número de defeitos	Mutantes gerados	Defeitos removidos	Casos de teste	Escore de mutação
17	439	8, 7, 23	4	0,2187
14	439	11	11	0,3303
13	431	22, 24	15	0,4640
11	433	29	17	0,4111
10	433	10	18	0,4273
9	441	17	33	0,5374
8	441	20, 21, 16	37	0,5488
5	1491	13, 31	49	0,4071
3	437	12, 18	44	0,6347
1	2214	27	64	0,4770
0	5129	-	87	0,3881

Tabela 31: Defeitos implantados no experimento III

DEFEITOS	33	27	12	18	22	8	20	21	
HI	0,05	0,1	0,15	0,15	0,2	1,25	1,9	1,9	
DEFEITOS	7	17	23	16	29	13	24	11	Média
HI	2,1	2,15	3,6	5,05	6,2	6,6	6,85	9,55	2,99

Tabela 32: Resultados da aplicação da Mutação de Interface no experimento III

Número de defeitos	Mutantes gerados	Defeitos removidos	Casos de teste	Escore de mutação
16	431	13	1	0,1740
15	433	29	4	0.2079
14	433	11	6	0,2702
13	432	24	16	0,3819
12	437	20, 21, 16	19	0,4691
9	437	8, 23	33	0,5378
7	437	7	36	0,5744
6	437	17	39	0,5995
5	437	12, 18	43	0.6201
3	435	22	48	0,6575
2	2214	27	69	0.4883
1	5129	(33)	96	0.3997

Tabela 33: Defeitos implantados no experimento IV

DEFEITOS	27	12	18	22	8	7	20	
HI	0,1	0.2	0,2	0, 2	1.25	1,35	1,85	
DEFEITOS	21	17	23	16	24	13	29	Média
HI	1,85	2.1	2,25	3,8	5,5	6,0	6,35	2,36

Tabela 34: Resultados da aplicação da Mutação de Interface no experimento IV

Número de defeitos	Mutantes gerados	Defeitos removidos	Casos de teste	Escore de mutação
14	430	17	5	0.2372
13	430	29	9	0.2814
12	430	7, 23, 16, 24	11	0.3907
8	437	13	23	0.5240
7	437	20, 21	32	0.5835
5	437	22	40	0.6316
4	437	12, 18	48	0.6270
2	1515	8	66	0.4607
1	2214	27	72	0.4865
0	5129	-	96	0.3964

Tabela 35: Defeitos implantados no experimento V

DEFEITOS	27	12	18	22	8	26	7	17	21
HI	0,15	0,25	0,25	0,5	1,1	1,65	1,65	1,75	2,15
DEFEITOS	20	23	10	16	29	13	24	11	Média
HI	2,2	2,65	3,95	4,65	5,3	6,05	6,65	9,0	2,94

Tabela 36: Resultados da aplicação da Mutação de Interface no experimento V

Número de defeitos	Mutantes gerados	Defeitos removidos	Casos de teste	Escore de mutação
17	439	7, 17, 23	1	0,1617
14	439	22, 11	6	0,4146
12	439	29	7	0,3394
11	439	24	8	0,3440
10	433	10	9	0,3626
9	441	26	15	0,4603
8	441	13	17	0,4535
7	437	27	17	0,4554
6	437	16	15	0,4371
5	437	12, 18	44	0,6156
3	447	21, 20	46	0,6264
1	447	8	55	0,6577
0	5129	–	98	0,4042

Tabela 37: Defeitos implantados no experimento VI

DEFEITOS	27	33	12	18	22	8	20	21	7
HI	0,05	0,05	0,1	0,1	0,6	1,15	1,3	1,35	1,45
DEFEITOS	17	23	16	29	13	24	11	10	Média
HI	1,65	2,85	4,4	6,15	6,95	7,15	8,75	9,9	3,17

Tabela 38: Resultados da aplicação da Mutação de Interface no experimento VI

Número de defeitos	Mutantes gerados	Defeitos removidos	Casos de teste	Escore de Mutação
17	439	13	2	0,2255
16	434	24	4	0,2512
15	442	17	9	0,2986
14	442	11	10	0,3213
13	436	7, 23	10	0,3395
11	436	10	13	0,3784
10	437	29	15	0,4005
9	437	8	26	0,4508
8	435	20, 21, 16	31	0,5149
5	437	22	39	0,6270
4	437	12, 18	44	0,6087
2	2214	27	75	0,4846
1	5129	(33)	97	0,3944

Tabela 39: Defeitos implantados no experimento VII

DEFEITOS	33	12	18	27	22	8	17	7	
HI	0,05	0,3	0,3	0,3	0,5	0,7	1,45	1,55	
DEFEITOS	20	21	23	16	13	29	24	11	Média
HI	1.65	1.65	2.95	5.0	5.95	6.25	6.9	8.5	2.75

Tabela 40: Resultados da aplicação da Mutação de Interface no experimento VII

Número de defeitos	Mutantes gerados	Defeitos removidos	Casos de teste	Escore de mutação
16	431	29	2	0.1671
15	431	24	4	0.2575
14	427	11	10	0.3489
13	437	27	16	0.3570
12	437	16	4	0.2609
11	437	13	27	0.4874
10	437	8, 23	33	0.5057
8	437	20, 21	38	0.5286
6	437	22	50	0.6362
5	437	12, 18	63	0.6568
3	1515	33	72	0.4673
2	5129	(17,7)	93	0.4022

Tabela 41: Defeitos implantados no experimento VIII

DEFEITOS	12	18	27	22	8	17	20	21
HI	0,15	0,15	0,15	0,5	1,35	1,75	1,75	1,75
DEFEITOS	7	23	16	29	13	24	11	Média
HI	1,75	3,35	4,8	6,35	7,1	7,35	9,65	3,19

Tabela 42: Resultados da aplicação da Mutação de Interface no experimento VIII

Número de defeitos	Mutantes gerados	Defeitos removidos	Casos de teste	Escore de mutação
15	431	13	1	0,1508
14	433	29	7	0,2771
13	433	7, 11	15	0,3903
11	432	24	27	0,4815
10	437	8, 23	30	0,4851
8	437	20, 21, 16	32	0,5034
5	437	12, 18	45	0,5858
3	435	22	50	0,6460
2	2214	27	67	0,4765
1	3946	17	80	0,4098
0	5129	-	86	0,3880

Tabela 43: Defeitos implantados no experimento IX

DEFEITOS	12	18	27	22	8	7	17	20
HI	0,2	0,2	0,2	0,25	0,8	1,25	1,3	1,85
DEFEITOS	21	23	16	24	13	29	11	Média
HI	1,85	2,15	4,7	6,05	6,4	7,25	8,9	2,89

Tabela 44: Resultados da aplicação da Mutação de Interface no experimento IX

Número de defeitos	Mutantes gerados	Defeitos removidos	Casos de teste	Escore de mutação
15	431	13, 11	2	0,1833
13	432	23 16 24	3	0,2362
10	437	29	6	0,3272
9	437	22, 7	23	0,5858
7	437	20 21	27	0,5492
5	437	27	31	0,5767
4	437	8	35	0,5995
3	437	17	45	0,6293
2	437	12, 18	47	0,6362
0	5129	–	89	0,3972

Tabela 45: Defeitos implantados no experimento X

DEFEITOS	33	12	18	22	27	8	7	17	
HI	0,05	0,25	0,25	0,25	0,3	0,65	1,05	1,5	
DEFEITOS	20	21	23	16	24	29	13	11	Média
HI	1,9	1,9	2,45	4,4	6,35	6,65	6,65	9,55	2,76

Tabela 46: Resultados da aplicação da Mutação de Interface no experimento X

Número de defeitos	Mutantes gerados	Defeitos removidos	Casos de teste	Escore de mutação
16	431	11	6	0.2784
15	430	20. 21, 16	7	0.3023
12	430	7, 23, 29	10	0.3581
9	430	22. 24	13	0.4884
7	441	13	15	0.4830
6	437	12. 18	23	0.5378
4	447	8	25	0.5526
3	447	17	42	0.6175
2	1515	33	55	0.4548
1	2214	27	63	0.4788
0	5129	–	81	0.3890

5.2.2 Análise dos Dados

A exemplo do experimento com o programa *Sort*, podem ser considerados encorajadores os resultados obtidos neste experimento. Foram implantados no programa *SPACE*, 159 defeitos dos quais apenas 4 não puderam ser revelados. Na média de cada experimento, a probabilidade de que esses defeitos sejam revelados – expressos por seus Índices de Dificuldade – varia de 2,36% a 3,79%. Para os defeitos não revelados, as probabilidades de selecionar casos de teste que demonstrem sua presença são de apenas 0,05%, 0,05%, 1,45% e 1,55%. Em outras palavras, os resultados do estudo indicam que a utilização da Mutação de Interface tende a selecionar casos de teste que revelem defeitos quando eles existem.

Analisando individualmente cada experimento nota-se que em 3 deles não foram revelados todos os defeitos implantados. No experimento III e no experimento VI, o defeito número 33 não foi revelado. Nesses casos, apenas um caso de teste de cada subdomínio seria capaz de revelar o defeito mas eles não foram selecionados pela aplicação da Mutação de Interface. Porém, em outros experimentos como I, VII, e X, o mesmo defeito obteve também o mesmo Índice de Dificuldade de 0,05% e foi revelado. Este fato pode ser relacionado com a maneira com que os casos de teste são selecionados dentro do subdomínio de cada experimento. Os casos de teste do subdomínio possuem números de 1 a 2000 e são utilizados nessa ordem. Assim, se um caso de teste que revele o defeito está no “início” do subdomínio, ele possui uma chance maior de ser selecionado pois o número de mutantes vivos com os quais o caso de teste vai ser experimentado é mais alto. Se ele estiver no “final” do domínio, a chance de ser selecionado é menor pois o conjunto de mutantes que seriam distinguidos por ele pode já ter sido distinguido por outros casos de teste. Nos experimentos III e VI, os casos de teste que revelam o defeito 33 estavam nas posições 1470 e 1276 dos subdomínios.

O mesmo acontece com os defeitos 17 e 7 que não foram revelados no experimento VII mas foram revelados em todos os demais experimentos em que foram implantados. Nesse caso porém, o número de casos de teste que revelam os defeitos é bem mais alto, ou seja, 29 e 31 respectivamente. Isso significa que ao tentar o caso de teste 54 - primeiro a revelar o defeito 17 - todos os mutantes que esse caso de teste poderia distinguir já foram distinguidos pelos casos de teste de 1 a 53. O mesmo acontece para os demais casos de teste que poderiam revelar algum dos defeitos. A Tabela 47 mostra quais são os casos de teste que revelam cada um dos dois defeitos e quais foram os casos de teste selecionados pela Mutação de Interface com o conjunto completo de 5129 mutantes. Apenas um caso de

teste revela ambos os defeitos. Assim, tem-se um conjunto de 59 casos de teste que poderiam revelar pelo menos um dos defeitos. Surpreendentemente, nenhum desses casos de teste foi selecionado pela aplicação da Mutação de Interface.

Tabela 47: Casos de teste selecionados e casos de teste que revelam defeitos 7 e 17 no experimento VII

Casos selecionados	Casos que revelam defeito 7	Casos que revelam defeito 17
1 2 4 5 6 7 8 9 10 11 12	226 339 397	54 82 340
14 15 17 20 21 22 29 30 33	458 474 512	453 464 581
40 44 45 50 55 56 58 65 68	759 760 790	616 671 759
81 85 86 97 98 100 107 108	828 915 968	864 903 907
123 126 133 134 165 177 179	1041 1132 1207	927 938 961
184 215 218 224 230 277 299	1283 1316 1332	1060 1098 1191
319 329 359 435 477 489 493	1341 1342 1380	1213 1324 1494
524 533 614 660 808 839	1522 1658 1663	1527 1545 1584
1218 1540 52 113 274 365	1715 1718 1736	1589 1613 1810
609 881 1369 1724 187 311	1869 1877 1880	1819 1854
1329 1889 1617 697 3 64 78	1890	
120 349 536 653 1560 1949		
110 1428 144 939		

A observação de que a ordem de seleção dos casos de teste pode influenciar na efetividade do critério sugere que talvez deva-se tentar estabelecer maneiras para aumentar a probabilidade de selecionar casos de teste que revelem defeitos. Como foi dito no Capítulo 2, cada mutante determina uma partição do domínio de entrada, formada pelos casos de teste que distinguem esses mutantes. Um caso de teste que revele erro nessa partição tem uma certa probabilidade de ser selecionado. Se ele é o único na partição, então essa probabilidade é 1. Se existem outros casos de teste na partição, então a chance é dividida entre outros que podem ou não revelar o defeito. Uma forma de tentar aumentar a chance de selecionar casos de teste que revelem defeitos seria determinar para os mutantes um “grau de resistência”, ou seja, determinar que para matar um mutante este deve ser distinguido por n casos de teste, $n \geq 1$. Isso faria com que mais do que um caso de teste fosse selecionado em cada partição. Essa estratégia aumenta o número de execuções de mutantes necessárias mas não deve aumentar muito o número de casos de teste requeridos. Considerem-se dois mutantes A e B e dois casos de teste t_1 que distingue A e t_2 que distingue B . Isso significa que A foi executado com t_1 e como morreu não foi executado com t_2 e B foi executado com t_1 , não morreu mas morreu com t_2 . Estabelecendo um grau de resistência de 2 para esses

mutantes, pode ser que t_2 também distinga A , sendo necessário só mais um caso de teste para distinguir B pela segunda vez. De um modo geral, na melhor das hipóteses, apenas um caso de teste extra seria necessário para matar os mutantes com grau de resistência 2. E no pior caso – que assim como o melhor caso é bastante improvável – seria requerido mais um caso de teste para cada mutante.

Outra idéia seria estabelecer um “grau de estresse” para cada caso de teste. Ou seja, após distinguir um número n de mutantes, o caso de teste deixa de ser usado, exigindo que novos casos de teste sejam fornecidos para matar os mutantes restantes. Nesse caso, o custo da execução dos mutantes permanece o mesmo porém o número de casos de teste deve aumentar sensivelmente. O que se procura com essa abordagem é evitar que um mesmo caso de teste distinga um número muito grande de mutantes e por isso seja “selecionado” um número muito grande de vezes porque foi utilizado antes. Dá-se assim, chance para que outros casos de teste – possivelmente reveladores de defeitos – sejam também selecionados. Essas duas abordagens podem ser úteis, principalmente se utiliza-se geração randômica de casos de teste para tentar distinguir os mutantes.

Em termos de custo, pode-se notar que com um número bastante reduzido de mutantes – aproximadamente 5100 mutantes para um programa com 4500 linhas de código – obtiveram-se bons resultados em termos de efetividade. Como já observado em experimentos anteriores, muitos dos defeitos podem ser revelados com um número bastante reduzido de mutantes. No caso do *SPACE*, notou-se que dos 159 defeitos implantados, 89,31% (142 defeitos) foram revelados pelo conjunto inicial de mutantes, gerados somente pelos operadores do Grupo II e que somam, no máximo, 447 mutantes. Em seguida, gerando-se até 1515 mutantes foram revelados mais 5 defeitos, até 2215 mutantes, mais 8 defeitos, e até 3946 mutantes, mais 1 defeito. Entre 3946 e 5129 mutantes, nenhum defeito a mais foi revelado e sobraram 4 não revelados, conforme mostra a Tabela 48. Isso, por um lado, demonstra certa eficiência na estratégia incremental utilizada pois permite que muitos dos defeitos – principalmente os mais fáceis – sejam revelados rapidamente e a um custo muito baixo. Além disso, com a geração de mais mutantes, obtém-se sucesso em revelar mais defeitos, na maioria dos casos. Por outro lado, nota-se que no extremo superior, em que o número de mutantes situa-se entre 3946 e 5129, nenhum defeito foi revelado e sobraram alguns defeitos não revelados. Em outras palavras, essa última etapa da estratégia foi para esse experimento totalmente ineficaz.

Tabela 48: Número de defeitos revelados X custo dos critérios

Número de mutantes gerados	Defeitos revelados
447	142
1515	5
2215	8
3946	1
5129	0

5.3 Considerações Finais

Neste capítulo mostraram-se os resultados da aplicação do critério Mutação de Interface em dois programas selecionados como estudos de caso. No primeiro, com o programa *Sort*, utilizou-se a abordagem de incluir artificialmente defeitos no programa, um de cada vez, e aplicar a Mutação de Interface sobre as versões do programa com os defeitos. A capacidade de revelar esses defeitos bem como o custo de sua aplicação puderam ser medidos e mostraram-se animadores, sugerindo a factibilidade de utilização do critério.

Nesse contexto foram também explorados diversos critérios alternativos, utilizando-se diferentes subconjuntos de operadores de mutação. Também para esses critérios, custo e efetividade apontam de maneira positiva para a sua possível utilização em aplicações práticas. Comparando-se esses critérios, principalmente alguns critérios restritos como MI-aleatório-10% e MI-19-seletivo, com o teste randômico, nota-se que os critérios Mutação de Interface são significativamente superiores em termos de seleção de casos de teste que revelem defeitos.

No segundo experimento procurou-se dar um passo à frente em termos de estudos empíricos que utilizam a análise de mutante, selecionando-se um programa de porte sensivelmente superior àqueles que costumeiramente são usados. No programa *SPACE* foram reimplantados simultaneamente vários dos defeitos que foram localizados e removidos durante seu desenvolvimento e aplicou-se a Mutação de Interface para avaliar quantos desses defeitos poderiam ser revelados. Os resultados confirmam a idéia de que a Mutação de Interface é um bom critério de adequação interprocedimental em termos de efetividade. Dos 159 defeitos implantados no programa nas 10 repetições do experimento, apenas 4 não foram revelados por conjuntos MI-adequados. Além disso, nenhum dos defeitos deixou de ser revelado por algum conjunto MI-adequado.

O experimento com o programa *SPACE* mostrou também que utilizando

critérios MI-alternativos dentro de uma estratégia incremental de teste pode-se reduzir sensivelmente o custo do teste. Com um número bastante restrito de mutantes é possível revelar a maioria dos defeitos existentes num programa. Em seguida, são utilizados critérios mais “fortes” com o intuito de revelar aqueles defeitos mais difíceis, até um ponto determinado por restrições de custo. Neste experimento utilizou-se uma seqüência de critérios MI-alternativos baseada em algumas características dos operadores de mutação apontadas no Capítulo 3. Porém, até este ponto não é clara a melhor seqüência de operadores de mutação que deve ser empregada, muito por desconhecer-se precisamente quais as características, na prática, de cada operador de mutação. Assim, novos trabalhos devem ser conduzidos, procurando identificar um conjunto essencial de operadores de Mutação de Interface, estabelecendo estratégias para sua utilização e, possivelmente, modificando os operadores existentes e criando operadores novos.

Capítulo 6

Conclusão

Um dos pontos fundamentais na atividade de teste é a necessidade de critérios de adequação que auxiliem no projeto de casos de teste fornecendo indicação de sua qualidade. Critérios funcionais, estruturais e baseados em defeitos devem, sempre que possível, ser utilizados em conjunto, aumentando as chances de que possíveis defeitos possam ser revelados.

Na fase do teste de unidade, quando o objetivo é validar separadamente cada uma das unidades do software, tanto critérios funcionais quanto estruturais e baseados em defeitos têm sido largamente aplicados. Já na fase de teste de integração, quando a atenção volta-se para a localização de possíveis erros decorrentes da interação entre as unidades, quase que exclusivamente tem-se utilizado critérios funcionais. Em princípio, também critérios estruturais e baseados em defeitos poderiam ser utilizados nesta fase. Porém, tal aplicação tende a ser ineficiente, devido ao tamanho dos programas que quando integrados não estão restritos a simples unidades, redundante, pois diferentes equipes de testadores poderiam estar testando a mesma parte do software e incompleta, pois programas grandes não permitem que esses critérios sejam aplicados exaustivamente. Além disso, os requisitos estabelecidos pelos critérios estruturais e baseados em defeitos correntemente utilizados são restritos ao escopo de unidades, não garantindo portanto que as interações entre as unidades, que devem ser testadas nessa fase, sejam exercitadas a contento.

Este trabalho apresentou, como opção para o teste de integração, um critério

interprocedimental baseado em mutações. Esse critério, chamado Mutaço de Interface exercita as interaes entre as unidades atravs de mutaes aplicadas em pontos do software que so diretamente relacionados com essas interaes. Basicamente, dadas duas unidades que so diretamente conectadas atravs de chamadas de uma para outra, so aplicadas mutaes nos pontos relacionados com essa conexo de modo que valores trocados entre essas unidades sejam perturbados. Dessa maneira, um mutante caracteriza um erro na interao entre as unidades, ou utilizando o termo empregado nesta tese, cada mutante caracteriza um possvel erro de integrao. Dessa forma, conjuntos de teste que distingam tais mutantes, devem exercitar de maneira eficiente as interaes entre as unidades, revelando outros possveis erros de integrao.

Um dos pontos importantes na utilizao de critrios baseados em mutaes estabelecer quais alteraes devem ser aplicadas no programa em teste para a criao dos mutantes, ou seja, determinar um conjunto de operadores de mutao que, em ltima anlise, caracteriza o critrio em si. Baseado em experincias anteriores, definiu-se um conjunto de 33 operadores de Mutao de Interface, divididos em dois grupos. O primeiro composto por operadores que efetuam alteraes dentro da unidade que chamada, naqueles pontos que so relacionados com a interface da unidade, por exemplo, parmetros formais e variveis globais. O segundo grupo aplica mutaes no ponto onde a unidade que faz as chamadas, chama a outra unidade a ela conectada, por exemplo, nos argumentos dessas chamadas. Note-se que dessa maneira um mutante est sempre relacionado a uma conexo entre duas unidades e no a uma unidade.

Um ponto importante desse conjunto de operadores de Mutao de Interface que eles so parametrizveis de vrias maneiras, permitindo a utilizao de critrios de Mutao de Interface alternativos, como MI-aleatria e MI-restrita, favorecendo assim o estabelecimento de estratgias incrementais de aplicao do critrio, de acordo com requisitos de teste e restries de custo.

Utilizando esse conjunto de operadores mostrou-se teoricamente que no existe uma relao de incluso entre a Mutao de Interface e o critrio estrutural interprocedimental baseado em fluxo de dados definido por Harrold e Soffa [32]. Em outras palavras, mostrou-se que nem o critrio Mutao de Interface inclui o critrio de Harrold e Soffa nem este inclui a Mutao de Interface. Mostrou-se tambm que o nmero de mutantes, mesmo para conexes entre unidades simples, pode ser bastante elevado, sendo influenciado, principalmente, pelo nmero de variveis do programa vezes o nmero de referncias s variveis na unidade chamada e pelo nmero de parmetros formais da unidade chamada.

É essencial que a aplicação de qualquer critério de teste seja feita de maneira automatizada, elevando a produtividade do teste e evitando erros provenientes da intervenção humana. Para a aplicação do critério Mutação de Interface desenvolveu-se a ferramenta *PROTEUM/IM* que apóia a aplicação do critério. Foram discutidos nesta tese os principais aspectos da funcionalidade da ferramenta, descrevendo-se as operações que devem ser realizadas na condução de uma sessão de teste, tanto através de uma interface gráfica, – indicada para testadores inexperientes e para atividades de ensino – quanto através de scripts de teste – que permite que testadores experientes desenvolvam sessões de teste de maneira mais eficiente, evitando constante interação com a ferramenta. Aspectos da implementação foram também apresentados, dando-se ênfase às abordagens utilizadas na geração e execução dos mutantes, que são pontos críticos no teste baseado em mutações.

A disponibilidade da ferramenta permitiu que se conduzissem estudos empíricos onde avaliou-se a aplicação do critério quanto à efetividade em revelar erros e quanto ao custo. O primeiro experimento foi desenvolvido utilizando-se o *Sort*, programa de ordenação, utilitário do sistema operacional UNIX. Nesse programa foram semeados artificialmente 11 defeitos, um de cada vez, criando-se 11 versões incorretas do programa. Nessas versões foi aplicada a Mutação de Interface, criando-se 30 conjuntos MI-adequados para cada versão. Avaliou-se a efetividade do critério através do número desses conjuntos capazes de revelar o defeito semeado. O custo de aplicação foi avaliado pelo número de mutantes gerados, bem como pelo número de casos de teste nos conjuntos MI-adequados. Nesse mesmo experimento foram utilizados os critérios MI-alternativos, procurando reduzir o número de mutantes. Os resultados apontam – tanto para o critério Mutação de Interface completo como para os critérios MI-alternativos – uma alta efetividade em revelar os defeitos. Percebeu-se também que critérios alternativos podem obter efetividade bastante próxima do critério Mutação de Interface completo com um custo de aplicação bastante reduzido, a exemplo do que mostram experimentos conduzidos para o teste de mutação no nível de unidade. Ainda no experimento do *Sort*, comparou-se a efetividade da Mutação de Interface com o teste randômico, podendo concluir-se que o critério proposto nesta tese apresenta-se sensivelmente mais efetivo que o teste randômico. O mesmo acontece para os critérios MI-alternativos.

O segundo experimento para avaliar o critério Mutação de Interface foi realizado utilizando-se um programa chamado *SPACE*. Esse programa foi desenvolvido para a Agência Espacial Européia para a descrição e cálculo de antenas. Durante

o processo de desenvolvimento desse programa foram encontrados vários defeitos. No experimento aplicou-se Mutação de Interface no programa com defeitos, avaliando-se sua efetividade em revelá-los. Iniciou-se o teste com vários defeitos e aplicou-se Mutação de Interface para selecionar casos de teste. Se um dos casos de teste selecionados revela algum defeito, “corrige-se” o programa retirando um ou mais defeitos. A nova versão é novamente testada, até que não se consiga mais através da Mutação de Interface revelar um defeito. Isso pode acontecer ou porque não existem mais defeitos a serem revelados ou porque nenhum caso de teste selecionado foi capaz de revelar um defeito. Esse processo foi repetido 10 vezes. Os resultados mostram que dos 156 defeitos implantados nas 10 repetições, apenas 4 não puderam ser revelados, indicando uma efetividade em revelar defeitos bastante alta.

Mostrou-se também como utilizar critérios MI-alternativos para determinar uma estratégia incremental de teste. Com um conjunto de mutantes inicial bastante reduzido – aproximadamente 400 mutantes – foi revelada grande parte dos defeitos. Em seguida, utilizando-se mais operadores de mutação foram construídos conjuntos de teste mais completos que revelaram a maioria dos defeitos. O custo de aplicação, com o conjunto de operadores utilizado para esse experimento e as parametrizações que permitiram uma sensível diminuição no número de mutantes, pode ser considerado baixo. O número máximo de mutantes gerados durante o teste do *SPACE* foi 5129 mutantes, um número baixo, para um programa de aproximadamente 4500 linhas de código.

6.1 Contribuições deste Trabalho

Pode-se destacar como contribuições deste trabalho:

- Definição, implementação e aplicação de um critério de adequação de casos de teste interprocedimental baseado em mutações.
- Definição de um modelo de falhas que caracteriza erros de integração de software.
- Definição de um conjunto de operadores para Mutação de Interface.
- Análise teórica da complexidade do critério e da relação de inclusão com o critério interprocedimental de fluxo de dados definido por Harrold e Soffa.

- Definição de critérios de Mutação de Interface alternativos que visam a diminuir o número de mutantes gerados e favorecem o estabelecimento de estratégias de teste incrementais, permitindo ajustar a aplicação do critério a restrições de custo.
- Especificação e implementação da ferramenta de teste *PROTEUM/IM* que apóia a aplicação do critério Mutação de Interface, bem como dos critérios MI-alternativos.
- Aplicação experimental da Mutação de Interface em dois estudos de caso com o objetivo de avaliar custo e efetividade do critério.

6.2 Trabalhos Futuros

Pode-se destacar algumas atividades que devem ser realizadas como continuidade deste trabalho:

- Estender os estudos empíricos para novos estudos de caso buscando ampliar a abrangência dos resultados obtidos nesta tese.
- Determinar um conjunto essencial de operadores de Mutação de Interface.

Tem sido demonstrado que a utilização de critérios alternativos, baseados em subconjuntos de operadores, produzem bons resultados mas ainda não se determinou precisamente quais são os operadores, ou quais combinações de operadores são as mais indicadas, ou seja, quais seriam as relações de custo/benefício da aplicação de cada operador. Essa atividade tem como objetivo determinar um conjunto essencial de operadores de mutação para os quais obtém-se uma otimização dessa relação custo/benefício. Ela deve também estabelecer uma estratégia incremental de aplicação dos operadores, de acordo com requisitos de teste e limitações de custo.

- Estudar e definir uma estratégia de teste do nível de unidade ao nível de integração utilizando critérios baseados em mutação.

Este estudo deve analisar quais são as implicações de utilizar-se o teste de mutação na fase de unidade e como isso pode influenciar ou guiar a aplicação da Mutação de Interface na fase de integração e vice-versa.

- Estudar e definir critérios de teste para o teste de programas orientados a objetos.

Muitos dos critérios de teste existentes não são capazes de exercitar de maneira eficiente todas as características de tais softwares como herança, polimorfismo e sobrecarga de operadores. Pretende-se investigar a utilização do teste baseado em mutação nesse contexto, levando-se em conta a flexibilidade desse critério, fator que deve permitir adaptá-lo às características intrínsecas das linguagens orientadas a objetos.

- Evoluir da ferramenta de teste *PROTEUM/IM*.

Nesta evolução devem ser adicionados novos módulos funcionais à ferramenta, como por exemplo, um módulo de minimização de casos de teste, a exemplo do que já foi feito para o nível de unidade [38].

- Utilizar o teste baseado em mutação no teste de projetos de hardware.

Mais uma vez, devido à flexibilidade de utilização do teste baseado em mutação e pela sua grande efetividade em revelar erros, considera-se factível definir e explorar critérios de seleção de casos de teste para projetos de hardware utilizando-se os conceitos do teste de mutação. Uma possível maneira de aplicar teste de mutação nesse escopo seria definir operadores de mutação para alguma linguagem de especificação de hardware como, por exemplo, VHDL.

Bibliografia

- [1] R. S. Pressman. *Software Engineering - A practitioner's approach*. McGraw-Hill, 3a. edição, 1992.
- [2] G. J. Myers. *The Art of Software Testing*. Wiley, New York - EEUU, 1979.
- [3] S. Rapps e E. J. Weyuker. "Data Flow Analysis Techniques for Program Test Data Selection". In *Anais da VI International Conference on Software Engineering*, pp. 272–278. Tokio - Japão, setembro 1982.
- [4] J. C. Maldonado. *Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software*. Tese de Doutorado, DCA/FEE/UNICAMP, Campinas - SP, julho 1991.
- [5] T. A. Budd. "Mutation Analysis: Ideas, Example, Problems and Prospects", chapter Computer Program Testing. North-Holland Publishing Company, 1981.
- [6] R. A. DeMillo, R. J. Lipton e F. G. Sayward. "Hints on Test Data Selection: Help for the Practicing Programmer". *IEEE Computer*, 11(4), abril 1978.
- [7] W. Howden. "Weak Mutation Testing and Completeness of Test Sets". *IEEE Transactions on Software Engineering*, SE-8(4):371–379, julho 1982.
- [8] P. G. Frankl e E. J. Weyuker. "An Applicable Family of Data Flow Testing Criteria". *IEEE Transactions on Software Engineering*, 14(10):1483–1498. outubro 1988.
- [9] P. G. Frankl e E. J. Weyuker. "A formal Analysis of the Fault-detecting Ability of Testing Methods". *IEEE Transactions on Software Engineering*, 19(3):202–213. março 1993.
- [10] A. P. Mathur e W. E. Wong. "A Theoretical Comparison between Mutation and Data Flow Based Test Adequacy Criteria". In *Anais da ACM Computer Science Conference*, pp. 38–45. Phoenix - AZ - EEUU, março 1994.
- [11] R. A. DeMillo, A. P. Mathur e W. E. Wong. "Some Critical Remarks on a Hierarchy of Fault-detecting Abilities of Test Methods". *IEEE Trans. on Software Engineering*, 21(10):858–8616, outubro 1995.
- [12] H. Zhu. "A Formal Analysis of the Subsume Relation between Software Test Adequacy Criteria". *IEEE Transactions on Software Engineering*, 22(4):248–255. abril 1996.

- [13] A. J. Offutt, G. Rothermel e C. Zapf. “An Experimental Evaluation of Selective Mutation”. In *Anais da XV International Conference on Software Engineering*, pp. 100–107, Baltimore - MD - EEUU, maio 1993.
- [14] A. J. Offut e J. H. Hayes. “A Semantic Model of Program Faults”. In *Anais do ISSTA '96*, San Diego - CA - EEUU, 1996.
- [15] A. J. Offutt and S. Lee. “An Empirical Evaluation of Weak Mutation”. *IEEE Transactions on Software Engineering*, 20(5):337–344, maio 1994.
- [16] J. C. Maldonado, S. R. S. Souza, M. E. Delamaro e P. C. Masiero. “Análise de Mutantes: Uma Avaliação Empírica da Propriedade de Antiextensionalidade”. In *Workshop de Qualidade de Software*, Recife - PE, outubro 1995.
- [17] W. E. Wong e A. P. Mathur. “Fault Detection Effectiveness of Mutation and Data Flow Testing”. *Software Quality Journal*, 4(1):69–83, março 1995.
- [18] W. E. Wong e A. P. Mathur. “Reducing the Cost of Mutation Testing: An Empirical Study”. *The Journal of Systems and Software*, 31(3):185–196, dezembro 1995.
- [19] A. P. Mathur e W. E. Wong. “An Empirical Comparison of Data Flow and Mutation Based Test Adequacy Criteria”. *The Journal of Software Testing, Verification, and Reliability*, 4(1):9–31, março 1994.
- [20] A. P. Mathur e W. E. Wong. “Evaluation of the Cost of Alternate Mutation Strategies”. In *Anais do VII Simpósio Brasileiro de Engenharia de Software*, pp. 320–335, Rio de Janeiro, Brazil, outubro 1993.
- [21] W. E. Wong, A. P. Mathur e J. C. Maldonado. “Mutation Versus All-uses: An Empirical Evaluation of Cost, Strength, and Effectiveness”. In *Anais da International Conference on Software Quality and Productivity*, pp. 258–265, Hong Kong, dezembro 1994.
- [22] W. E. Wong, J. C. Maldonado, M. E. Delamaro e A. P. Mathur. “Constrained Mutation in C Programs”. In *Anais do VIII Simpósio Brasileiro de Engenharia de Software*, pp. 439–452, Curitiba - PR, outubro 1994.
- [23] E. J. Weyuker. “The Cost of Data Flow Testing: an Empirical Study”. *IEEE Transactions on Software Engineering*, SE-16(2):121–128, fevereiro 1990.
- [24] M. L. Chaim. *POKE-TOOL - Uma Ferramenta para Suporte ao Teste Estrutural de Programas Baseado em Análise de Fluxo de Dados*. Dissertação de Mestrado, DCA/FEE/UNICAMP, Campinas - SP, abril 1991.
- [25] R. P. Fonseca. *Suporte ao Teste Estrutural de Programas FORTRAN no Ambiente POKE-TOOL*. Dissertação de Mestrado, DCA/FEE/UNICAMP, Campinas - SP, janeiro 1993.
- [26] P. S. J. Leitão. *Suporte ao Teste Estrutural de Programas Cobol no Ambiente POKE-TOOL*. Dissertação de Mestrado, DCA/FEE/UNICAMP, Campinas - SP, agosto 1992.

- [27] B. J. Choi, R. A. DeMillo, E. W. Krauser, A. P. Mathur, R. J. Martin, A. J. Offutt, H. Pan e E. H. Spafford. "The Mothra Toolset". In *Anais da XXII Annual Hawaii International Conference on System Sciences*, Hawai - EEUU, janeiro 1989.
- [28] J. R. Horgan e S. A. London. "ATAC- Automatic Test Coverage Analysis for C Programs". Documentação interna Bellcore, junho 1990.
- [29] M. E. Delamaro. "*Proteum: Um Ambiente de Teste Baseado na Análise de Mutantes*". Dissertação de Mestrado, SCE-ICMSC-USP, São Carlos - SP, outubro 1993.
- [30] M. E. Delamaro, J. C. Maldonado, M. Jino e M. L. Chaim. "Proteum: Uma Ferramenta de Teste Baseada na Análise de Mutantes (Proteum: A Testing Tool Based on Mutation Analysis)". In *Caderno de Ferramentas do VII Simpósio Brasileiro de Engenharia de Software*, Rio de Janeiro - RJ, outubro 1993.
- [31] M. E. Delamaro e J. C. Maldonado. "Proteum - A Tool for the Assesment of Test Adequacy for C Programs". In *Anais da Conference on Performability in Computing Systems (PCS 96)*, Brunswick - NJ - EEUU, julho 1996.
- [32] M. J. Harrold e M. L. Soffa. "Selecting and Using Data for Integration Test". *IEEE Software*, 8(2):58-65, março 1991.
- [33] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero e M. E. Delamaro. "Análise de Mutantes Baseada em Máquinas de Estado Finito". In *Anais do XI Simpósio Brasileiro de Redes de Computadores*, pp. 407-425, Campinas - SP, maio 1993.
- [34] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero e M. E. Delamaro. "Mutation Analysis Testing for Finite State Machines". In *Anais do V International Symposium on Software Reliability Engineering*, San Jose - CA - EEUU, 1994.
- [35] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero e M. E. Delamaro. "Mutation Analysis Applied to Validate Specifications Based on Petri Nets". In *Anais do VIII IFIP Conference on Formal Descriptions Techniques for Distributed Systems and Communication Protocols*, Montreal - Canadá, outubro 1995.
- [36] S. C. P. F. Fabbri. "*A Análise de Mutantes no Contexto de Sistemas Reativos: Uma Contribuição para o Estabelecimento de Estratégias de Teste e Validação*". Tese de Doutorado, IFSC - USP, São Carlos - SP, outubro 1996.
- [37] M. R. Woodward. "Mutation Testing - its Origin and Evolution". *Information and Software Technology*, 35(3):163-169, março 1993.
- [38] S. R. S. Souza. "*Avaliação do Custo e Eficácia do Critério Análise de Mutantes na Atividade de Teste de Programas*". Dissertação de Mestrado, SCE-ICMSC-USP, São Carlos - SP, junho 1996.

- [39] W. E. Wong e A. P. Mathur. "How Strong is Constrained Mutation in Fault Detection?". In *Anais do International Computer Symposium*, pp. 515–520, Taiwan, dezembro 1994.
- [40] A. Haley e S. Zweben. "Development and Application of a White Box Approach to Integration Testing". *The Journal of Systems and Software*, 4:309–315, 1984.
- [41] U. Linnenkugel e M. Müllerburg. "Test Data Selection Criteria for (Software) Integration Testing". In *Anais da I International Conference on Systems Integration*, pp. 709–717, Mornstown - NJ - EEUU, abril 1990.
- [42] IEEE. Standard glossary of software engineering terminology. Padrão 610.12, IEEE, 1990.
- [43] H. Zhu e P. Hall. "Test Data Adequacy Measurements". *Software Engineering Journal*, 8(1):211–224, janeiro 1993.
- [44] M. R. Woodward, D. Heddley e M. A. Hennel. "Experience with Path Analysis and Testing of Programs". *IEEE Transactions on Software Engineering*, SE-6:278–286, maio 1980.
- [45] W. Howden. "Methodology for the Generation of Program Test Data". *IEEE Computer*, C-24(5):554–559, 1975.
- [46] S. C. Ntafos. "On Required Element Testing". *IEEE Transactions on Software Engineering*, SE-10:795–803, novembro 1984.
- [47] L.J. Morell. "A Theory of Fault-based Testing". *IEEE Transactions on Software Engineering*, 16(8):844–857, agosto 1990.
- [48] T. A. Budd e D. Angluin. "Two Notions of Correctness and their Relation to Testing". *Acta Informatica*, 18(1):31–45, novembro 1982.
- [49] R. A. DeMillo e A. J. Offutt. "Constraint Based Automatic Test Data Generation". *IEEE Trans. on Software Engineering*, 17(9):900–910, setembro 1991.
- [50] H. Agrawal, R. A. DeMillo, R. Hataway, Wm. Hsu, W. Hsu, E. Krauser, R. J. Martin, A. P. Mathur e E. Spafford. "Design of Mutant Operators for C Programming Language". Relatório Técnico SERC-TR41-P, Software Engineering Research Center, Purdue University, West Lafayette - IN - EEUU, março 1989.
- [51] T. A. Budd. "*Mutation Analysis of Program Test Data*". Tese de Doutorado, Yale University, New Haven - CT - EEUU. 1980.
- [52] E. W. Krauser, A. P. Mathur e V. J. Rego. "High Performance Software Testing on Simd Machines". *IEEE Transactions on Software Engineering*, 17(5):403–422, maio 1991.

- [53] A. P. Mathur e E. W. Krauser. “Modeling Mutation on Vector Processor”. In *Anais da X International Conference on Software Engineering*, pp. 154–161, Singapura, abril 1988.
- [54] B. J. Choi, A. P. Mathur e A. P. Pattison. “pMothra: Scheduling Mutants for Execution on a Hypercube”. In *Anais do III Symposium on Software Testing, Analysis and Verification*, pp. 58–65, Key West - FL - EEUU, dezembro 1989.
- [55] B. J. Choi e A. P. Mathur. “High-performance Mutation Testing”. *Journal of System and Software*, 1(20):135–152, fevereiro 1993.
- [56] A. T. Acree. “On Mutation”. Tese de Doutorado, Georgia Institute of Technology, Atlanta - GA - EEUU, 1980.
- [57] A. P. Mathur. “Performance, Effectiveness and Reliability Issues in Software Testing”. In *Anais da XV Annual International Computer Software and Applications Conference*, pp. 604–605, Tokio - Japão, setembro 1991.
- [58] E. J. Weyuker. “The Complexity of Data Flow for Test Data Selection”. *Information Processing Letters*, 19(2):103–109, agosto 1984.
- [59] W. E. Wong. “On Mutation and Data Flow”. Tese de Doutorado, Departamento de Ciência da Computação, Purdue University, W. Lafayette - IN - EEUU, dezembro 1993.
- [60] P. G. Frankl e E. J. Weyuker. “Provable Improvements on Branch Testing”. *IEEE Transactions on Software Engineering*, 19(10):962–975, outubro 1993.
- [61] E. J. Weyuker, T. Goradia e A. Singh. “Automatically Generating Test Data from a Boolean Specification”. *IEEE Transactions on Software Engineering*, 20(5):353–363, maio 1994.
- [62] M. R. Girgis e M. R. Woodward. “An Integrated System for Program Testing Using Weak Mutation and Data Flow Analysis”. In *Anais da VIII International Conference on Software Engineering*, pp. 313–319, 1985.
- [63] F. G. Frankl e E. J. Weyuker. “Data Flow Testing Tools”. In *Anais da Softfair II*, pp. 46–53, San Francisco - CA - EEUU, dezembro 1985.
- [64] J. R. Horgan e P. Mathur. “Assessing Testing Tools in Research and Education”. *IEEE Software*, 9(3):61–69, maio 1992.
- [65] P. R. S. Vilela. *Uma Ferramenta para Auxílio Visual ao Teste e Depuração de Programas*. Dissertação de Mestrado, DCA/FEE/UNICAMP, Campinas - SP, março 1994.
- [66] M. E. Delamaro, J. C. Maldonado e A. P. Mathur. “Proteum - A Tool for the Assesment of Test Adequacy for C Programs - User’s Guide”. Relatório Técnico SERC-TR168-P, Software Engineering Research Center, Purdue University, West Lafayette - IN - EEUU, abril 1996.

- [67] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold Company, New York, 2a. edição, 1990.
- [68] V. R. Basili e B. T. Perricone. “Software Errors and Complexity: An Empirical Study”. *Communications of the ACM*, 27(1):42–52, janeiro 1994.
- [69] L. J. White e H. K. N. Leung. Integration Testing. *Encyclopedia of Software Engineering*, pp. 573–577, 1994.
- [70] H. K. N. Leung e L. White. “A Study of Integration Testing and Software Regression at the Integration Level”. In *Anais da Conference on Software Maintenance-90*, pp. 290–301, San Diego - CA - EEUU, 1990.
- [71] F. Solheim e J. H. Rowland. “An Empirical Study of Testing and Integration Strategies Using Artificial Software Systems”. *IEEE Transactions on Software Engineering*, 19(10):941–949, outubro 1993.
- [72] W. Howden. “Reliability of the Path Analysis Testing Strategy”. *IEEE Transactions on Software Engineering*, SE-2:208–214, setembro 1976.
- [73] A. J. Offutt. “Coupling Effect: Fact or Fiction”. In *Anais do III Symposium on Software Testing, Analysis, and Verification*, pp. 131–140, Key West - FL - EEUU, dezembro 1989.
- [74] E. H. Spafford. “Extending Mutation Testing to Find Environmental Bugs”. *Software Practice and Experience*, 20(2):181–189, fevereiro 1990.
- [75] M. E. Delamaro, J. C. Maldonado e A. P. Mathur. “Integration Testing Using Interface Mutation”. In *Anais do VII International Symposium of Software Reliability Engineering (ISSRE)*, New York - NY - EEUU, novembro 1996.
- [76] E. F. Johnson. *Graphical Applications with Tcl & Tk*. M&T Books, New York, 1996.
- [77] M. E. Delamaro e J. C. Maldonado. “Proteum/IM - User’s Guide”. Relatório Técnico em preparação, ICMSC - USP, São Carlos - SP, abril 1997.
- [78] M. Carnassale. *GFC - Uma Ferramenta Multilíngua para Geração de Grafo de Programa*. Dissertação de Mestrado, DCA/FEE/UNICAMP, Campinas - SP, fevereiro 1991.
- [79] A. Pasquini. “Space Program”. Correspondência Pessoal, 1996.

Apêndice A

Defeitos do Programa *SPACE*

São apresentados nesse Apêndice os 33 defeitos que foram encontrados no processo de desenvolvimento do programa *SPACE* e utilizados no experimento do Capítulo 5.

- Defeito número 1.

Comando com defeito

```
curr_ptr=&p1;
```

Correção

```
*curr_ptr=p1;
```

- Defeito número 2.

Comando com defeito

```
curr_ptr=&p1;
```

Correção

```
*curr_ptr=p1;
```

- Defeito número 3.

Comando com defeito

```
curr_ptr=tp;
```

Correção

```
*curr_ptr=*tp;
```

- Defeito número 4.

Comando com defeito

```
pp2=curr_ptr;
```

Correção

```
*pp2=*curr_ptr;
```

- Defeito número 5.

Comando com defeito

```
pp2=curr_ptr;
```

Correção

```
*pp2=*curr_ptr;
```

- Defeito número 6.

Comando com defeito

```
tp=curr_ptr;
```

Correção

```
*tp=*curr_ptr;
```

- Defeito número 7.

Comando com defeito

```
a = ((p1_et+p2_et-2*e)/(2*x1*x1));
c = ((a*x1*x1+e-p1_et)/x1);
b = ((q1_et+q2_et-2*e)/(2*y1*y1));
d = ((b*y1*y1+e-q1_et)/y1);
```

Correção

```
if (x1==0) {
    a=c=0;
}
else {
    a = ( (p1_et+p2_et-2*e) / (2*x1*x1) );
    c = ( (a*x1*x1+e-p1_et) / x1 );
};
if (y1==0) {
    b=d=0;
}
else {
    b = ( (q1_et+q2_et-2*e) / (2*y1*y1) );
    d = ( (b*y1*y1+e-q1_et) / y1 );
};
```

- Defeito número 8.

Comando com defeito

```
a = ( (p1_ep+p2_ep-2*e) / (2*x1*x1) );
c = ( (a*x1*x1+e-p1_ep) / x1 )      ;
b = ( (q1_ep+q2_ep-2*e) / (2*y1*y1) );
d = ( (b*y1*y1+e-q1_ep) / y1 )      ;
```

Correção

```
if (x1==0) {
    a=c=0;
}
else {
a = ( (p1_ep+p2_ep-2*e) / (2*x1*x1) );
c = ( (a*x1*x1+e-p1_ep) / x1 )      ;
    };
    if (y1==0) {
        b=d=0;
    }
    else {
b = ( (q1_ep+q2_ep-2*e) / (2*y1*y1) );
d = ( (b*y1*y1+e-q1_ep) / y1 )      ;
    };
```

- Defeito número 9.

Comando com defeito

```
x = p*pstep + q*qstep*cos(angle);
y = q*qstep*sin(angle);
```

Correção

```
y = q*qstep*dsin(angle);
x = p*pstep + q*qstep*dcos(angle);
```

- Defeito número 10.

Comando com defeito

```
x = P[i]*pstep + Q[i]*qstep*cos(angle);
y = Q[i]*qstep*sin(angle);
```

Correção

```
x = P[i]*pstep + Q[i]*qstep*dcos(angle);
y = Q[i]*qstep*dsin(angle);
```

- Defeito número 11.

Comando com defeito

```
x = p*pstep + q*qstep*cos(angle);
y = q*qstep*sin(angle);
```

Correção

```
x = p*pstep + q*qstep*dcos(angle);  
y = q*qstep*dsin(angle);
```

- Defeito número 12.

Comando com defeito

```
XE = ((XD-XC) * cos(phi)) - ((YD-YC) * sin(phi)) + XC ;  
YE = ((XD-XC) * sin(phi)) + ((YD-YC) * cos(phi)) + YC ;
```

Correção

```
XE = ((XD-XC) * dcos(phi)) - ((YD-YC) * dsin(phi)) + XC ;  
YE = ((XD-XC) * dsin(phi)) + ((YD-YC) * dcos(phi)) + YC ;
```

- Defeito número 13.

Comando com defeito

```
qstep = 2 * radius * 2 * cos(angle);
```

Correção

```
qstep = 2 * radius * 2 * dcos(angle);
```

Comando com defeito

```
angle=atan( (2 * qdim)/(pdim) );
```

Correção

```
angle=datan( (2 * qdim)/(pdim) )
```

Comando com defeito

```
pstep=pdim; qstep=(qdim)/(sin(angle));
```

Correção

```
pstep=pdim; qstep=(qdim)/(dsin(angle));
```

Comando com defeito

```
if ( (grid->ANGLE <= (atan(elem->QDIM / elem->PDIM))) ||  
      (grid->ANGLE > 90) ) {
```

Correção

```
if ( (grid->ANGLE <= (datan(elem->QDIM / elem->PDIM))) ||  
      (grid->ANGLE > 90) ) {
```

- Defeito número 14.

Comando com defeito

```
GetKeyword(Keywords[88],curr_ptr);
```

Correção

```
error=(GetKeyword(Keywords[88],curr_ptr));
```

- Defeito número 15.

Comando com defeito

```
GetKeyword(Keywords[88],curr_ptr);
```

Correção

```
error=(GetKeyword(Keywords[88],curr_ptr));
```

- Defeito número 16.

Comando com defeito

```
(gnode_ptr->GEOMPORT_PTR)->PPA += phase;
```

Correção

```
(gnode_ptr->GEOMPORT_PTR)->PPA = phase;
```

Comando com defeito, quando defeito 27 também está presente

```
(group_ptr->ELEM_PTR)->PORT_PTR->PPA =  
    (geomnode_app_ptr->GEOMPORT_PTR)->PPA;
```

Correção

```
(group_ptr->ELEM_PTR)->PORT_PTR->PPA +=  
    (geomnode_app_ptr->GEOMPORT_PTR)->PPA;
```

Comando com defeito, quando defeito 27 não está presente

```
(geomnode_app_ptr->GEOMPORT_PTR)->PPA =  
    (group_ptr->ELEM_PTR)->PORT_PTR->PPA;
```

Correção

```
(geomnode_app_ptr->GEOMPORT_PTR)->PPA +=  
    (group_ptr->ELEM_PTR)->PORT_PTR->PPA;
```

- Defeito número 17.

Falta comando. Correção é a inclusão de

```
port_ptr->OMIT_POL    = 1
```

- Defeito número 18.

Comando com defeito

```
app_ptr->PSEA += phi;
```

Correção

```
app_ptr->PHEA += phi;
```

- Defeito número 19.

Comando com defeito

```
if ((group_ptr->ELEM_PTR)->POLARIZATION != 0 )
    return 0;
```

Correção é eliminação dos comandos acima.

- Defeito número 20.

Comando com defeito

```
can = angle_step;
```

Correção

```
can += angle_step;
```

- Defeito número 21.

Comando com defeito

```
cph = phase_step;
```

Correção

```
cph += phase_step;
```

- Defeito número 22.

Correção é a inclusão do seguinte trecho. Note que o ramo “else” é parte do programa original, com defeito.

```
if ( (pmin==pmax)&&(qmin==qmax) ) {
    gnodevis(pmin,qmin,cph,can,g);
    cph += phase_step;
    can += angle_step;
    cont++;
    if (cont==nodes_num) endvisit=1;
}

else {
...
}
```

- Defeito número 23.

Comando com defeito

```
while (app_ptr->NEXT!=((void *) 0) ) {
```

Correção

```
while (app_ptr!=((void *) 0) ) {
```

- Defeito número 24.

Correção é a inclusão desses 4 trechos. Note que os ramos “else” são parte do programa original, com o defeito.

```
if ((group_ptr->GRPHAEXC_PTR)->TYPE==ROTATION_SEQUENTIAL_LAW) {
    grid->PSTEP = elem->RADIUS;
    grid->QSTEP = elem->RADIUS;
}
else {
    ...
}
```

```
if ((group_ptr->GRPHAEXC_PTR)->TYPE==ROTATION_SEQUENTIAL_LAW) {
    if (grid->PSTEP < elem->RADIUS)
        printf("\n%s%s",MOSErrors[12],group_ptr->NAME);
}
else {
    ...
}
```

```
if ((group_ptr->GRPHAEXC_PTR)->TYPE==ROTATION_SEQUENTIAL_LAW) {
    grid->PSTEP = elem->RADIUS;
    grid->QSTEP = elem->RADIUS;
}
else {
    ...
}
```

```
if ((group_ptr->GRPHAEXC_PTR)->TYPE==ROTATION_SEQUENTIAL_LAW) {
    if (grid->PSTEP < elem->RADIUS)
        printf("\n%s%s",MOSErrors[12],group_ptr->NAME);
}
else {
    ...
}
```

- Defeito número 25.

Comando com defeito

```
for (q=q1;q<=q2;q++) {
    for (p=p1;p<=p2;p++) {
```


Correção

```
for (q=intmin(q1,q2);q<=intmax(q1,q2);q++) {  
    for (p=intmin(p1,p2);p<=intmax(p1,p2);p++) {
```

- Defeito número 26.

Comando com defeito

```
ultimo_ptr=((void *) 0) ;
```

Correção

```
ultimo_ptr->NEXT=((void *) 0) ;
```

- Defeito número 27.

Comando com defeito, se defeito 16 também está presente

```
(group_ptr->ELEM_PTR)->PORT_PTR->PSH =  
    (geomnode_app_ptr->GEOMPORT_PTR)->PSH;  
(group_ptr->ELEM_PTR)->PORT_PTR->PSC =  
    (geomnode_app_ptr->GEOMPORT_PTR)->PSC;  
(group_ptr->ELEM_PTR)->PORT_PTR->PPA =  
    (geomnode_app_ptr->GEOMPORT_PTR)->PPA;
```

Correção

```
(geomnode_app_ptr->GEOMPORT_PTR)->PSH =  
    (group_ptr->ELEM_PTR)->PORT_PTR->PSH;  
(geomnode_app_ptr->GEOMPORT_PTR)->PSC =  
    (group_ptr->ELEM_PTR)->PORT_PTR->PSC;  
(geomnode_app_ptr->GEOMPORT_PTR)->PPA =  
    (group_ptr->ELEM_PTR)->PORT_PTR->PPA;
```

Comando com defeito, se defeito 16 não está presente

```
(group_ptr->ELEM_PTR)->PORT_PTR->PSH =  
    (geomnode_app_ptr->GEOMPORT_PTR)->PSH;  
(group_ptr->ELEM_PTR)->PORT_PTR->PSC =  
    (geomnode_app_ptr->GEOMPORT_PTR)->PSC;  
(group_ptr->ELEM_PTR)->PORT_PTR->PPA +=  
    (geomnode_app_ptr->GEOMPORT_PTR)->PPA;
```

Correção

```
(geomnode_app_ptr->GEOMPORT_PTR)->PSH =  
    (group_ptr->ELEM_PTR)->PORT_PTR->PSH;  
(geomnode_app_ptr->GEOMPORT_PTR)->PSC =  
    (group_ptr->ELEM_PTR)->PORT_PTR->PSC;  
(geomnode_app_ptr->GEOMPORT_PTR)->PPA +=  
    (group_ptr->ELEM_PTR)->PORT_PTR->PPA;
```

- Defeito número 28.

Comando com defeito

```
app_ptr=addrem_ptr;
```

Correção

```
*app_ptr=*addrem_ptr;
```

- Defeito número 29.

Comando com defeito

```
strcpy(Keywords [84], "P1_ET");  
strcpy(Keywords [85], "P2_ET");  
strcpy(Keywords [86], "Q1_ET");  
strcpy(Keywords [87], "Q2_ET");
```

Correção

```
strcpy(Keywords [84], "P1_VAL");  
strcpy(Keywords [85], "P2_VAL");  
strcpy(Keywords [86], "Q1_VAL");  
strcpy(Keywords [87], "Q2_VAL");
```

Comando com defeito

```
strcpy(Keywords [89], "P1_EP");  
strcpy(Keywords [90], "P2_EP");  
strcpy(Keywords [91], "Q1_EP");  
strcpy(Keywords [92], "Q2_EP");
```

Correção

```
strcpy(Keywords [89], "P1_VAL");  
strcpy(Keywords [90], "P2_VAL");  
strcpy(Keywords [91], "Q1_VAL");  
strcpy(Keywords [92], "Q2_VAL");
```

- Defeito número 30.

Comando com defeito

```
addrem_ptr=((void *) 0) ;
```

Correção

```
*addrem_ptr=((void *) 0) ;
```

- Defeito número 31.

Comando com defeito

```
if (error!=0) {
```

Correção

```
if (error==17) {
```

- Defeito número 32.

Correção é trocar o lugar do comando

```
g->PHASE_UNIT = 0 ;
```

- Defeito número 33.

Comando com defeito

```
gnode_ptr->PHEA = angle;
```

Correção

```
gnode_ptr->PHEA += angle;
```