Teste de Integração: Projeto de Operadores para o Critério Mutação de Interface

Márcio Eduardo Delamaro
Instituto de Física de São Carlos - USP

José Carlos Maldonado
Instituto de Ciências Matemáticas de São Carlos - USP
Cx Postal 668, 13560-970
São Carlos - SP
{med,jcmaldon}@icmsc.sc.usp.br

Resumo

Um dos mais importantes pontos na atividade de teste é a seleção de conjuntos de teste que sejam eficazes. Diversos critérios de adequação de teste têm sido propostos para auxiliar na seleção de casos de teste mas a maioria deles é restrita ao teste de unidade. Esse fato deve-se principalmente às características dos requisitos de teste requeridos por esses critérios que limitam-se ao escopo de uma única unidade. Este artigo apresenta um critério interprocedimental baseado em mutação, denominado Mutação de Interface, que pode ser aplicado no nível de integração de software e o conjunto de operadores de mutação especificamente projetado para esse critério. É apresentado também um resumo de resultados obtidos com a aplicação desse critério em alguns estudos empíricos já conduzidos utilizando-se a ferramenta *PROTEUM/IM* que apóia a aplicação do critério Mutação de Interface.

Palavras-chave: Teste de Software, Teste de Mutação, Mutação de Interface, Operadores de Mutação

Abstract

One of the most important points in the testing activity is the selection of effective test case sets. Several test adequacy criteria have been proposed to support the selection of test cases but most of them are restricted to unit test. This fact is mostly due to the characteristcs of such testing criteria that specify test requirements in the scope of a single unit. This paper presents a mutation based interprocedural test criterion, named Interface Mutation, suitable to be used at integration testing phase and the set of mutation operators designed specifically to that criterion. Also presents a summary of results obtained by applying the criterion in some empirical studies already conducted using *PROTEUM/IM*, a tool that supports the application of Interface Mutation criterion.

Keywords: Software Testing, Mutation Testing, Interface Mutation, Mutation Operators

1 Introdução

Uma das mais caras e difíceis atividades no desenvolvimento de software é o teste. Ele é iniciado no nível de unidade, quando cada unidade é testada separadamente e cujo objetivo

é verificar aspectos algorítmicos de cada uma delas. No outro extremo está o teste de sistema quando o software como um todo é testado e seu comportamento funcional, bem como sua interação com o sistema em que está inserido são avaliados. Entre estas duas fases existe ainda o teste de integração cujo objetivo é revelar erros na interação entre as unidades, quando elas são integradas para construir o software como um todo.

No contexto de teste de software, um ponto fundamental é o projeto de casos de teste. Diversos métodos têm sido propostos para que se avalie quão "bom" é um determinado conjunto de teste. Infelizmente, a maioria desses critérios de avaliação é restrita ao teste de unidade. Entre eles, critérios de fluxo de dados [11] e teste de mutação [4] podem ser citados como alguns dos mais promissores.

Por outro lado, existe uma falta de critérios de adequação de conjuntos de teste para fases de teste de mais alto nível. Em particular, para o teste de integração tem-se utilizado quase que exclusivamente critérios de teste funcionais. Umas das poucas exceções são o trabalho de Haley e Zweben [5] que propõe um critério para seleção de caminhos de uma unidade que devem ser re-testados na fase de integração, baseado na interface dessa unidade; o trabalho de Linnenkugel e Müllerburg [7] que propuseram critérios baseados na estrutura modular do software e na análise do fluxo de dados entre as unidades; e de Harrold e Soffa [6] que propõem um critério interprocedimental baseado em fluxo de dados. Esse último trabalho deve ser destacado pois, além de especificar requisitos de teste a serem satisfeitos, apresenta também soluções para que esse critério possa ser implementado, ou seja, para que dado um programa, possa-se automaticamente determinar os requisitos a serem satisfeitos e, dado um conjunto de teste, determinar o quanto ele é adequado segundo esses requisitos. Esse é um dos sérios problemas a serem resolvidos na especificação de critérios de teste.

Delamaro e Maldonado [3] propuseram um critério interprocedimental baseado em mutações. Esse critério, batizado Mutação de Interface exige que sejam selecionados casos de teste que exercitem as conexões entre as diversas unidades do software. Isso é feito através da criação de programas alternativos, chamados mutantes, que possuem pequenas diferenças sintáticas em relação ao programas em teste. Essas diferenças sintáticas, introduzidas em pontos do programa relacionados às conexões entre unidades, produzem mutantes que possuem "erros de integração", ou seja, valores incorretos sendo propagados através das interações entre as unidades. Casos de teste que distingam esses mutantes do programa em teste devem exercitar essas interações de maneira eficaz.

Um dos pontos fundamentais no teste baseado em mutações e em particular no critério Mutação de Interface, é o tipo de alterações sintáticas – as mutações – que são introduzidas na criação dos mutantes. Elas são definidas por **operadores de mutação** que buscam modelar defeitos usualmente cometidos em programas numa certa linguagem. O propósito desse artigo é apresentar e discutir o conjunto de operadores definidos especificamente para a Mutação de Interface. Esses operadores foram implementados na ferramenta de teste *PROTEUM/IM* e foram utilizados em alguns experimentos que visam a avaliar a eficácia do critério.

Na próxima seção, faz-se uma breve discussão sobre o critério Mutação de Interface. A Seção 3 apresenta os operadores de mutação. A Seção 4 faz uma comparação entre o critério Mutação de Interface utilizando os operadores propostos e o critério interprocedimental proposto por Harrold e Soffa [6]. A Seção 5 apresenta o sumário de alguns experimentos conduzidos utilizando-se Mutação de Interface. A Seção 6 apresenta a conclusão do artigo.

2 Mutação de Interface

A idéia do teste de mutação é avaliar a qualidade de um conjunto de teste baseado na sua habilidade em revelar defeitos simples injetados no programa sendo testado. Tais conjuntos de teste devem ser eficazes também para revelar outros tipos de defeitos, de acordo com a hipótese conhecida como **Efeito de Acoplamento** [9]

Na fase de integração do software é necessário que sejam testadas as interações das unidades com as outras partes do software, e não as unidades em si. Esse é o objetivo do critério Mutação de Interface. Erros de integração são causados por valores trocados através da conexão entre unidades [1]. Basicamente, podem ser identificados três tipos de erros de integração entre as unidades F e G, mostrados na Figura 1: 1) o conjunto de valores de entrada $S_I(G)$ possui algum valor incorreto que leva a um resultado incorreto (uma falha) antes do retorno de G; 2) $S_I(G)$ possui algum valor incorreto que faz com que algum valor do conjunto de retorno $S_O(G)$ seja incorreto, causando uma falha após o retorno de G; e 3) todos os valores de entrada $S_I(G)$ são corretos porém G produz algum valor incorreto para $S_O(G)$, conduzindo a uma falha após o seu retorno;

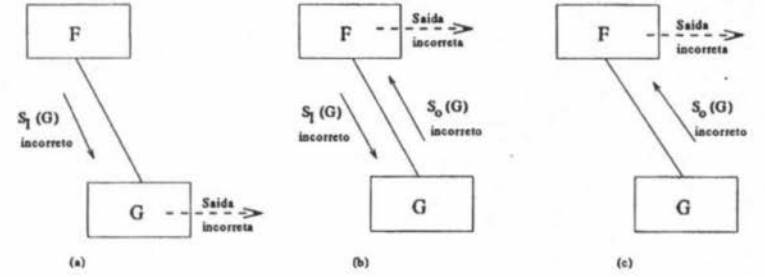


Figura 1: Modelo de falhas causadas por erros de integração: (a) Tipo 1; (b) Tipo 2; e (c) Tipo 3

As mutações a serem realizadas nesta fase do teste – visando reproduzir erros de integração – são introduzidas diretamente naqueles pontos relacionados com a interação entre módulos. Utilizando o mesmo pensamento do teste de mutação convencional – e baseado em experimentos preliminares [2] – assume-se que casos de teste que são capazes de distinguir esses "mutantes-de-interface" devem ser capazes de revelar também a maioria dos erros de integração. Obviamente, essa afirmação depende de quais mutantes são utilizados, ou em outras palavras, de quais são os operadores de mutação utilizados.

Cada mutante na Mutação de Interface, ao contrário do teste de mutação convencional, não está associado a uma unidade mas sim a uma conexão entre duas unidades. Por exemplo, dada uma conexão entre as unidades f e g, onde f faz chamadas a g, ao testar-se essa conexão os operadores de mutação são aplicados em dois pontos do programa: 1) nas chamadas a g dentro da unidade f; e 2) nos pontos dentro de g relacionados com sua interface. Nesse segundo caso, é necessário um mecanismo adicional que permita identificar o ponto de onde g foi chamada. Uma vez que somente a conexão f-g está sendo testada, a mutação deve acontecer somente se g é chamada por f. Senão, a unidade g deve comportar-se como no programa original, de modo que não se possa distinguir o mutante se a conexão f-g não for exercitada. Essa característica pode requerer, para algumas

linguagens de programação como C, que a decisão de aplicar-se ou não uma mutação seja feita em tempo de execução do mutante.

3 Operadores de Mutação de Interface para C

Os operadores de Mutação de Interface apresentados nessa 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.

O conjunto de operadores de Mutação de Interface apresentados 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 com o qual aplicar o critério, estabelecendo estratégias incrementais de aplicação do critério. Uma maneira de customizar a aplicação do critério, da mesma maneira que Wong et al. [13] fizeram para o teste de mutação convencional, é aplicar 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.

Outro tipo de parametrização desejável, 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 [2].

Conforme mostram os resultados de estudos empíricos relatados na Seção 5, os operadores de Mutação de Interface podem ser aplicados – graças a essas características de parametrização – com custos bastante reduzidos e ainda assim apresentam um alto grau de efetividade em revelar defeitos. Em resumo, procurou-se projetar um conjunto de operadores eficaz em exercitar todas as possíveis formas de interação entre duas unidades e que, embora possa conduzir a um conjunto de mutantes cujo custo de utilização completa seja muito alto, possa também ser parametrizado e ajustado a restrições de custo.

Para o teste de unidade, operadores de mutação buscam modelar os defeitos 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 "instrumen-

tados". Eles não modelam defeitos comuns mas procuram assegurar aos casos de teste características mínimas desejadas. Por 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, apresentados neste trabalho, 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. 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, a cada chamada de g corresponde um conjunto de mutantes que só podem ser distinguidos através da execução da chamada correspondente. Se chamarmos g' a função criada ao aplicar-se o 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. Por isso uma mutação aplicada dentro da função chamada só deve ser efetivamente aplicada se a função foi chamada do ponto cuja conexão deseja-se testar. Somente dessa maneira pode-se exercitar de maneira mais completa a conexão, que pode ser composta por mais de uma chamada de função.

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 dereferências a parâmetros do tipo ponteiro ou vetor. Por exemplo, se um parâmetro formal v é definido como "int *v", v e "*v" pertencem a esse conjunto[1]

$G(g)$: 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

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

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

C e operações associadas a esses tipos. Por exemplo, para o tipo inteiro, esses valores seriam 0, 1, -1, o maior inteiro representável e o inteiro mais negativo.

A Tabela 1 apresenta um resumo com todos os operadores de mutação definidos.

Tabela 1: 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	incrementa e decrementa variável de interface
IndVarIncDec	incrementa e decrementa variável não de interface e constante
DirVarAriNeg	acrescenta operador de negação aritmética em variáveis de interface
IndVarAriNeg	acrescenta operador de negação aritmética em variáveis não de interface
DirVarLogNeg	acrescenta operador de negação lógica em variáveis de interface
IndVarLogNeg	acrescenta operador de negação lógica em variáveis não de interface
DirVarBitNeg	acrescenta operador de negação de bit em variáveis de interface
IndVarBitNeg	acrescenta operador de negação de bit em variáveis não de interface
RetStaDel	elimina comando de retorno
RetStaRep	troca comando de retorno
CovAllNode	cobertura de nós
CovAllEdge	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 operador de negação aritmética antes de argumento
ArgLogNeg	acrescenta operador de negação lógica antes de argumento
ArgBitNeg	acrescenta operador de negação de bit antes de argumento
FuncCalDel	elimina chamada de função

3.1 Grupo I - Mutações Efetuadas na Função Chamada

Os operadores desse grupo aplicam mutações dentro do corpo da função chamada. 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* [1] optou-se por incluir um mecanismo que imediatamente antes da chamada da função, no ponto desejado, habilita a mutação e no retorno desabilita-a.

Cada operador de mutação tem um certo escopo de aplicação. Por exemplo, alguns

podem ser aplicados em referências a variáveis, outros a expressões e outros até em comandos inteiros. Para todos os operadores deste grupo, quando uma variável é um ponto de mutação, considera-se que derreferências ou indexações àquela variável também o são. Assim, ao dizer-se que uma variável v é um ponto de mutação, subentende-se que também "*v" e "v[i]" são pontos de aplicação e devem também ser alterados.

O programa da Figura 2 é usado para exemplificar a aplicação dos operadores de mutação. A Tabela 2 mostra os mutantes gerados por operadores do Grupo I, para a conexão r-s.

```
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, vet, *vet}
$G(s)$= {a}
$L(s)$= {j}
$E(s)$= {b}
$C(s)$= {1, 0}

Figura 2: Programa exemplo para aplicação dos mutantes

Operadores de Troca de Variáveis de Interface

A primeira e mais direta maneira de perturbar um valor entrando ou saindo na 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 contém o prefixo "DirVarRep", mostrados na Tabela 1. 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.

Operadores de Troca de Variáveis Não de Interface

Outra maneira de modificar um valor entrando ou saindo da função é 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 1.

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 eficiência, esses operadores 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

Tabela 2: Mutantes para operadores do Grupo I

Operador	Mutante	Operador	Mutante
DirVarRepPar	i = 0	DirVarRepGlob	j < a
	*vet = 0		a += a
	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
DirVarRepConst	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
IndVarRepPar	a = i	IndVarRepReq	a = -1
	a = *vet		a = MININT
	i < i		a = MAXINT
	*vet < i		-1 < i
	a += vet[i]		MININT < i
	a += vet[*vet]		MAXINT < i
IndVarRepConst	a = 1		a += vet[-1]
	0 < i		a += vet[MININT]
	1 < i		a += vet[MAXINT]
	a += vet[0]	IndVarRepGlob	a = a
	a += vet[1]		a < i
IndVarRepLoc	a = j		a += vet[a]
DirVarIncDec	j < ++i	IndVarIncDec	a = (0 + 1)
	j < --i		a = (0 - 1)
	a += ++(vet[j])		++j < i
	a += --(vet[j])		--j < i
	return ++a		a += vet[++j]
	return --a		a += vet[--j]
DirVarAriNeg	j < -i	IndVarAriNeg	a = -0
	a += -vet[j]		-j < i
	return -a		a += vet[-j]
DirVarLogNeg	j < ! i	IndVarLogNeg	a = ! 0
	a += ! vet[j]		! j < i
	return ! a		a += vet[! j]
DirVarBitNeg	j < ~ i	IndVarBitNeg	a = ~ 0
	a += ~ vet[j]		~ j < i
	return ~ a		a += vet[~ j]

utilizadas numa expressão que também utiliza variáveis de interface; e 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:

```
if (x < i) ...
```

```
j = x[i];
x << i;
return i;
```

mas não o será nos seguintes casos:

```
i += 10;
printf("\%d", i);
```

Operadores de Incremento e Decremento de Variáveis

O operador "DirVarIncDec" insere um operador de decremento (−−) e de incremento (++) em cada referência a uma 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.

Operadores de Inclusão de Operadores Unários

Esses operadores de mutação incluem operadores unários em cada uso de variável ou constante. O operador "DirVarAriNeg" inclui o operador de negação aritmético em cada referência a variáveis de interface, o operador "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".

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 programa da Figura 2 o operador "RetStaDel" exclui o comando "*return a*" e o operador "RetStaRep" não gera mutantes.

Operadores de Cobertura de Código

Esses operadores são do tipo instrumentado. 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. O segundo, "CovAllEdge", garante a cobertura de todos os desvios. Essa cobertura é conseguida incluindo-se no código do mutante uma

função que ao ser executada força o mutante a ser distinguido. Assim, um caso de teste para distinguir o mutante tem que executar o ponto onde a função foi incluída (um bloco ou um desvio). A Tabela 3 mostra os pontos do programa da Figura 2 onde essas funções são inseridas.

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.

Tabela 3: Exemplo de mutantes para operadores de cobertura

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

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. Os mutantes gerados pelos operadores desse grupo para o programa da Figura 2 são mostrados na Tabela 4.

Tabela 4: Exemplo de mutantes para operadores do Grupo II

Operador	Mutante	Operador	Mutante
ArgRepReq	c = s(0, b)	ArgAriNeg	c = s(-d, b)
	c = s(1, b)	ArgLogNeg	c = s(!d, b)
	c = s(-1, b)		c = s(d, !b)
	c = s(MININT, b)	ArgBitNeg	c = s(~ d, b)
	c = s(MAXINT, b)	FuncCalDel	c = 0
ArgIncDec	c = s(d+1, b)		c = -1
	c = s(d, b+1)		c = 1
ArgSwiDif	c = s(b,d)		c = MININT
ArgDel	c = s(d)		c = MAXINT
	c = s(b)		

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".

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.

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 tipos compatíveis. Outro operador similar é "ArgSwiDif" que faz o mesmo tipo de troca, porém entre argumentos de tipos diferentes.

O operador "ArgSwiAli" não gera mutantes para o programa da Figura 2 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.

Operador de Eliminação de Argumento

Esse operador, "ArgDel", elimina cada um dos argumentos da chamada. Os mutantes desse operador, criados para o programa exemplo seriam "$c = s(d)$" e "$c = s(b)$". Sua aplicação está sujeita às mesmas restrições feitas ao operador "ArgSwiDif" pois pode gerar os mesmos tipos de erros sintáticos nos mutantes.

Operadores de Inclusão de Operadores Unários

O operador "ArgAriNeg" acrescenta o operador de negação aritmético antes de cada argumento, operador de mutação "ArgLogNeg" acrescenta o operador de negação lógica (!) e o operador "ArgBitNeg" acrescenta o operador de negação bit a bit (~).

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 4 dá exemplos de mutantes desse tipo.

4 Análise Teórica do Critério Mutação de Interface

Uma das maneiras de se avaliar um critério de teste é através da análise de inclusão com outro critério conhecido. Dados dois critérios C_1 e C_2, diz-se que C_1 inclui C_2 se para qualquer programa **P**, todo conjunto de teste **T** C_1-adequando é também C_2-adequado.

Wong [12] mostrou analiticamente que no nível de unidade o teste de mutação e o critério todos-usos de Rapps e Weyuker [10] são incoparáveis segundo a relação de inclusão.

Mostrou também através de estudos empíricos que na prática o teste de mutação tende a incluir os critério de fluxo de dados [12].

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 [6], ou seja, mostra-se que eles são imcomparáveis.

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 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 usado. Exige também que seja exercitado pelo menos um caminho livre de definição que vai de cada ponto em g onde v é definido 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 pelo menos um caminho que "atravesse" g sem definição de v (e consequentemente de u) seja exercitado, caso existam tais caminhos. Por exemplo, dado o programa da Figura 3, definem-se as seguintes associações definição-uso: $<f_1,(g_1,g_2)>$, $<f_1,(g_1,g_3)>$, $<f_1,g_2>$, $<f_1,f_3>$ e $<g_2,f_3>$.

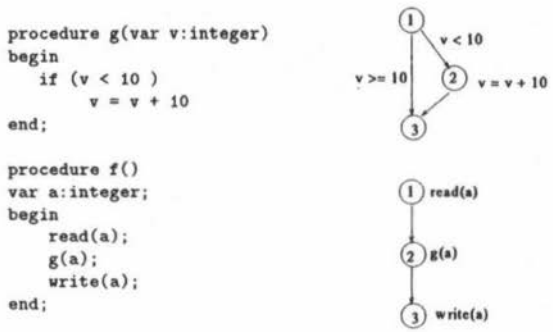


Figura 3: Programa para exemplificar associações requeridas pelo critério de Harrold e Soffa

Pode-se então estabelecer o seguinte teorema:

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

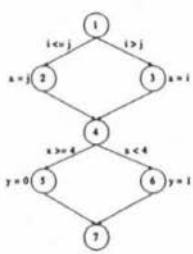
Para demostrar esse teorema deve-se mostrar que 1) o critério MI não inclui o critério H&S; e 2) o critério H&S não inclui o critério MI. Para tal, considera-se o programa **P** do qual fazem parte as unidades f e g, de forma que f faz chamadas a g e toda passagem de parâmetros entre elas é feita por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

[2]Essa 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.

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 um parâmetro formal v. A única operação realizada em g é a atribuição de um valor (uma definição) de v. O critério de H&S exige que pelo menos um caminho livre de definição que vai dessa definição até cada ponto f_{i1}, ..., 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 o comando onde v foi definida ser executado (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 produz 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. Porém, 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. Assim, mostrou-se que o critério MI não inclui o critério de H&S.

A segunda parte da demostração 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 4. Para ele são definidas as seguintes associações def-uso a serem cobertas: $<f_1,(g_1,g_2)>$, $<f_1,(g_1,g_3)>$, $<f_1,g_2>$, $<f_1,g_3>$, $<g5,f_3>$, $<g_6,f_3>$. Para satisfazer o critério de H&S os seguintes casos de teste podem ser usado: (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.

```
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
```

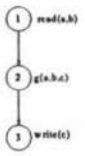


Figura 4: Programa que mostra que H&S não inclue MI

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 [1]. Note-se que esses casos especiais apresentam res-

trições 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 casos, como por exemplo, uma única definição de uma variável usada posteriormente como argumneto 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.

5 Estudos Empíricos

Os operadores de mutação discutidos na Seção 3 foram implementados numa ferramenta chamada *PROTEUM/IM* [1]. A disponibilidade da ferramenta permitiu que se conduzisem estudos empíricos onde avaliou-se a aplicação do critério quanto à efetividade em revelar erros e quanto ao custo. O primeiro experimento [3] 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-dequados. Nesse mesmo experimento foram utilizados critério alternativos, procurando reduzir-se 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. Ainda no experimento do *Sort*, comparou-se a efetividade da Mutação de Interface com o teste randômico, podendo concluir-se que o critério proposto apresenta-se sensivelmente mais efetivo que o teste randômico. O mesmo acontece para os critérios MI-alternativos.

O segundo experimento [1] foi realizado utilizando-se um programa chamado *SPACE*. Esse programa foi desenvolvido para a agência espacial italiana para a descrição e cálculo de antenas. Durante o processo de desenvolvimento desse programa foram encontrados vários defeitos. No experimento iniciou-se o teste do programa 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 uma estratégia incremental com o conjunto de operadores de mutação propostos neste trabalho. Com um conjunto 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 utilizados 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.

Para que se possa ter uma idéia do custo de aplicação dos operadores aqui propostos, a Tabela 5 mostra o número de mutantes utilizados para o programa *Sort* e para o programa *SPACE* (versão sem defeitos implantados). A numeração dos operadores representa a seqüência utilizada na estratégia incremental de geração de mutantes para o programa *SPACE*.

Tabela 5: Número de mutantes gerados para os programas *SPACE* e *Sort*

Operador	*Sort*	*SPACE*	Operador	*Sort*	*SPACE*
1- Grupo II	123	447	2- DirVarRep	744	1068
3- DirVarIncDec	240	700	4- DirVarxxxNeg	174	240
5- RetSta	22	186	6- IndVarRep	575	1305
7- IndVarIncDec	230	484	IndVarxxxNeg	201	699

6 Conclusão

A Mutação de Interface é um critério de teste interprocedimental para avaliação e seleção de conjuntos de teste baseado em mutações que tem como base a idéia de efetuarem-se no programa em teste pequenas alterações que criem erros de integração. Tais erros são caracterizados, de acordo com um modelo de falhas apresentado neste artigo, por valores incorretos sendo trocados além dos limites das unidades, através de suas interfaces. Essas alterações são definidas por operadores de mutação. O projeto de tais operadores é parte fundamental para a definição de critérios baseados em mutação pois em última análise são esses operadores que definem as características do critério.

Este artigo apresenta um conjunto de operadores para o critério interprocedimental Mutação de Interface. No projeto desses operadores procurou-se elaborar um conjunto que possa exercitar de maneira completa todos os tipos de interações entre duas unidades. Como resultado obteve-se um conjunto de 33 operadores que, como mostram estudos empíricos, possuem um alto grau de efetividade em revelar defeitos. Embora a utilização desse conjunto completo de operadores possa elevar excessivamente o custo da aplicação do critério devido ao grande número de mutantes que podem ser gerados, mostrou-se também que através da parametrização dos operadores pode-se ajustar o teste a restrições de custo e ainda assim obterem-se ótimos resultados em termos de efitividade.

Novos experimentos devem ainda ser conduzidos, procurando caracterizar de maneira mais precisa o comportamento de cada operador de mutação identificando um conjunto de operadores "essenciais" que minimizem o custo de aplicação do critério sem levar a um decréscimo na efetividade em revelar erros. Esses estudos devem também propor uma estratégia incremental de aplicação dos operadores – utilizando as características de parametrização disponíveis – que permita ao testador adequar o critério a restrições de custo e a requisitos de um plano de teste.

Esse artigo mostrou ainda a incomparabilidade do critério Mutação de Interface com o critério interprocedimental de Harrold e Soffa [6]. Esse resultado 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 um estratégia de teste desde o nível de unidade até os níveis mais altos, utilizando critérios de fluxo de dados e baseados em mutação.

Referências

[1] M. E. Delamaro. *"Mutação de Interface: Um Critério de Adequação Inter-procedimental para o Teste de Integração"*. Tese de Doutorado, IFSC - USP, São Carlos - SP, junho 1997.

[2] M. E. Delamaro, J. C. Maldonado e A. P. Mathur. "Integration Testing Using Interface Mutation". *Anais do VII International Simposyum of Software Reliability Engineering (ISSRE)*, New York - NY - EEUU, novembro 1996.

[3] M. E. Delamaro, J. C. Maldonado e A. P. Mathur. "Interface Mutation: An Approach to Integration Testing". submetido a uma revista, julho 1997.

[4] 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.

[5] 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.

[6] M. J. Harrold e M. L. Soffa. "Selecting and Using Data for Integration Test". *IEEE Software*, 8(2):58–65, março 1991.

[7] U. Linnenkugel e M. Müllerburg. "Test Data Selection Criteria for (Software) Integration Testing". *Anais da I International Conference on Systems Integration*, pp 709–717, Mornstown - NJ - EEUU, abril 1990.

[8] J. C. Maldonado. *Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software*. Tese de doutorado, DCA/FEE/UNICAMP, Campinas - SP, julho 1991.

[9] A. J. Offutt. "Coupling Effect: Fact or Fiction". *Anais do III Symposium on Software Testing, Analysis, and Verification*, pp 131–140, Key West - FL - EEUU, dezembro 1989.

[10] S. Rapps e E. J. Weyuker. "Data Flow Analysis Techniques for Program Test Data Selection". *Anais da VI International Conference on Software Engineering*, pp 272–278, Tokio - Japão, setembro 1982.

[11] S. Rapps e E. J. Weyuker. "Selecting Software Test Data Using Data Flow Information". *IEEE Transactions on Software Engineering*, SE-11(4):367–375, abril 1985.

[12] W. E. Wong. *"On Mutation and Data Flow"*. Tese de doutorado, Departamento de Ciência da Computação, Purdue University, W. Lafayette - IN - EEUU, dezembro 1993.

[13] W. E. Wong, J. C. Maldonado, M. E. Delamaro e A. P. Mathur. "Constrained Mutation in C Programs". *Anais do VIII Simpósio Brasileiro de Engenharia de Software*, pp 439–452, Curitiba - PR, outubro 1994.