Uma abordagem para o ensino de teste estrutural baseada na integração dos testes caminho básico, todos-usos e análise de mutantes

Daniel Lucas Santana Santos[1], Marcos Rogério Pinheiro Sousa[2], Sérgio Fred Ribeiro Andrade[3]

Ciência da Computação - Departamento de Ciências Exatas e Tecnológicas -
Universidade Estadual de Santa Cruz (UESC)
45662-999 – Ilhéus – BA – Brasil

lucasdlss@gmail.com[1], mrpsousa@outlook.com[2], sergiof@uesc.br[3]

Abstract. *This paper presents a new approach to the application of structural software testing, the integration of three techniques for employment in the teaching-learning process and the objective of showing the student a way to reduce the amount of program mutants and their generation process. Therefore, an example algorithm used and a script of the techniques used was applied, comparing the result with a metric found in the literature. The approach showed a significant reduction of 288 mutants by the traditional method to 112 mutants by the presented method, with no apparent loss of flows for measurement and acceptance in the teaching-learning process.*

Resumo. *Este trabalho apresenta uma nova abordagem da aplicação de teste estrutural de software, na integração de três técnicas para emprego no processo ensino-aprendizagem e no objetivo de mostrar ao discente uma forma de reduzir a quantidade de mutantes de programa e seu processo de geração. Para tanto, foi utilizado um algoritmo exemplo e aplicado um roteiro de utilização das técnicas abordadas, comparando-se o resultado com uma métrica encontrada na literatura. A abordagem mostrou uma redução significativa de 288 mutantes pelo método tradicional para 112 mutantes pelo método apresentado, sem perda aparente de fluxos para medição e com aceitação no processo ensino-aprendizagem.*

1. Introdução

As atividades de teste de software, seja teste de sistema, teste de integração ou teste estrutural [Delamaro e Maldonado 2017], buscam garantir a qualidade do software. Segundo [Pressman e Maxim 2016], a qualidade do software é inerente à exigência dos requisitos funcionais, das especificações explicitamente descritas e do desempenho estabelecido.

Um motivo cabível para aplicação de testes de software está indicado em [Howden 1976], o qual observa que no processo de construção de software os erros de semântica e lógicos na maioria das vezes ocorrem involuntariamente. Apesar do uso de métodos consistentes, ferramentas de suporte apropriadas e profissionais treinados.

Entre as diversas fases do teste estrutural, o teste de unidade é realizado quando se conhece as funções ou métodos do programa. São destacados aqui, o Teste do Caminho Básico (TCB) baseado num Grafo de Fluxo de Controle (GFC) [McCabe 1976], o Teste Baseado no Grafo do Fluxo de Dados (GFD) [Hecht 1977] e a Análise de Mutantes (AM) [Acree et al. 1979].

O primeiro teste resulta do grafo de um programa para evidenciar os possíveis fluxos e exercitar as estruturas de decisão na formação dos caminhos independentes, os quais contêm pelo menos uma nova aresta visitada, como também, para auxiliar no conhecimento da complexidade ciclomática que define a quantidade de casos de teste. O segundo, seleciona caminhos a partir do grafo de controle, desde a definição de variáveis até os efetivos usos computacionais ou por predicados lógicos, no propósito de conhecer caminhos não executáveis. O terceiro, Análise de Mutantes, emprega transformações sintáticas no código obtendo diversos derivados para comparação das saídas entre o programa original e seus mutantes, no objetivo de encontrar o erro ou a probabilidade de erros [Frankl e Weyuker 1988].

Essas três técnicas são descritas na literatura isoladamente, sem aplicação integrada ou complementar. Segundo [Pressman e Maxim 2016] é possível combinar técnicas de testes em um só caso de teste. Para [Barbosa et al. 2005] as técnicas de teste estrutural devem ser empregadas de forma complementares, portanto, devem ser utilizadas explorando a estratégia da boa qualidade, eficácia e baixo custo da aplicação. Outros autores têm manifestado nesse mesmo raciocínio, a exemplo de [Beizer 1990, Rocha et al. 2001, Vicenzi et al. 2007].

A falta da integração das técnicas de teste de unidade, possibilita morosidade no emprego dos procedimentos de teste estrutural, principalmente para iniciantes na engenharia de software, ou no entendimento da integração dessas técnicas no processo ensino-aprendizagem.

Por outro lado, alguns problemas de desempenho são apresentados quanto à aplicação prática dessas técnicas. Em [Wong e Maldonado 1995] é citado o alto custo da complexidade $O(n^2)$ para análise de mutantes, com um elevado número de variantes do programa testado, tornando o procedimento oneroso e demorado. Em [Rapps e Weyuker 1985], é dito que essas técnicas que utilizam somente todos-caminhos como teste baseado no fluxo de controle e no fluxo de dados não garantem que todos os erros sejam detectados, pois nos laços de repetições podem ter grande quantidade de fluxo ou um número infinito de caminhos a depender dos casos de teste e, induzir a parada dos testes quando a saída estiver correta mas a entrada indevida.

Para estudantes que precisam executar esses testes sem automatização, a quantidade de mutantes e caminhos para exercitar é muito grande e torna o trabalho demorado, por isso estudos indicam a redução de mutantes, a exemplo das técnicas Mutação Aleatória, Mutação Restrita e Mutação Seletiva, entre outra, que buscam selecionar subconjuntos de mutantes, porém com perda de desempenho [Barbosa et al. 2005].

Outra dificuldade, é que não existe uma métrica de consenso para saber o número de mutantes necessários por programa testado. Em [Wong e Maldonado 1995], pode ser feito um cálculo para conhecer aproximadamente a quantidade de mutantes de

acordo com o número de predicados e todos-usos. Por exemplo, para um programa com 17 predicados e 119 caminhos todos-usos são necessários 916 mutantes, para outro com 16 predicados e 63 caminhos todos-usos são 627 mutantes e para 12 predicados e 66 caminhos todos-usos são necessários 510 mutantes, para um teste confiável. Significa que, aproximadamente, para cada caminho todos-usos no limite superior, seja necessária uma quantidade muito grande de mutantes por variáveis, considerando alterações em operadores e operandos.

Esse é dos maiores problemas para a aplicação da técnica AM, pois está relacionado ao seu alto custo de aplicação, uma vez que o número de mutantes gerados, mesmo para pequenos programas, pode ser muito grande, exigindo um tempo de execução muito alto, podendo chegar a um problema indecidível.

Desta forma, o objetivo do presente trabalho é a introdução de uma abordagem para integrar os testes e inferir caminhos mais curtos com menos predicados para gerar menor quantidades de mutantes e assim, mostrar ao aprendiz o processo de geração de mutantes com redução de suas quantidades necessárias.

2. Materiais e Métodos

A metodologia considerou uma integração das três técnicas de testes de unidades referidas, visando diminuir o custo da aplicação do teste Análise de Mutantes, reduzindo a quantidade de programas derivados do principal de acordo com os caminhos todos-usos, por variável.

O TCB resulta num GFC representante de um programa para evidenciar os possíveis fluxos e exercitar as estruturas de decisão na formação dos caminhos independentes, os quais contêm pelo menos uma nova aresta visitada, como também para auxiliar no conhecimento da complexidade ciclomática e da quantidade de casos de teste.

Um programa é representado num fluxograma de algoritmo que pode ser convertido num GFC, ou seja, um grafo orientado $G = (V, E, n, k)$. Onde, cada nó V representa um procedimento (declaração, atribuição, operação aritmética e afins) e uma decisão (controle de fluxo para seleção ou repetição), que são associados através de arestas E, que é um par (n, m) de V que representa sequência de controle de n para m. O grafo resultante do programa é um grafo orientado com um único nó de entrada $n \in V$ e um único nó de saída $k \in V$. Um caminho ou fluxo no grafo é uma sequência de nós (n_1, n_2, n_3, ..., n_k), sendo $k \geq 2$.

Um caminho simples é um fluxo que contém todos os nós sem possibilidade de retorno, sendo todos os nós distintos, também chamado de caminho livre de laço. Um caminho completo contém n_1 na entrada e n_k como saída. Um caminho independente é um fluxo completo que contém uma nova aresta não percorrida anteriormente e não composto por combinação de fluxos antes visitados [McCabe 1976].

Um caminho independente é executável quando existem variáveis de entradas definidas, ou seja, inicializadas no código ou de *input* externo. Observa-se a possibilidade de não haver nenhuma entrada para uma variável numa sequência de procedimentos em um determinado fluxo do grafo, fazendo com que o caminho seja não

executável, essa situação é conhecida como caminho livre de definição [Frankl 1987, Rapps e Weyuker 1985].

Em um nó que contém um fluxo condicional, seleção ou repetição, é chamado de nó predicado *(P)* caracterizado por duas ou mais arestas de saídas. O nó *P* é aplicado para o cálculo do teste da complexidade ciclomática do grafo *V(G)* através de eq. 1.

$$V(G) = P + 1 \qquad (1)$$

Da mesma forma, pode-se conhecer o *V(G)* pela quantidade de nós e arestas visitadas em todos os caminhos do programa, dada pela eq. 2.

$$V(G) = E - V + 2 \qquad (2)$$

O resultado *V(G)* denota a quantidade de casos de teste que devem ser realizados no limite superior para o número de caminhos independentes do programa [Pressman e Maxim 2016].

A partir do GFC, o grafo contempla as interações entre as definições e referências de variáveis (*def*), o uso computacional (*c-uso*) – nas operações e o uso predicados (*p-uso*) – nas proposições lógicas, considerando Todas-Definições e Todos-Usos de dados *(def-uso)* no GFD, com o objetivo de conhecer os caminhos livres de definições [Rapps e Weyuker 1985].

Definição (*def*) ocorre quando uma variável recebe um valor atribuído na primeira ocorrência e em ocorrências posteriores quando altera o valor, por exemplo: *int* ***var*** *= 0; ...* ***var*** *= m + 1;* . Uso Computacional (*c-uso*) ocorre quando a variável é utilizada em uma computação, como a operação aritmética, saída para impressão e envio de parâmetros, por exemplo: *m =* ***var*** *+ 1;* . Uso predicativo (*p-uso*) ocorre quando a variável é utilizada em uma condição lógica, por exemplo: *if (****var*** *!= "\n") {...}* .

O critério *def-uso* para a variável *x* corresponde a um par de nós do grafo (*n*, *m*), de modo que *x* está em *def* (*n*). A definição de *x* em *n* atinge *m*, *x* está *uso* em (*m*). Assim, o valor atribuído a *x* em *n* é usado em *m,* computacional ou predicativo, uma vez que a definição atinge *m*. O valor não é nulo ao longo do caminho *n ... m*. As notações *(n, m, var)* e *(n, (m, k), var)* indicam que a variável *var* é definida no nó *n* e existe um uso computacional de *var* no nó *m* ou um uso predicativo de *var* no arco *(m, k)*, respectivamente.

A AM é uma técnica que visa análise de derivados de um programa *P*, o qual sofre alterações semânticas ou lógicas (basicamente na substituição de operandos e operadores relacionais, lógicos e aritméticos), dando origem aos derivados $p_1, p_2,...,p_k$ denominados mutantes de *P*, diferenciados de *P* apenas pela mudança sintática. Para cada mutante *p*, deve-se executar um conjunto de casos de teste *T* no propósito da detecção de erros no código. Se *p* apresentar resultados incorretos em comparação à saída de *P,* o mutante é chamado de *morto*, então um erro foi encontrado e este caso de teste termina. No contrário, o programa ainda pode conter erros, que o conjunto *T* não conseguiu revelar, então o mutante é equivalente, considerado *vivo,* e outro caso de teste deve ser feito.

Com base nesses critérios, realizou-se uma integração entre as três técnicas citadas, no propósito de reduzir a quantidade de caminhos *def-uso* para aplicação de análise mutantes num determinado programa exemplo.

Foram executadas as seguintes tarefas metodológicas:

1) Elaboração de um grafo (GFC) do programa para o TCB;
2) Determinação do número de nós, arestas e predicados para cálculo da complexidade ciclomática;
3) Apuração do conjunto de caminhos independentes, pela métrica tradicional, para indicar quantidade de casos de teste para cada variável e aplicação em outras técnicas de testes;
4) Representação do fluxo de dados num GFD, indicando *def-uso,* através das definições *def, c-uso* e *p-uso*, para cada nó e para cada variável, no propósito de conhecer caminhos sem definições;
5) Indicação para cada nó de *def-uso* na notação *(n,m ,var)* uso computacional e *(n,(m, k),var)* uso de predicado, para facilitar a construção de subcaminhos por variável em uso, visando diminuição de casos de teste;
6) Comparação de cada subcaminho com os demais para verificar sobreposições e descartar o menor fluxo que foi contemplado pelo maior subcaminho, permanecendo um único subcaminho para cada *def-uso* por variável;
7) Separação dos subcaminhos *def-uso* por variável para aplicação em casos de teste com AM;
8) Indicação da quantidade de mutantes para cada subcaminho por variável;
9) Verificação da redução de mutantes comparando-se o método tradicional de AM com esta abordagem de integração das técnicas e redimensionamento dos caminhos independentes em subcaminhos independentes.

Para comparação entre os subcaminhos determinados pelo *def-uso* e o método tradicional de AM, foi adotada a métrica descrita em [Wong e Maldonado 1995], onde há uma relação entre a quantidade de predicado no programa, a quantidade de fluxo em Todos-Usos e a quantidade aproximada de mutantes para casos de teste.

3. Resultados

Com base no exposto, foi utilizada uma função simples que recebe dois inteiros e calcula a potenciação, mostrada na Figura 1(a). Os GFC e GFD estão representados na Figura 1(b). A escolha desse algoritmo foi pela simplicidade da apresentação e entendimento. A função tem 3 predicados, 11 nós e 13 arestas.

O resultado do cálculo da complexidade ciclomática *V(G)* do grafo de fluxo de controle na Figura 1(b) é igual a 4, pois são apresentados 3 nós predicados mais 1 constante (*V(G) = 3 + 1*), como também, é definido por 13 arestas, menos 11 nós, mais 2 constantes (*V(G) = 13 - 11 + 2*).

Isso pressupõe que existem 4 caminhos independentes para formação de casos de teste no limite superior. Os caminhos independentes resultantes do GFC para dados genéricos, na notação pela métrica tradicional, são:

1) 1,2,3,4,6,7,8,6,9,10,11
2) 1,2,3,5,6,7,8,6,9,10,11
3) 1,2,3,4,6,9,10,11
4) 1,2,3,5,6,9,10,11

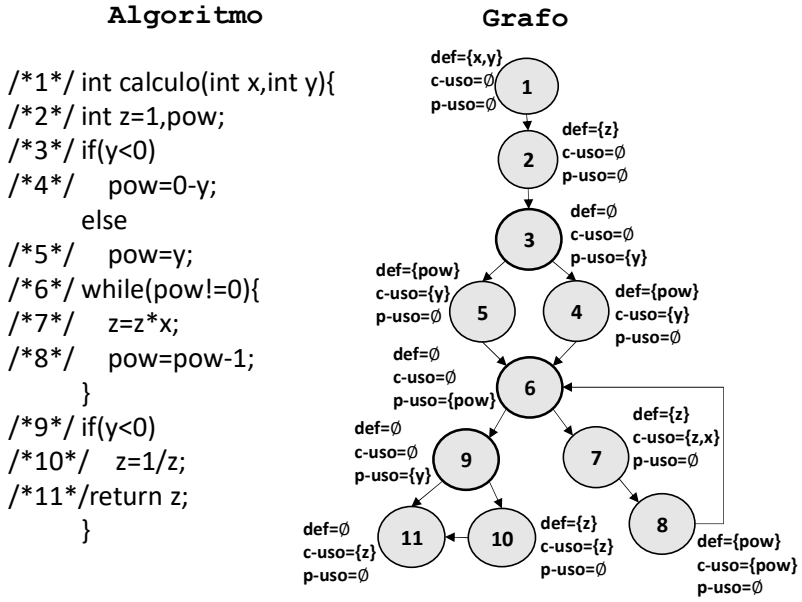

Figura 1. (a) Algoritmo do programa estudado; (b) GFC/GFD do programa.

A Tabela 1 mostra os nós com respectivas variáveis *(nó, var)* do grafo na Figura 1(b), as representações das definições das variáveis *(def)*, uso computacional *(c-uso)* e uso em predicados *(p-uso),* as notações *def-uso* para todas as variáveis no conjunto Todos-Usos para formação dos subcaminhos independentes, de acordo com as variáveis de influência.

Na coluna "Variáveis para Análise de Mutantes" estão as variáveis em *def-uso* de acordo com os subcaminhos. Cada subcaminho corresponde a um fluxo *def - c-uso* e/ou *p-uso* e a coluna "Ordem do subcaminho incluído" indica a ordem do subcaminho que lhe representou inclusive a sua variável, na fase de comparação.

Tabela 1. Subcaminhos resultantes para análise de mutantes.

Ordem	*(nó,var)*	Definição – c-uso		Definição – p-uso		Subcaminhos para casos de teste	Ordem do subcaminho incluído	Variáveis para Análise de Mutantes
		def -> c-uso	*(n, m, var)*	*def -> p-uso*	*(n,(m, k),var)*			
1	(1,x)	{7}	(1,7,x)	Ø	Ø	1,2,3,4,6,7	-	x,z,pow
						1,2,3,5,6,7	-	x,z,pow
2	(1,y)	{4}	(1,4,y)	{(1,3),(1,9)}	(1,(3,4),y)	1,2,3,4	2	-
					(1,(3,5),y)	1,2,3,5	2	-
		{5}	(1,5,y)		(1,(9,10),y)	1,2,3,4,6,9	-	y,pow
					(1,(9,11),y)	1,2,3,5,6,9	-	y,pow
3	(4,pow)	{8}	(4,8,pow)	{4,6}	(4,(6,7),pow)	4,6,7,8	-	pow
					(4,(6,9),pow)	4,6,9	2	-
4	(5,pow)	{8}	(5,8,pow)	{5,6}	(5,(6,7),pow)	5,6,7,8	-	pow
					(5,(6,9),pow)	5,6,9	2	-
5	(8,pow)	{8}	(8,8,pow)	{8,6}	(8,(6,7),pow)	8,6,7	-	pow
					(8,(6,9),pow)	8,6,9	7	-
6	(2,z)	{7}	(2,7,z)	Ø	Ø	2,3,4,6,7	1	-
						2,3,5,6,7	1	-
		{10}	(2,10,z)			2,3,4,6,9,10	-	z
						2,3,5,6,9,10	-	z
		{11}	(2,11,z)			2,3,4,6,9,11	-	z
						2,3,5,6,9,11	-	z
7	(7,z)	{7,10}	(7,10,z)	Ø	Ø	7,8,6,9,10	-	z,pow
		{7,11}	(7,11,z)			7,8,6,9,11	-	z
8	(10,z)	{11}	(10,11,z)	Ø	Ø	10,11	-	z

Os subcaminhos que vigaram foram resultantes do confronto com todos os outros subcaminhos, observando se fazia parte do subconjunto. Quando incluído em algum outro subcaminho era descartado, caso contrário era efetivamente eleito para o teste de AM.

Para verificação se houve redução do número de mutantes por essa abordagem, utilizou-se a métrica em [Wong e Maldonado 1995]. Onde foram calculadas as médias das quantidades dos mutantes gerados, dos predicados e *def-uso's* de 8 programas indicados na Tabela 2.

Tabela 2. Métrica sobre mutantes por predicados e *def-uso*.

Programa	Predicados	*def-uso*	Mutantes
1	17	119	916
2	16	63	627
3	12	66	510
4	18	53	938
5	5	37	490
6	8	53	415
7	6	22	251
8	8	54	446
Média	**12**	**59**	**575**

Observa-se na Tabela 3 a comparação entre os mutantes gerados de acordo com os caminhos independentes gerais para todas as variáveis, calculados de acordo com o TCB, e os mutantes calculados para os subcaminhos em conformidade com esta abordagem de integração das três técnicas consideradas.

As colunas “Predicado” e “*def-uso*” foram apuradas pelo GFC/GFD e a “Mutante por variável” foi calculada por regra de três composta entre os valores das duas colunas citadas e as médias apuradas na Tabela 2.

Notou-se uma redução significativa de 288 mutantes calculados pelas técnicas tradicionais para 112 mutantes pelo método apresentado, sem perda aparente de fluxos para medir ou subcaminhos sem definição não vislumbrados.

Tabela 3. Mutantes calculados por TCB e calculados por subcaminhos *def-uso*.

Ordem	Predicado	*def-uso*	Mutante por variável	Variáveis	Total de Mutante
Cam. 1	3	9	21	x,y,z,pow	84
Cam. 2	3	9	21	x,y,z,pow	84
Cam. 3	3	6	15	x,y,z,pow	60
Cam. 4	3	6	15	x,y,z,pow	60
			Total pelo cálculo tradicional		**288**
Subcam. 1	2	4	7	x,z,pow	21
Subcam. 1	2	4	7	x,z,pow	21
Subcam. 2	3	4	10	y,pow	20
Subcam. 2	3	4	10	y,pow	20
Subcam. 3	1	2	2	pow	2
Subcam. 4	1	2	2	pow	2
Subcam. 5	1	2	2	pow	2
Subcam. 6	3	1	3	z	3
Subcam. 6	3	1	3	z	3
Subcam. 6	3	1	3	z	3
Subcam. 6	3	1	3	z	3
Subcam. 7	2	2	4	z,pow	8
Subcam. 7	2	2	4	z	4
Subcam. 8	0	1	0	z	0
			Total por esta abordagem		**112**

4. Considerações finais

Este artigo apresentou uma abordagem sobre aplicação de testes estruturais fazendo uma integração de três técnicas, mostrando a redução de uso de casos de teste e com a preocupação de ilustrar para o aprendiz o processo para auxílio na garantia de qualidade de software.

Por essa nova abordagem, observou-se redução na quantidade de mutantes derivados do programa original e consequente diminuição dos casos de teste necessários.

Esta abordagem foi debatida em classe da disciplina engenharia de software, de um curso de ciência da computação, com aceitação e aprendizado pelos discentes. Gerando debates e seminários que motivaram à iniciação científica e novas pesquisas.

Para o futuro pretende-se avançar com esse conteúdo em programas pedagógicos de novas turmas e com novos algoritmos, para então, gerar condições de pesquisas qualitativas através de questionários e outras técnicas estatísticas. Pretende-se também o desenvolvimento de uma ferramenta que automatize essa prática, com licença do software e código para domínio público, no propósito da disseminação da abordagem e melhor auxílio no processo ensino-aprendizagem.

Referências

Acree, A. T., Budd, T. A., DeMillo, R. A., Lipton, R. J., and Sayward, F. G. (1979). “Mutation Analysis” (No. GIT-ICS-79/08). Georgia Inst. of Tech Atlanta School of Information and Computer Science, Atlanta-Georgia, USA.

Barbosa, E. F., Maldonado, J. C., Vincenzi, A. M. R. e Delamaro, M. (2005). “Teste estrutural e de mutação no contexto de programas OO”. In: IV Escola Regional de Informática de Minas Gerais (ERI-MG 2005), texto didático. Belo Horizonte, MG.

Beizer, Boris (2003). Software testing techniques. Dreamtech Press.

Delamaro, M., Jino, M. e Maldonado, J. C. (2017). Introdução ao teste de software. Elsevier Brasil, 2ª. edição.

Frankl, Phyllis G. (1987). “Use of data-flow information for the selection and evaluation of software test data”. Doctoral Dissertation, New York University, NY, USA.

Frankl, P. G. and Weyuker, Elaine. J. (1988). “An Applicable Family of Data Flow Testing Criteria”, Journal IEEE Transactions on Software Engineering, Volume 14 Issue 10, USA, p. 1483-1498.

Hecht, Matthew S. (1977). Flow Analysis of Computer Programs Elsevier Science Inc. New York, NY, USA.

Howden, William E. (1976). "Reliability of the path analysis testing strategy." IEEE Transactions on Software Engineering 3, p. 208-215.

Maldonado, J. C. (1991). “Critérios potenciais usos: Uma contribuição ao teste estrutural de software”. Tese Doutorado, DCA/FEE/UNICAMP, Campinas-SP.

McCabe, Thomas J. (1976). "A complexity measure." IEEE Transactions on software Engineering 4, p. 308-320.

Pressman, R. e Maxim, B. (2016). Engenharia de Software, McGraw Hill Brasil, 8ª Edição.

Rapps, Sandra and Weyuker, Elaine J. (1985). “Selecting software test data using data flow information”. IEEE transactions on software engineering 4, p. 367-375.

Rocha, A. R. C. D., Maldonado, J. C. e Weber, K. C. (2001). Qualidade de software: teoria e prática. Prenttice Hall.

Vincenzi, A., Delamaro, M., Höhn, E., and Maldonado, J. C. (2007). “Functional, control and data flow, and mutation testing: Theory and practice”. In: Pernambuco Summer School on Software Engineering, p. 18-58. Springer, Berlin, Heidelberg.

Wong, W. E., Mathur, A. P. and Maldonado, J. C. (1995). “Mutation versus all-uses: An empirical evaluation of cost, strength and effectiveness”. In: Software Quality and Productivity, p. 258-265. Springer, Boston-MA, USA.