Assinatura do Teste Estrutural (*AtE*) - métrica baseada em teste fluxo de dados e análise de mutantes

Signature of the Structural Test (*SSt*) - metric based on data flow test and mutant analysis

Sérgio Fred Ribeiro Andrade
Departamento de Ciências Exatas e Tecnológicas (DCET)
Universidade Estadual de Santa Cruz (UESC)
Ilhéus – Bahia - Brasil
sergiof@uesc.br

RESUMO

O teste estrutural possibilita garantir a qualidade do software pela análise do código fonte. Nesse sentido, este trabalho apresenta um novo método de teste estrutural com a aplicação das técnicas teste fluxo de dados e análise de mutantes. Trata-se de um modelo aritmético intitulado Assinatura do Teste Estrutural (*AtE*), que recebe parâmetros como variáveis, operandos, operadores e comandos, determina uma métrica com objetivos de detectar possíveis erros semânticos e lógicos entre as definições de variáveis e seus usos, como também, reduzir os caminhos de testes no grafo de dados e os mutantes gerados. O resultado mostrou-se favorável à continuação da pesquisa com outros ensaios e direcionamento para construção de ferramenta apropriada.

Palavras chaves

Teste de software; Teste estrutural; Grafo fluxo de dados; Métrica de software

ABSTRACT

The structural test makes it possible to guarantee the quality of the software by analyzing the source code. In this sense, this paper presents a new method of structural testing with the application of data flow testing and mutant analysis techniques. It is an arithmetic model called Signature of the Structural Test (*SSt*), which receives parameters such as variables, operands, operators and commands, determines a metric with the objective of detecting possible semantic and logical errors between the definitions of variables and their uses, such as also, reducing the test paths in the data graph and the mutants generated. The result was favorable to the continuation of the research with other tests and guidance for the construction of an appropriate tool.

Keywords

Structural testing; Graph data flow; Software metrics.Teste estrutural; Grafo fluxo de dados; Métrica de software

CCS Concepts

Software and its engineering→Software creation and management→Software verification and validation→ Software defect analysis →Software testing and debugging.

1. INTRODUÇÃO

As atividades de teste de software, em especial de teste estrutural, buscam garantir a qualidade do software pela análise do código fonte [9]. Segundo Pressman e Maxim [24], 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 teste estrutural está indicado em Howden [19], o qual observa que no processo de implementação de código os erros de semântica ou lógicos, na maioria das vezes ocorrem involuntariamente, apesar do uso de métodos consistentes, ferramentas de suporte para depuração de sintaxe e semântica e profissionais treinados.

Entende-se como erro semântico um problema de significado, que ocorre mesmo quando uma sentença esteja sintaticamente correta, mas o resultado não é satisfatório, e esse erro pode ser encontrado numa análise de *parser*. Erro lógico ocorre quando o código está correto em sintaxe e semântica, mas após a execução o resultado não é o esperado.

Erros de semântica e lógicos podem ser tratados através de teste estrutural. Desta forma, são destacados neste trabalho os testes Fluxo de Dados (FD) [17, 18] – fluxo de dados é o caminho entre definição e uso de variáveis e Análise de Mutantes (AM) [1] – mutantes são derivados de um programa com alguma alteração sintática. Como também, é enfatizado o princípio "baseado em erro" [11], que utiliza informações sobre os tipos de erros mais frequentes no processo de codificação de software para derivar os requisitos de testes.

Especificamente, o teste FD é baseado no Grafo Fluxo de Dados (GFD) onde seleciona-se caminhos a partir do Grafo Fluxo de Controle (GFC), desde a definição de variáveis até os efetivos usos computacionais ou por predicados lógicos, no propósito de conhecer caminhos não executáveis. A AM emprega transformações sintáticas no código obtendo diversos derivados para comparação das saídas entre o programa original e seus mutantes, no objetivo de encontrar a probabilidade de erros [13].

Essas técnicas são descritas na literatura isoladamente, sem aplicação integrada ou complementar. Segundo Pressman e Maxim [24] é possível combinar técnicas de testes em um só caso de teste. Para Barbosa et al. [4] 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 [5, 26, 29].

Em Rapps e Weyuker [25], é dito que essas técnicas que utilizam tradicionalmente 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 ou vice-versa.

Em Wong e Maldonado [30] é citado o alto custo da complexidade *O(n2)* para AM, com um elevado número de variantes do programa testado, tornando o procedimento oneroso e demorado. Na execução desses testes a quantidade de mutantes e caminhos para exercitar é muito grande e torna o trabalho demorado por isso. Alguns estudos indicam a redução de mutantes, a exemplo das técnicas Mutação Aleatória, Mutação Restrita e Mutação Seletiva, entre outras, que buscam selecionar subconjuntos de mutantes [4].

Alguns trabalhos encontrados na literatura reportam soluções congêneres. Como exemplo de Buy et al. [6] que coletam as pré-condições numa relação entre valores de entrada e saída e combinam análise de fluxo de dados com um procedimento automático para realizar testes.

Foi apresentado por Girgis [14] um algoritmo genético para usar cadeias binárias de comprimento como um cromossomo para um caso de teste, e representar os valores das variáveis de entrada para verificar a saída esperada, numa razão entre o número de caminhos cobertos e o número total de caminhos de *def-uso*. E, Denaro et al. [10] usam um algoritmo genético semelhante para aumentar os conjuntos de testes iniciais com casos de testes baseados em fluxo de dados de sistemas orientados a objetos.

A proposta de Santelices et al. [27] é uma técnica de localização de falhas que usa diferentes critérios de cobertura para detectar declarações defeituosas em um programa, através de testes em instruções, ramificações e pares de *def-uso*. Em Alshahwan e Harman [2] é proposto uma técnica de fluxo de dados baseada em estado para aplicativos Web, que gera novas sequências de solicitações *http* para aprimorar os conjuntos de testes existentes em termos de cobertura e detecção de falhas.

Uma nova família de cobertura de testes foi introduzida por Hassan e Andrews [16], denominada cobertura multidimensional de passos, que registra a cobertura das tuplas dos ramos adotados. Já Grechanik e Devanla [15], propõem um modelo para detecção de erros que usa análise de fluxo de dados e geração de mutantes, aplicando operadores em instruções de componentes integrados nos caminhos do fluxo de dados. O trabalho de Chaim e De Araújo [7] sugere um algoritmo intitulado *Bitwise Algorithm*, que usa vetores de bits em operações para rastrear a cobertura do fluxo de dados para programas em Java.

Trabalhos de revisões integrativas da literatura em [8, 20, 21, 28] tratam sobre fluxo de dados, análise de mutantes e métricas de complexidade de software, e indicam análises dos diferentes aspectos de FD, AM e métricas de software em testes estruturais. Entretanto, não foram encontradas na literatura métricas que possibilitam detecção de erros semânticos ou lógicos, com uso de variáveis ou predicados lógicos, entre as fases de projeto de software e implementação de código.

Desta forma, o objetivo do presente trabalho é a introdução de uma nova abordagem de teste estrutural que aplica um modelo aritmético para calcular uma métrica intitulada de Assinatura do Teste Estrutural (*AtE*), a qual busca detectar possíveis erros semânticos e lógicos em unidades estruturais de código e validar a implementação de uma unidade de programa de acordo com o projeto de software. Aliado ao propósito da redução de caminhos para testes em fluxo de dados e redução da quantidade de mutantes gerados para testes.

2. MÉTODOS

Na presente metodologia foi considerada a integração das técnicas citadas de testes estruturais e adotados os seguintes princípios: a) testes em unidades menores para detecção de erros em menor tempo do que outras técnicas que possibilitam caminhos entre a definição e uso de variáveis; b) detecção de falhas na estrutura do código com localização de erros semânticos e lógicos em uso computacional e em predicado lógico; e, c) possibilidade de aplicação de mutantes em menor quantidade do que a técnica tradicional.

2.1 Fundamentação teórica

A notação, parâmetros e procedimentos adotados neste trabalho estão descritos a seguir.

Um programa é um conjunto finito de instruções sequenciais e seleção/repetição condicionais para resolver problemas com dados. É representado num fluxograma de algoritmo que pode ser convertido num GFC, ou seja, um grafo orientado $G = (V, E, n, k)$. Onde, cada nodo ou 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 $(n_1, n_2, n_3, ..., n_k)$, com $k \geq 2$, sendo que um caminho completo contém n_1 na entrada e n_k como saída, e, um subcaminho é um fluxo intermediário qualquer com $k \geq 2$ num caminho completo. Nesse sentido, quando um subcaminho abranger outros subcaminhos menores, terá prioridade para Casos de Teste (CT) em razão da sua amplitude para as variáveis testadas.

Em um nó que contém um fluxo condicional, seleção ou repetição, é chamado de nó predicado lógico (p) caracterizado por duas ou mais arestas de saídas.

Um caminho independente é um fluxo completo que contém uma nova aresta não percorrida anteriormente e não composto por combinação de fluxos antes visitados [22]. A quantidade de caminhos independentes é conhecida pelo cálculo da complexidade ciclomática C, através da fórmula $C= p+1$, onde p é a quantidade de predicados do programa [22, 24], e são executáveis quando existem variáveis de entradas definidas ou inicializadas no código.

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 [12, 21, 25].

A partir do GFC, indica-se as interações entre as definições e referências de variáveis (*def*), o uso computacional (*c-uso*) – nas operações e, o uso predicado (*p-uso*) – nas proposições lógicas, considerando Todas-Definições e Todos-Usos de dados *(def-uso)* no GFD [25].

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 predicado (*p-uso*), nas proposições onde a função booleana *p: x*→{verdadeiro, falso}, ou seja, quando a variável é utilizada em uma condição lógica, por exemplo: *if (**var** != 0) {...}*.

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* e *x* está *uso* em (*m*). Assim, o valor atribuído a *x* em *n* é usado em *m,* computacional (*def→c-uso*) ou predicado (*def→p-uso*). O valor não é nulo ao longo do caminho *n ... m*. As notações *(n, m, var)* e *(n, (m, k), bool, var)* indicam que a variável *var* é definida no nó *n* e existe um uso computacional de *var* no nó *m* ou um uso predicado de *var* no arco *(m, k)*, com a saída booleana em *bool*, respectivamente. [13, 21, 25].

A AM é uma técnica que visa análise de derivados de um programa *P*, o qual sofre mudanças (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 apenas por uma alteração sintática. Para cada mutante P_k, deve-se executar um conjunto de CT no propósito da detecção de erros no código. Caso a comparação entre *P* e P_k, no mesmo CT, tenha saída diferente, P_k é considerado morto e o teste acaba; caso o resultado seja igual, P_k é considerado vivo, e são equivalentes embora sintaticamente diferentes, e um novo caso de teste deve ser aplicado [1].

Neste trabalho, a utilização de AM justifica-se como uma garantia para detecção de erros, caso não sejam descobertos com as determinações iniciais de *AtE*. As alterações sintáticas nos mutantes são de acordo com Offutt et al. [23], conforme Tabela 1. A razão da escolha desses operadores recaí por serem usados nas estruturas em *c-uso* e *p-uso*, foco deste trabalho.

Tabela 1. Operadores utilizados em mutantes.

Sigla	Operador	Modificadores em mutantes
AOR	Aritmético	Substitui (+, -, *, /, +=, -=, *=, /=, ++, --)
LCR	Lógico	Substitui (&&, &, |, ||, !, true, false)
ROR	Relacional	Substitui (<, >, <=, >=, ==, !=)

2.2 Assinatura do teste estrutural

A métrica *AtE* gera um somatório nos subcaminhos de *def→c-uso* ou *def→p-uso,* através da Equação 1 - principal*,* que é composta pelas parcelas representadas pela Equação 2 – a qual representa uma métrica para *def→c-uso* e permite conhecer erros de semântica ou lógicos relacionados ao uso computacional, pela Equação 3 – que possibilita conhecer erros de semântica ou lógicos em predicados através da métrica em *def→p-uso*, e, pela Equação 4 – que representa uma métrica da saída esperada do programa de acordo com a entrada escolhida. O resultado em hexadecimal é denominado de Assinatura do Teste Estrutural (*AtE*), conforme Equação 1.

$$AtE = M_{(c\text{–}uso)} + M_{(p\text{–}uso)} + V_{(var)} \quad (1)$$

A fórmula geral *AtE* é composta por:

i) $M_{(c\text{-}uso)}$, na Equação 2, recebe valores em hexadecimal e determina uma métrica para o uso computacional *def→c-uso* de acordo com os comandos no código do subcaminho do programa para efetuar cálculo aritmético, recebimento e retorno de parâmetro, atribuição de valor, impressão de valores e correlatos.

$$M_{(c\text{-}uso)} = \sum_{i=1}^{N} \left(\sum_{j=1}^{M} w_j + \sum_{l=1}^{Q} d_l + \sum_{k=1}^{R} c_k \right)_i \quad (2)$$

Onde:

N = quantidade de usos computacionais (*def→c-uso)* para cálculo aritmético, recebimento e retorno de parâmetro, atribuição de valor, impressão de valores e correlatos, no subcaminho;

i = *i*-ésimo comando de uso computacional (*c-uso*)*,* no subcaminho;

w_j= *j*-ésima variável *w*, operando aritmético numa linha executável;

M = quantidade de variáveis numa linha executável;

d_l= *l*-ésimo operador aritmético, operador de atribuição, comandos de impressão, função/comando de recebimento de parâmetro, função de retorno de parâmetro *d*, numa linha executável;

Q = quantidade de operadores/comandos numa linha executável;

c_k = *k*-ésima constante *c*, numa linha executável;

R = quantidade de constantes numa linha executável;

i,j,k,l = (1,2,3,...,N).

ii) $M_{(p\text{-}uso)}$, na Equação 3, recebe parâmetros em hexadecimal e determina uma métrica para o uso de predicado lógico (*def→p-uso),* no código de estruturas de seleção (simples, composta e múltipla) e repetição.

$$M_{(p\text{–}uso)} = \sum_{n=1}^{S} \left(\sum_{m=1}^{T} o_m + \sum_{f=1}^{V} (a + q + b + r)_f + z \right)_n \quad (3)$$

Onde:

S = quantidade de predicados lógicos (*p-uso)* em estruturas de seleção/repetição no subcaminho;

n = *n*-ésima estrutura de seleção/repetição (*p-uso*) no subcaminho;

o_m = *m*-ésimo operador de conjunção, disjunção ou negação *o* em proposição lógica composta numa estrutura de seleção/repetição;

T = quantidade de operadores de conjunção, disjunção ou negação em proposição composta numa estrutura de seleção/repetição;

V = quantidade de proposições lógicas simples numa estrutura de seleção/repetição;

f = *f*-ésima proposição lógica simples numa estrutura de seleção/repetição;

a, b = operandos da proposição lógica simples numa estrutura de seleção/repetição;

q = operador relacional da proposição lógica simples numa estrutura de seleção/repetição;

r = resultado da proposição lógica simples numa estrutura de seleção/repetição;

z = resultado da proposição lógica composta numa estrutura de seleção/repetição;

$n,m,f = (1,2,3,...,N)$.

iii) Para $V_{(Var)}$, que também recebe parâmetros em hexadecimal, é um somatório de todas as variáveis no último nodo do programa testado, mais o resultado de saída das variáveis após execução do programa.

$$V_{(var)} = \sum_{g=1}^{X} v_g + \sum_{h=1}^{Y} s_h \qquad (4)$$

Onde:

X = quantidade de variáveis no último nodo executado do programa;

v_g = g-ésimo valores das variáveis v no último nodo executado do programa;

Y = quantidade de variáveis de saída resultantes após execução do programa;

s_h = h-ésimo valores de saída das variáveis de retorno, impressão, atribuição ou outra operação resultante s, após execução do programa;

$g,h = (1,2,3,...,N)$.

Para todas as equações: inexistindo quaisquer valores em parâmetros correspondentes no código do programa, o valor deve ser igual a zero; para evitar confusão e duplicidade entre o sinal negativo de um número e o operador de subtração, computa-se todos os números como reais não negativos, ou seja, conjunto R+ = {x ∈ R | x ≥ 0}; em estruturas de dados homogêneas e heterogêneas cada posição de memória ou variável são consideradas parcelas somatórias; para operadores aritméticos, relacionais, lógicos, conectivos e símbolos de atribuição, utiliza-se o valor em hexadecimal da tabela original [3]; para função/método de recebimento e retorno de parâmetros em uso computacional utiliza-se o somatório dos valores em hexadecimal; para valores de variáveis derivadas do tipo primitivo ou *string* ou de instanciação de objetos utiliza-se o somatório dos caracteres em hexadecimal.

2.3 Tarefas metodológicas

As tarefas metodológicas estão descritas a seguir e ilustradas na Figura 1:

(1) Implementar em manuscrito e validar, em tempo de projeto, o código do programa original *O* (unidade, método ou função), em uma linguagem formal compilada ou interpretada;

(2) Representar o GFD para *O*, e indicar para cada nó o *def-uso* na notação *(n,m ,var)* - para uso computacional (*c-uso)* e *(n,(m, k),bool, var)* – para uso de predicado (*p-uso)*;

(3) Implementar para processamento em máquina o programa de produção *P*, a partir do programa original *O,* em uma linguagem formal, compilada ou interpretada, conforme especificação do projeto;

(4) Identificar os subcaminhos do GFD, por variável, na notação *def→c-uso* e *def→p-uso*;

(5) Definir CT adequado para executar todos os nós do subcaminho do programa *O,* que deseja validar;

(6) Fazer Teste de Mesa (TM) em *O* com CT adequado para executar o subcaminho a ser testado, conhecer a evolução dos valores nas variáveis e verificar se a saída esperada está em conformidade com a entrada, de acordo com o projeto de software;

(7) Calcular pela Equação (1) as *AtE's* para o subcaminho a ser testado dos programas *O* e *P,* para comparação simples;

(8) Caso o resultado das *AtE's* sejam iguais, considera-se o programa *P* correto neste subcaminho para este CT (9);

(10) Caso o resultado das *AtE's* sejam diferentes, considera-se o programa *P* incorreto, neste subcaminho para este CT;

(11) Verificar erro detectado conforme divergência em *AtE*, para $M_{(c\text{-}uso)}$, $M_{(p\text{-}uso)}$ e/ou $V_{(Var)}$, e alterar código de *P* conforme programa *O*;

(12) Se necessário, em caso de não detecção de erro, aplicar mutantes P_k de *P*, com alterações sintáticas de AOR/LCR/ROR (Tabela 1), para o subcaminho testado. (13) Não sendo necessário este procedimento, segue fluxo;

(14) Aplicar novo CT e calcular novo TM para *P* ou P_k. Retornar ao passo (7) e substituir *P* por P_k se existir.

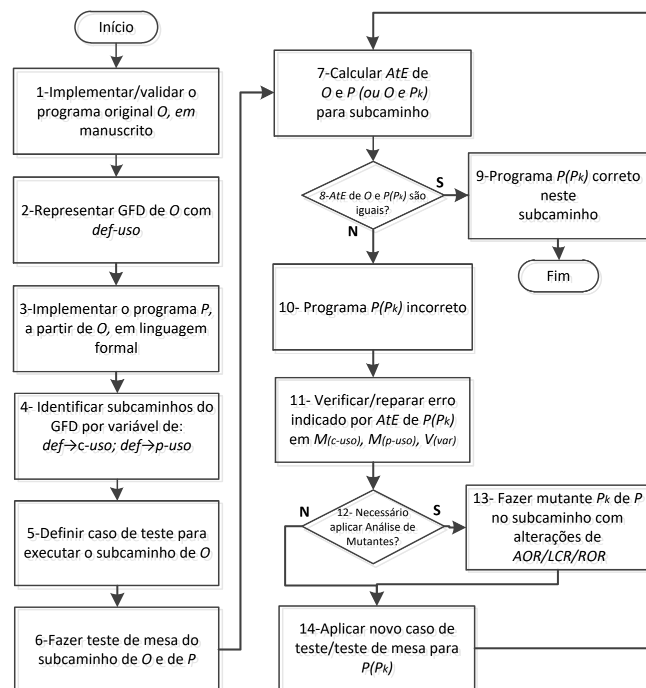


Figura 1. Fluxograma das tarefas metodológicas.

3. RESULTADOS

Com base no exposto, foi utilizada uma função simples de busca binária em um vetor ordenado, mostrada na Figura 1(a), que recebe como parâmetros um vetor, tamanho desse vetor e uma chave para busca, representando o programa original *O*. A escolha desse algoritmo foi pela simplicidade na apresentação e facilidade no entendimento.

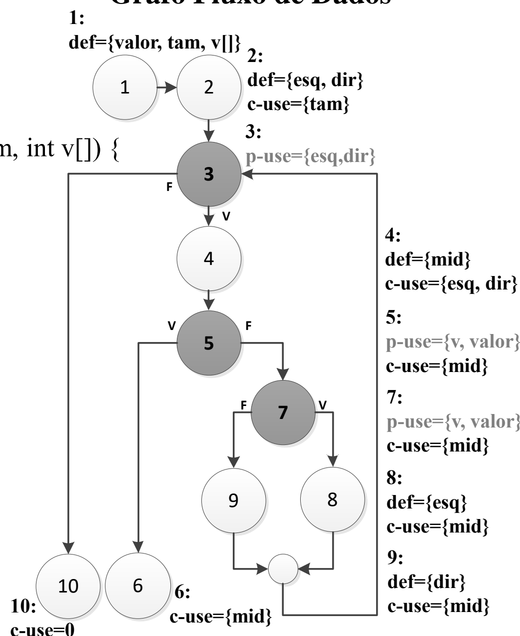
O GFD correspondente está ilustrado na Figura 1(b). A cada nó do grafo contém a notação *def-uso*, e são mencionadas informações sobre o fluxo de dados, entre os pontos de definição de variáveis e os pontos onde essas variáveis são usadas, seja *def→c-uso* ou *def→p-uso*.

Programa original (*O*)

```
/*1*/ int buscaBin (int valor, int tam, int v[]) {
/*2*/ int esq = 0, mid, dir = tam-1;
/*3*/ while (esq <= dir) {
/*4*/   mid = (esq + dir)/2;
/*5*/   if (v[mid] == valor)
/*6*/     return mid;
/*7*/   if (v[mid] < valor)
/*8*/     esq = mid + 1;
        else
/*9*/     dir = mid - 1;
      }
/*10*/ return -1;
}
```

(a)

Grafo Fluxo de Dados



(b)

Figura 1. (a) Unidade do programa *O*; (b) GFD do programa *O*.

A Tabela 2 mostra os nós com respectivas variáveis *(nodo, var)* do grafo na Figura 1(b), as representações de uso computacional *(def→c-uso)* e uso em predicados *(def→p-uso),* as notações *def-uso* para as variáveis no conjunto todos-usos e respectivo subcaminho para CT.

Para exemplificar a aplicação da *AtE* foram considerados os caminhos independentes *1,2,3,4,5,6; 1,2,3,4,5,7,8,3,10; 1,2,3,4,5,7,9,3,10*; e, *1,2,3,10;* que são aqueles de fluxo completo com variáveis de entradas definidas ou inicializadas no código ou de *input* externo, indicados aqui apenas para justificar os subcaminhos usados no GFD.

Dentre os subcaminhos determinados na Tabela 2, foi adotado para exemplificar a métrica *AtE*, o subcaminho *1,2,3,4,5,7,8*, com início no nodo-variável *(1,v)*, em *def→p-uso* de *(1,(7,8),V,v)*. Este subcaminho está contido no caminho independente *1,2,3,4,5,7,8,3,10* e inclui mais outros 10 subcaminhos de N. (1, 3, 4, 6, 8, 9, 10, 15, 17, 18 e 19) ou 53% do total, por representar a amplitude das variáveis no mesmo caso de teste, reduzindo a necessidade de testes para apenas 10 outros subcaminhos de N. (2, 5, 7, 11, 12, 13, 14, 16, 20 e 21).

Tabela 2. Subcaminhos resultantes para análise de mutantes.

N.	(no, var)	Definição – c-uso		Definição – p-uso		Subcaminho para CT
		def -> c-uso	(n, m, var)	def -> p-uso	(n, (m, k), bool, var)	
1	(1,valor)	∅	∅	{1,7}	{1,(7,8),V,valor}	1,2,3,4,5,7,8
2					{1,(7,9), F,valor}	1,2,3,4,5,7,9
3	(1,tam)	{2}	(1,2,tam)	∅	∅	1,2
4	(1,v)	∅	∅	{1,7}	{1,(7,8),V,v}	1,2,3,4,5,7,8
5					{1,(7,9),F,v}	1,2,3,4,5,7,9
6	(2,esq)	∅	∅	{2,3}	(2,(3,4),V,esq)	2,3,4
7					(2,(3,10), F,esq)	2,3,10
8	(2,esq)	{2,4}	(2,4,esq)	{2,3}	∅	2,3,4
9	(8,esq)	{8,4}	(8,4,esq)	∅	∅	8,3,4
10	(8,esq)	∅	∅	{8,3}	(8,(3,4),V,esq)	8,3,4
11					(8,(3,10), F,esq)	8,3,10
12	(9,dir)	∅	∅	{9,3}	(9,(3,4),V,dir)	9,3,4
13					(9,(3,10), F,dir)	9,3,10
14	(9,dir)	{9,4}	(9,4,dir)	∅	∅	9,3,4
15	(2,dir)	∅	∅	{2,3}	(2,(3,4),V,dir)	2,3,4
16					(2,(3,10), F,dir)	2,3,10
17	(4,mid)	{5}	(4,5,mid)	∅	∅	4,5
18	(4,mid)	{7}	(4,7,mid)	∅	∅	4,5,7
19	(4,mid)	{8}	(4,8,mid)	∅	∅	4,5,7,8
20	(4,mid)	{9}	(4,9,mid)	∅	∅	4,5,7,9
21	(4,mid)	{6}	(4,6,mid)	∅	∅	4,5,6

Para determinação da *AtE,* iniciou-se na definição de *v[]* no nodo 1 com fluxo até o uso em predicado no nodo 7 e arco (7, 8). Onde existem definições (*def*) para as variáveis *valor, tam, v[], esq, dir, mid*, usos computacionais (*c-uso*) para as variáveis *tam, esq, dir, mid* e, uso em predicado (*p-uso*) para *esq, dir, v, valor*. Todas essas variáveis estão contempladas pelo subcaminho *1,2,3,4,5,7,8*, para casos de testes com *AtE*, em uso computacional e uso em predicado.

O teste de mesa na Tabela 3, foi realizado para o programa original *O*, em tempo de projeto de software, e teve como entrada o seguinte caso de teste (CT-1): *valor=40; tam=5; v[]={10,20,30,40,50}*. O resultado da saída após execução, está demonstrado na linha do laço de repetição do ***while*** (nodo 6 do GFD), onde também estão os valores das variáveis e o valor retornado.

Tabela 3. Teste de mesa para o caso de teste aplicado.

Programa *O* CT-1: *valor=40; tam=5; v[]={10,20,30,40,50}*.*	**valor**	**tam**	**v[]**	**esq**	**mid**	**dir**	**v[mid]**	**while (esq <= dir)**	**if (v[mid] == valor)**	**if (v[mid] < valor)**	**return**
/*1*/ int bBinaria (int valor, int tam, int v[]) {	40	5	*	-	-	-	-	-	-	-	-
/*2*/ int esq = 0, mid, dir = tam-1;	40	5	*	0	-	4	-	-	-	-	-
/*3*/ while (esq <= dir) {	40	5	*	0	-	4	-	V	-	-	-
/*4*/ mid = (esq + dir)/2;	40	5	*	0	2	4	-	V	-	-	-
/*5*/ if (v[mid] == valor)	40	5	*	0	2	4	30	V	F	-	-
/*6*/ return mid;	-	-	-	-	-	-	-	-	-	-	-
/*7*/ if (v[mid] < valor)	40	5	*	0	2	4	30	V	V	-	-
/*8*/ esq = mid + 1;	40	5	*	3	2	4	30	V	V	-	-
else	-	-	-	-	-	-	-	-	-	-	-
/*9*/ dir = mid - 1;	-	-	-	-	-	-	-	-	-	-	-
}											
/*10*/ return -1;											
repetição while											
/*3*/ while (esq <= dir) {	40	5	*	3	2	4	30	V	-	-	-
/*4*/ mid = (esq + dir)/2;	40	5	*	3	3	4	40	V	-	-	-
/*5*/ if (v[mid] == valor)	40	5	*	3	3	4	40	V	V	-	-
/*6*/ return mid;	40	5	*	3	3	4	40	V	V	-	3
/*7*/ if (v[mid] < valor)											
/*8*/ esq = mid + 1;											
else											
/*9*/ dir = mid - 1;											
}											

A métrica *AtE* está demonstrada nos cálculos ilustrados da Tabela 4. Cada linha representa um nodo do GFD e contém o código executado no subcaminho, o número da equação, os valores em hexadecimal dos parâmetros na respectiva equação e o resultado para $M_{(c\text{-}uso)}$ ou $M_{(p\text{-}uso)}$. Para o cálculo de $V_{(var)}$ aplicada pela equação 4, os valores foram considerados da última linha executada do programa, conforme o teste de mesa da Tabela 3.

A Tabela 4(a) ilustra a execução do subcaminho *1,2,3,4,5,7,8* do programa original *O*, para o CT-1. A Tabela 4(b) mostra o Programa *P* – com o mesmo erro no nodo 3 citado, também com CT-1 e o mesmo subcaminho considerado.

Embora os valores das variáveis e da saída ao final da execução, sejam os mesmos nesse subcaminho testado nos programas *O* e *P*, com o mesmo CT-1 (ver Tabela 3), as métricas *AtE's* apresentaram resultados diferentes (**59A** e **55D**, respectivamente). Como também, sem muito esforço, ficou evidente a indicação do erro específico em $M_{(p\text{-}uso)}$, no nodo 3 de *P*, com a diferença entre os resultados **7E** e **41**, respectivamente (ver Tabela 4).

Por outro lado, para verificar a aplicação somente com testes FD e AM, sem uso da métrica *AtE*, no subcaminho mencionado *(1,(7,8),V,v)*, do programa *P*, foram executados casos de testes CT-1 e CT-2, numa simulação hipotética com um erro semântico no nodo 3, em ***while (esq < dir)***.

O CT-1 foi aplicado conforme os mesmos valores de entrada (*valor=40; tam=5; v[]={10,20,30,40,50})*. O resultado de saída de *P* (com erro) foi o mesmo do programa *O,* em ***return mid,*** com valor **3** (ver Tabela 3). Ou seja, nesse ensaio, caso fosse aplicado apenas o teste FD para os programas *O* e *P*, com o mesmo caso de teste, os resultados das saídas seriam os mesmos, possibilitando confundir o testador. O mesmo aconteceria com aplicação de AM quando o resultado seria mutante vivo, sem detecção do erro e necessidade de usar muito mais mutantes do programa.

Para o teste com CT-2 com: *valor=**50**; tam=5; v[]={10,20,30,40,50},* também no Programa *P* e com o mesmo erro anterior no nodo 3, o valor de saída de retorno seria **-1**,

diferente do valor correto esperado **4**. Evidenciando que esse erro lógico no programa *P,* somente poderia ser detectado em modo de operação do programa. Com aplicação de AM, seria necessário gerar vários mutantes com as alterações sintáticas no nodo 3, com *while(...)*: *(esq >= dir), (esq == dir), (esq != dir), (esq < dir), (esq > dir),* todos resultariam mortos e com possibilidade de confundir o testador pois seria preciso mais esforço para achar a falha.

Observa-se que *AtE* possibilita detecção de erros, tanto no cálculo geral da métrica, como nas parcelas intermediárias de *c-uso*, *p-uso* ou na saída de resultado, na comparação entre o programa original idealizado em tempo de projeto e a codificação na implementação do software.

Por outro lado, essa métrica permite a indicação mais precisa sobre a localização dos erros semânticos e lógicos no contexto do código, comparando-se com as técnicas fluxo de dados e análise de mutantes. Além da diminuição da necessidade de verificação de usos de variáveis em *def-uso*, para cada variável, e da redução de geração de mutantes para detecção de erros. Por conseguinte, menores esforço computacional e custos de execução.

Tabela 4. (a) Cálculo de *AtE* para o programa *O*; (b) Cálculo de *AtE* para o programa *P*.

(a)

Subcaminho do programa (*O*) do projeto de software. Caso de Teste: valor=40; tam=5; v[]={10,20,30,40,50}	**Eq**	**Valores nas parcelas (em hexadecimal)**	**M (c-uso)**	**M (p-uso)**
/*1*/ int bBinaria (int valor, int tam, int v[]) {	2	(28+5+96)+(0)+(0)	C3	-
/*2*/ int esq = 0, mid, dir = tam-1;	2	(0+0+4+5)+(3D+3D+2D)+(1)	B1	-
/*3*/ while (esq <= dir) {	3	(0)+(0+3C+3D+4+1)+(0)	-	7E
/*4*/ mid = (esq + dir)/2;	2	(2+0+4)+(3D+2B+2F)+(2)	9F	-
/*5*/ if (v[mid] == valor)	3	(0)+(30+3D+3D+40+0)+(0)	-	EA
/*6*/ return mid;	2	-	-	-
/*7*/ if (v[mid] < valor)	3	(0)+(30+3C+40+1)+(0)	-	AD
		Soma	213	215
		Sub-Total(Mc-uso+Mp-uso)	428	

V(var)
valor=40;tam=5;v[]={10,20,30,40,50};
esq=3;mid=3; dir=4;v[mid]=40;while(esq<=dir)=V;
if(v[mid]==valor)=V;if(v[mid]<valor)=;
Saída=3
	4	(28+5+96+3+3+4+28+1+1+0)+(3)	172	

		Cálculo *AtE* do programa *O*	59A	

(b)

Subcaminho do programa (*P*) com erro no nodo 3. Caso de Teste: valor=40; tam=5; v[]={10,20,30,40,50}	**Eq**	**Valores nas parcelas (em hexadecimal)**	**M (c-uso)**	**M (p-uso)**
/*1*/ int bBinaria (int valor, int tam, int v[]) {	2	(28+5+96)+(0)+(0)	C3	-
/*2*/ int esq = 0, mid, dir = tam-1;	2	(0+0+4+5)+(3D+3D+2D)+(1)	B1	-
/*3*/ while (esq < dir) {	3	(0)+(0+3C+4+1)+(0)	-	41
/*4*/ mid = (esq + dir)/2;	2	(2+0+4)+(3D+2B+2F)+(2)	9F	-
/*5*/ if (v[mid] == valor)	3	(0)+(30+3D+3D+40+0)+(0)	-	EA
/*6*/ return mid;	2	-	-	-
/*7*/ if (v[mid] < valor)	3	(0)+(30+3C+40+1)+(0)	-	AD
		Soma	213	1D8
		Sub-Total(Mc-uso+Mp-uso)	3EB	

V(var)
valor=40;tam=5;v[]={10,20,30,40,50};
esq=3;mid=3;dir=4;v[mid]=40;while(esq<=dir)=V;
if(v[mid]==valor)=V;if(v[mid]<valor)=;
Saída=3
	4	(28+5+96+3+3+4+28+1+1+0)+(3)	172	

		Cálculo *AtE* do programa *P*	55D	

4. CONCLUSÃO

Este artigo apresentou uma nova abordagem sobre aplicação de teste estrutural intitulada de Assinatura do Teste Estrutural (*AtE*), que possibilita detectar possíveis erros semânticos e lógicos em unidades de código e validar a implementação de uma unidade de programa de acordo com o projeto de software.

Por essa abordagem, observou-se a detecção de erro lógico em unidade de código entre a definição e o uso de variáveis, tornando mais rápido a descoberta de falha se comparado com outras técnicas tradicionais como fluxo de dados e análise de mutantes. Mostrou redução em 53% da necessidade de testes para todo o fluxo de dados e, por variável, na simulação realizada.

Trata-se de ensaio inicial que ainda depende de experimentação pela indústria do software, porém a métrica mostra-se favorável à possibilidade de encontrar falhas em unidades menores para teste estrutural, a redução de esforço com mutantes e o custo computacional, o que é um incentivo para continuação da pesquisa nesta linha.

Para o futuro pretende-se avançar com ensaios utilizando-se algoritmos com diversas linguagens de programação para então, gerar condições de pesquisas operacionais através da validação com especialistas e por técnicas estatísticas. Pretende-se também, o desenvolvimento de uma ferramenta que automatize essa prática, no propósito da disseminação da abordagem e da extensão para a área de indústria do software.

5. REFERÊNCIAS

[1] 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.

[2] Alshahwan, N., and Harman, M. (2012, July). State aware test case regeneration for improving web application test suite coverage and fault detection. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis* (pp. 45-55).

[3] ASCII, American Standard Code for Information Interchange, American Standards Association, New York. Retrieved from https://www.ascii-code.com/

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

[5] Beizer, B. (2003). *Software testing techniques*. Dreamtech Press.

[6] Buy, U., Orso, A., and Pezze, M. (2000). Automated testing of classes. *ACM SIGSOFT Software Engineering Notes*, *25*(5), 39-48.

[7] Chaim, M. L., and De Araujo, R. P. A. (2013). An efficient bitwise algorithm for intra-procedural data-flow testing coverage. *Information Processing Letters*, *113*(8), 293-300.

[8] Debbarma, M. K., Debbarma, S., Debbarma, N., Chakma, K., and Jamatia, A. (2013). A review and analysis of software complexity metrics in structural testing. *International Journal of Computer and Communication Engineering*, *2*(2), 129-133.

[9] Delamaro, M., Jino, M., and Maldonado, J. (2016). *Introdução ao teste de software*. 2a. Ed, Elsevier Brasil. Campus.

[10] Denaro, G., Margara, A., Pezze, M., and Vivanti, M. (2015, May). Dynamic data flow testing of object oriented systems. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* (Vol. 1, pp. 947-958). IEEE.

[11] Endres, A. (1975). An analysis of errors and their causes in system programs. *IEEE Transactions on Software Engineering*, (2), 140-149.

[12] Frankl, P.G., 1988. The use of data flow information for the selection and evaluation of software test data.

[13] Frankl, P.G. and Weyuker, E.J., 1988. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, *14*(10), pp.1483-1498.

[14] Girgis, M. R. (2005). Automatic Test Data Generation for Data Flow Testing Using a Genetic Algorithm. *J. UCS*, *11*(6), 898-915.

[15] Grechanik, M., and Devanla, G. (2016, August). Mutation integration testing. In 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS) (pp. 353-364). IEEE.

[16] Hassan, M. M., and Andrews, J. H. (2013, May). Comparing multi-point stride coverage and dataflow coverage. In *2013 35th International Conference on Software Engineering (ICSE)* (pp. 172-181). IEEE.

[17] Hecht, M.S., 1977. *Flow analysis of computer programs*. Elsevier Science Inc.

[18] Herman, P. M. (1976). A data flow analysis approach to program testing. *Australian Computer Journal*, *8*(3), 92-96.

[19] Howden, W.E., 1976. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, (3), pp.208-215.

[20] Jia, Y., & Harman, M. (2010). An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, *37*(5), 649-678.

[21] Maldonado, J.C., 1991. Critérios potenciais usos: Uma contribuição ao teste estrutural de software. *Ph.D. Dissertation.* DCA/FEE/UNICAMP.

[22] McCabe, T.J., 1976. A complexity measure. *IEEE Transactions on software Engineering*, (4), pp.308-320.

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

[24] Pressman, R. and Maxim, B., 2016. *Engenharia de Software* (8ª. ed). McGraw Hill. Brasil.

[25] Rapps, S., and Weyuker, E. J. (1985). Selecting software test data using data flow information. *IEEE transactions on software engineering*, (4), 367-375.

[26] Rocha, Ana Regina Cavalcanti da, José Carlos Maldonado, and Kival Chaves Weber. (2001). Qualidade de software: teoria e prática.

[27] Santelices, R., Jones, J. A., Yu, Y., and Harrold, M. J. (2009, May). Lightweight fault-localization using multiple coverage types. In *2009 IEEE 31st International Conference on Software Engineering* (pp. 56-66). IEEE.

[28] Su, T., Wu, K., Miao, W., Pu, G., He, J., Chen, Y., and Su, Z. (2017). A survey on data-flow testing. *ACM Computing Surveys (CSUR)*, *50*(1), 1-35.

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

[30] 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 (pp. 258-265). Springer, Boston, MA.