MSUT

METRIC FOR SOFTWARE UNIT TESTING

Metric for Software Unit Testing (MSUT) is a new approach to testing and detecting logical and se-mantic errors in computer programs.

Sergio Fred Andrade

Sobre o autor

Sergio Fred Andrade

PhD in Science, Master in Systems and Computing, specialist in Advanced Computing and Distance Education, Bachelor in Information Systems and Mathematics. He has experience in computing, analysis and information systems, database, data science, and software engineering.

Table Of Contents

Metric for Software Unit Testing (MSUT)

Sérgio Fred Ribeiro Andrade [Universidade Estadual de Santa Cruz | sergiof@uesc.br]

Abstract

Metric for Software Unit Testing (MSUT) is a new approach to testing and detecting logical and semantic errors in computer programs, with the purpose of meeting software requirements. MSUT compares the specification in the software design and the prototype implementation, with the application of summations on symbols, coded sentences and data such as variables, data types, operands, operators, assignments, parameter passes and others to check for errors. The results can contribute to the code structural testing stage in the software development process. The tests shown indicate that the metric has great potential for automated software testing and suggests continuity for further research as well as the development of specific tools for this area.

Keywords: Software Unit Testing, Structural Testing, White Box Testing, Software Metrics

Introduction

Software testing activities, especially structural and unit testing, seek to ensure software quality through source code analysis (Delamaro, Jino and Maldonado 2016). According to Pressman and Maxim (2016), software quality is inherent to the requirement of functional requirements, explicitly described specifications and established performance.

A pertinent reason for applying software unit testing is indicated in Howden (2011), which observes that in the code implementation process, semantic and logical errors most often occur unintentionally, despite the use of consistent methods, support tools for syntax and semantic debugging and trained professionals.

A semantic error is understood as a problem of meaning, which occurs even when a sentence is syntactically correct, but with some change in operands and/or operators causing an unsatisfactory result, which cannot be identified by a parser analysis. A logical error occurs when the code is correct in terms of syntax, but inadequate in its algorithmic construction, which can result in an implementation that does not meet the functional requirements and, after execution, does not generate the expected result.

Semantic and logical errors can be dealt with through software unit testing, which is used in the structural analysis of small blocks of code capable of receiving inputs and generating outputs after processing.

As mentioned in Pressman (2016), a recommended process for testability is software metrics when they seek to understand the independent paths exercised with data, the logical decisions of conditional structures, the cycles in their operational limits and the internal data structures.

Therefore, this work adopts the "error-based" principle of Endres (1975), which uses information about the types of failures, including those of semantics and logic. And the metrics for testing Cyclomatic Complexity and linearly independent paths in McCabe (1976), Control Flow Graph (CFG) and Data Flow Graph (DFG) in Rapps and Weyuker (1982) are considered.

The number of independent paths is calculated by cyclomatic complexity, based on the logical predicates of the algorithm and the CFG, and used in the DFG according to the processes in computational and predicate uses, that is, from the definition of variables to the effective uses.

These techniques are described in the literature in isolation without complementation, however, in this paper the purpose is to improve the search for errors in programs with the integration between them and the inclusion of a metric for detecting errors, in an adaptation to the work of Andrade (2020).

To detect errors, the paths are searched for syntactic changes in the meaning of the code or erroneous logical constructions, with the application of an oracle and "trace table", for comparison between the output and the expected result, between the project program and the implemented program.

Therefore, the objective of the present paper is to introduce a metric for software unit testing, which seeks to: a) carry out tests on smaller units considering the definition and use of variables; and, b) identification of flaws in the code structure, location of semantic, logical and algorithmic construction errors to validate the implementation of a program unit in accordance with the software project.

Methodology

The methodological task begins with the calculation of cyclomatic complexity to determine the number of independent paths by the equivalence equation $p+1 = e–n+2$, where p is the number of logical predicates of the algorithm, e is the number of edges, and n the number of CFG nodes (McCabe 1976).

An independent path is a sequence of nodes ($n1, n2, ..., nk$), with $k \geq 2$, with a complete path containing $n1$ at the input and nk as the output. The graph nodes cannot be visited more than once with the exception of repetitions and are quantified by calculating the cyclomatic complexity.

The CFG becomes DFG to indicate the definitions of variables (def), the computational use (c-use) – in operations, and the predicate use (p-use) – in logical propositions, considering All-Definitions and All-Uses of data (def-use), according to Rapps and Weyuker (1982). The def-use criterion for the variable x corresponds to a pair of nodes in the graph (n, m), so that x is in def(n). The definition of x in n reaches m and x is used in (m). Thus, the value assigned to x in n is used in m, computational (def→c-use) or predicate (def→p-use).

Definition (def) occurs when a variable receives a value assigned in the first occurrence and in subsequent occurrences when the value changes, for example for var: *int var = 0*; ... *var = m + 1;* . Computational use (c-use) occurs when the variable is used in a process, such as assignment to variable, arithmetic operation, output for printing and sending parameters, for example for var: *m = var + 1;*. Predicate use (p-use), in propositions where the boolean function $p: x \rightarrow \{true, false\}$, that is, when the variable is used in a logical condition, for example for var: *if (var != 0) {...}*.

On the path to be tested, the Trace Table (TT) with Test Case (TC) is applied to find semantic errors with the exchange of operands and/or operators through the general result of equation 1, which is determined by def→c-use with equation 2, def→p-use with equation 3 and input/output variables with equation 4.

When semantic errors are not due to syntactic exchange of symbols, variables or data, but due to erroneous algorithmic construction in disregard of functional requirements and cause unexpected output after execution, another CT is applied to TT and equation 5. If the error is not found, the calculation is returned with a new TC, TT and calculations with equations 1 – 5, repeating the previous steps.

2.1 Justification for applying the metric

MSUT is calculated by the sums of grammatical symbols in the code and by the evolution of data into variables using the hexa-decimal system of numbers and characters of ASCII (2025), in the def→c-use and def→p-use structures and by comparing the data of inputs, outputs and those expected after execution, as shown in equations 1 - 5.

The technique adapts to the Signature of the Structural Test (SST) method proposed by Andrade (2020), as there was a need to use an oracle, that is, an ideal project program considered correct to serve as a comparison with the implementation of the prototype, in the same test case or in different test cases.

Modifications were made to equation 4 to include the input data in the processing and enable comparisons between the received and output values, between the project codes and the implemented one. And, the inclusion of equation 5 to check errors in the algorithmic construction when there are no syntactic changes in the code.

2.2 Equations 1-5 for MSUT

The MSUT formulations are:

Equation 1: is the result of the MSUT metric with equations 2, 3, 4.

Eq. (1)

$$\boldsymbol{MSUT} = M_{(c-use)} + M_{(p-use)} + V_{(var)}$$

Equation 2: M(c-use) - is a metric that identifies semantic or logical errors related to computational use (def→c-use).

Eq. (2)

$$M_{(c-use)} = \sum_{i=1}^{N} \Big(\sum_{j=1}^{M} w_j + \sum_{l=1}^{O} d_l + \sum_{k=1}^{P} c_k \Big)_i$$

N = Number of computational uses (def→c-use) in the path for all processing in the executable lines.

i = *i-th* computational use command in the path.

M = Number of variables in an executable line.

wj = *j-th* variable used in processing in an executable line.

O = Number of operators, commands, function, data type in an executable line.

dl = *l-th* arithmetic/relational/logical operator, assignment, function, data type, print command in an executable line.

P = Number of constants in an executable line.

ck = *k-th* constant in an executable line.

i, j, l, k = Indexes ranging from *1* to *n*.

Equation 3: M(p-use) - Metric that evaluates semantic or logical errors in structures with logical predicates (def→p-use).

Eq. (3)

$$M_{(p-use)} = \sum_{x=1}^{K}(\sum_{m=1}^{R} o_m + \sum_{f=1}^{S}(a+q+b+r)_f \ + z)_x$$

K = Number of logical predicates (p-use) in selection/repetition structures in the path.

x = *x-th* selection/repetition (p-use) structure in the path.

R = Number of conjunction, disjunction or negation operators in propositions in a logical sentence structure.

om = *m-th* operator of conjunction, disjunction or negation in a logical proposition.

S = Number of simple logical propositions in a logical sentence structure.

f = *f-th* simple logical proposition in an executable line.

a, b = Operands of the simple logical proposition in an executable line.

q = Relational operator in a structure with a logical sentence.

r = Result of the simple logical proposition.

z = Result of the compound logical proposition.

x, m, f = Indexes ranging from 1 to n.

Equation 4: V(Var): Metric that evaluates the effective output of the program based on the inputs provided.

Eq. (4)

$$V_{(var)} = \sum_{t=1}^{T} e_t + \sum_{g=1}^{U} v_g + \sum_{h=1}^{V} s_h$$

T = Amount of data at the program entry, at the beginning of the path.

et = t-th value of the data in the input or received parameters.

U = Number of variables after the program ends, at the end of the path.

vg = g-th value of the variables after the program ends.

V = Number of output or return variables after executing the program.

sh = h-th output value or return of variables after executing the program.

t, g, h = Indexes ranging from 1 to n.

Equation 5: V(errorLogic): Applies when the semantic/logical error was not detected with equations 1 - 4 and does not occur due to the use of operators, operands, commands and identifiers exchanged or different between programs *O* and *P*. Or, in the suspicion of a discrepancy between the expected value and the effective output value of the program. The sum of identifier symbols, operands and operators, per line, is justified to determine whether programs *O* and *P* are exactly the same and do not present any syntactical differences.

Eq. (5)

$$V_{(errorLogic)} = \sum_{t=1}^{X} operator_t + \sum_{f=1}^{Y} entry_f + \sum_{g=1}^{Z} result_g + \sum_{h=1}^{Q} exit_h$$

X = Number of characters of operands, operators, data types, functions in an executable line.

operatort = t-th character of operands, operators, identifiers, data types or functions in an executable line.

Y = Number of input variables for program execution, at the beginning of the path.

entryf = f-th entry value for executing the program.

Z = Number of data in expected values after executing the program.

resultg = g-th expected value after executing the program.

Q = Number of output variables after executing the program.

exith = h-th exit value after executing the program.

t, f, g, h = Indexes ranging from 1 to *n*.

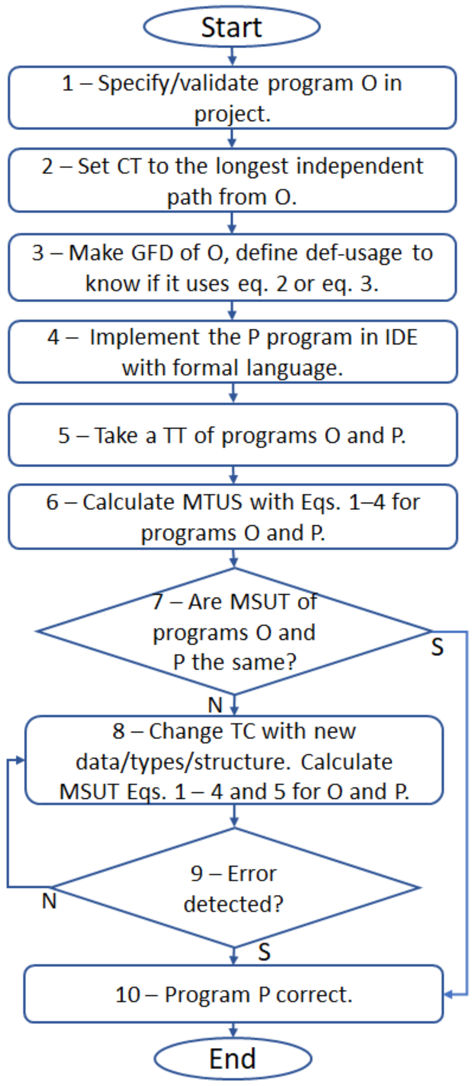
For all equations: 1) If there are no values in the corresponding parameters in the program code, the value must be equal to zero; 2) To avoid confusion between the negative sign of a number and the subtraction operator, all numbers are computed as non-negative real numbers, that is, $R+ = \{x \in R \mid x \geq 0\}$; 3) In homogeneous and heterogeneous data structures, each memory position or variable is considered as a sum portion; 4) For arithmetic, relational, logical, connective and assignment operators, including if concatenated, the characters in hexadecimal from the table in ASCII (2025) are added; 5) For data types, functions/methods for receiving and returning parameters in characters, identifiers, values of derived variables, strings or object instantiation, the characters of the name are added in hexadecimal; 6) For decimal values with floating point, the hexadecimal numbers are added as characters using the ASCII (2025).

2.3 Procedure flowchart for MSUT

The procedures for calculating the metric are in **Figure 1** and described below:

Figure 1. Flowchart with procedures for MSUT.

MSUT


Start
1 – Specify/validate program O in project.
2 – Set CT to the longest independent path from O.
3 – Make GFD of O, define def-usage to know if it uses eq. 2 or eq. 3.
4 – Implement the P program in IDE with formal language.
5 – Take a TT of programs O and P.
6 – Calculate MTUS with Eqs. 1–4 for programs O and P.
7 – Are MSUT of programs O and P the same?
S
N
8 – Change TC with new data/types/structure. Calculate MSUT Eqs. 1 – 4 and 5 for O and P.
9 – Error detected?
N
S
10 – Program P correct.
End

1. Specify program algorithm *O* and validate at design time;

2. Define TC for the longest independent path or the path with suspected errors;

3. Make the DFG for *O*, and indicate in each node the def-use, (c-use - computational – equation 2) and/or (p-use – logical predicate – equation 3);

4. Implement the *P* program in an IDE with a compiled or interpreted formal language;

5. Make TT in *O* and *P* with TC defined to check the expected output according to the code input;

6. Calculate MSUT (equations 1 - 4) for programs *O* and *P*, for simple comparison;

7. Check MSUT results. If they are the same, *P* is correct and go to procedure 10, if different, go to 8;

8. Change the TC or range of data or types and ranges used. Apply TT and equations 1 – 4 and go to 9.

9. If the result of the MSUT's is different from what was expected, go back to 8 and apply a new TC with the programs *O* and *P* in equation 5, check if *O* and *P* have syntactically the same codes. Being equal, compare the expected result and the output after executing *P* to detect the error. If different, assign a new TC and check the logical construction of program *O*. Repeat the action until you obtain the same MSUT results for *O* and *P*, equations 1 – 5.

Results

Based on the methodology presented, two programs were used for testing with MSUT. These algorithms were chosen for their simplicity and ease of application of the metric.

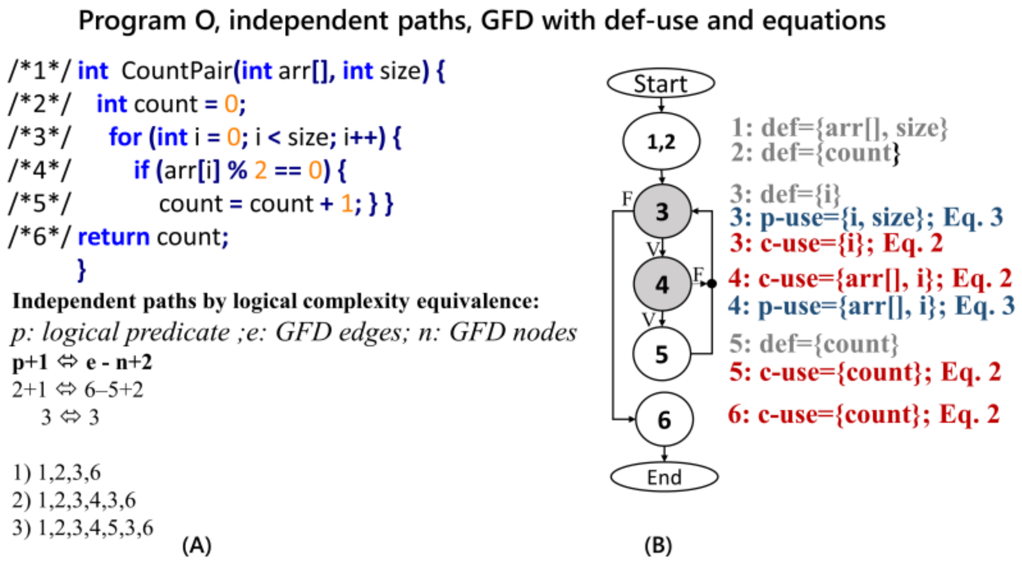
The results of testing these programs are presented below, each with versions of program *O* (design) and program *P* (implementation), with test cases.

The first is a program that counts even numbers in a collection. Version *P* for testing contains a syntactical error on line 4 with the use of the relational operator **(2 <= 0)**, which should be **(2 == 0)**. The second algorithm performs the exponential value calculation, where the respective *O* and *P* programs present a logical construction error that allows correct execution for positive numbers and incorrect execution for negative numbers.

3.1 Application of Equations 1 - 4

The first algorithm receives as parameters a vector and its size, shown in **Figure 2A** for program *O*. Each node of the graph in **Figure 2B** contains the notation for *def→c-use* for equation 2 and/or *def→p-use* for equation 3.

The cyclomatic complexity calculation resulted in 3 independent paths and the longest one (1,2,3,4,5,3,6) was chosen for testing.



Program O, independent paths, GFD with def-use and equations

```
/*1*/ int  CountPair(int arr[], int size) {
/*2*/   int count = 0;
/*3*/     for (int i = 0; i < size; i++) {
/*4*/        if (arr[i] % 2 == 0) {
/*5*/            count = count + 1; } }
/*6*/ return count;
        }
```

Independent paths by logical complexity equivalence:

p: logical predicate ;e: GFD edges; n: GFD nodes

p+1 ⇔ e - n+2

2+1 ⇔ 6–5+2

3 ⇔ 3

1) 1,2,3,6
2) 1,2,3,4,3,6
3) 1,2,3,4,5,3,6

(A)

Figure 2. (A) Algorithm and (B) DFG of Program *O*.

The metric is demonstrated in **Table 1A** and **Table 1B** when executing programs *O* and *P*, where the calculations per line are shown.

Each line represents a DFG node executed by TC-1 *(arr=[3,4,5] and size=3)* and displayed by TT with variable data, the applied equation, the MSUT calculation in hexadecimal and the results for *c-use* (eq. 2) and *p-use* (eq. 3). For the calculation of V(var) applied by equation 4, the values considered were in the return of the function, at the end of the execution scope.

Table 1. Application of equations 1 – 4. A) Program *O* and B) Program *P*.

A)						**Path: 1,2,3,4,5,3,6**					
Program O	Trace table - variables							Calculation MTUS			
TC-1: arr[3,4,5]; size = 3	**arr[i]**	**size**	**cout**	**i**	**(i < size)**	**(arr[i] % 2 == 0)**	**Eq.**	**Sum in hexadecimal (Eq. 2)**	**c-use**	**Sum in hexadecimal (Eq. 3)**	**p-use**
/*1*/ int CountPair(int arr[], int size) {	[3,4,5]	3					2	(3+4+5)+0+(32B+32B+32B)+0+3	990	*	*
/*2*/ int count = 0;	[3,4,5]	3	0				2	32B+3D+0	368	*	*
/*3*/ for (int i = 0; i < size; i++) {	[3,4,5]	3	0	0; 1; 2	V; V; V		2;3	(32B+0+3D+0)+(0+2B+2B)+(1+2B+2B)+(2+2B+2B)	46D	(0)+(0+3C+3+1)+(1+3C+3+1)+(2+3C+3+1) + 0	C3
/*4*/ if (arr[i] % 2 == 0) {	[3,4,5]	3	0			F; V; F	2;3	(3+25+2)+(4+25+2)+(5+25+2)	81	(0)+(1+3D+3D+0+0) + (0+3D+3D+0+1) +(1+3D+3D+0+0)	171
/*5*/ count = count + 1; } }	[3,4,5]	3	1				2	(1+3D+0+2B+1)	6A	*	*
/*6*/ return count;	[3,4,5]	3	**1**	3	F	F	2	(0+1+0)	1	*	*
}								Total	**1251**	Total	**17D**
										1251 + 17D = 13CE	
arr[]={3,4,5}; tam=3; count=1; i=3	[3,4,5]	3	1	3			4			3 + 4 + 5 + 3 + 1 + 3 = 13	
							1	TOTAL		**13CE + 13 = 13E1**	

B)						**Path: 1,2,3,4,5,3,6**					
Program P	Trace table - variables							Calculation MTUS			
TC-1: arr[3,4,5]; size = 3	**arr[i]**	**size**	**count**	**i**	**(i < size)**	**(arr[i] % 2 <= 0)**	**Eq.**	**Sum in hexadecimal (Eq. 2)**	**c-use**	**Sum in hexadecimal (Eq. 3)**	**p-use**
/*1*/ int CountPair(int arr[], int size) {	[3,4,5]	3					2	(3+4+5)+0+(32B+32B+32B)+0+3	990	*	*
/*2*/ int count = 0;	[3,4,5]	3	0				2	32B+3D+0	368	*	*
/*3*/ for (int i = 0; i < size; i++) {	[3,4,5]	3	0	0; 1; 2	V; V; V		2;3	(32B+0+3D+0)+(0+2B+2B)+(1+2B+2B)+(2+2B+2B)	46D	(0)+(0+3C+3+1)+(1+3C+3+1)+(2+3C+3+1) + 0	C3
/*4*/ if (arr[i] % 2 <= 0) {	[3,4,5]	3	0			F; V; F	2;3	(3+25+2)+(4+25+2)+(5+25+2)	81	(0)+(1+3C+3D+0+0) + (0+3C+3D+0+1) +(1+3C+3D+0+0)	**16E**
/*5*/ count = count + 1; } }	[3,4,5]	3	1				2	(1+3D+0+2B+1)	6A	*	*
/*6*/ return count;	[3,4,5]	3	**1**	3	F	F	2	(0+1+0)	1	*	*
}								Total	**1251**	Total	**231**
										1251 + 231 = **1482**	
arr[]={3,4,5}; tam=3; count=1; i=3	[3,4,5]	3	1	3			4			3 + 4 + 5 + 3 + 1 + 3 = 13	
							1	TOTAL		1482 + 13 = **1495**	

The output after executing the programs is shown on line 6 with the returned value and the values of the variables at the end of the scope after execution.

The Program *P* was implemented with a semantic error on line 4 ***(arr[i] % 2 <= 0)***. Both do not produce errors in the output after execution, because for TC-1 *(arr=[3,4,5] and size=3)*, the value returned was 1, which corresponds to the number of pairs in the numeric collection of the input (**Table 1A** and **Table 1B**).

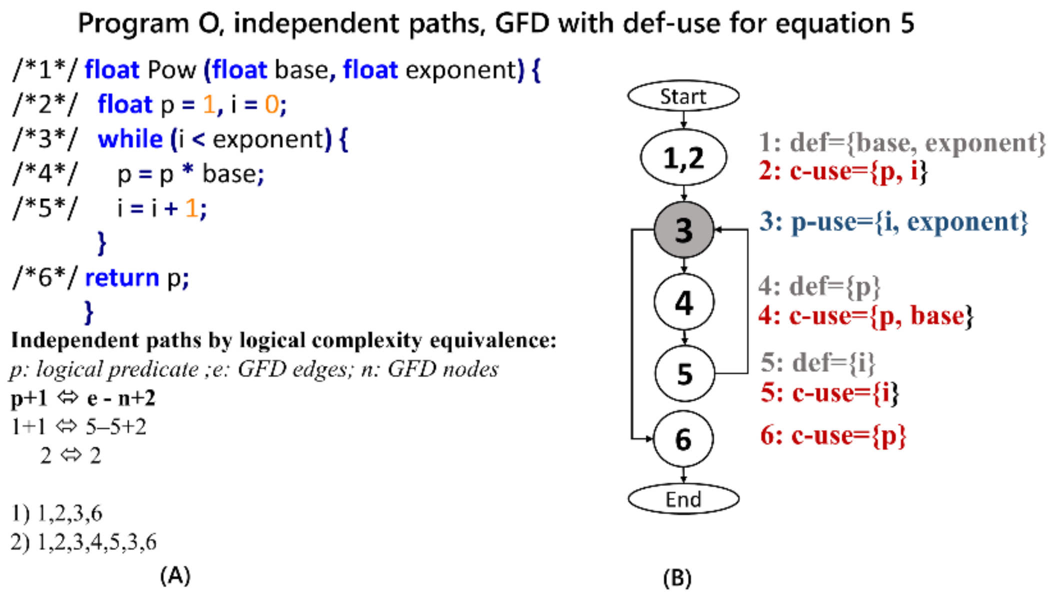
Although this return value is the same, it is observed that the results of the MSUT calculations for *O* and *P* are different in line 4, presenting a result of **171** for *O* and **16E** for *P*. Which totals different MSUT (**13E1** for *O* and **1495** for *P*). This difference indicates that **program P presents a semantic error in line 4**, in the logical predicate, which was clearly detected through equation 3 - M(p-use).

For simple validation with the MSUT metric in program P, other test cases were executed, TC-2 *(arr=[6,7,9] and size=3)* and TC-3 *(arr=[7,9,10] and size=3)*, with the same semantic error in line 4, in *(arr[i] % 2 <= 0)*.

The output results of *P* were the same as those of program *O*, with a *return count* of 1. In other words, in these tests, if MSUT were not applied to programs *O* and *P*, they would have identical output results, even with a semantic error in line 4, which could confuse the user that the code was correct. However, Program *P* contains an error in line 4 and may present incorrect output depending on the input data and this error was detected.

3.2 Application of Equation 5

The second program, in **Figure 3**, *O* and *P* do not present a syntax error or change of operators or operands, but they have algorithmic construction flaws due to disregard for functional requirements, which do not allow the correct calculation of power with exponents of negative integers and always returns 1.


Program O, independent paths, GFD with def-use for equation 5

```
/*1*/ float Pow (float base, float exponent) {
/*2*/   float p = 1, i = 0;
/*3*/   while (i < exponent) {
/*4*/     p = p * base;
/*5*/     i = i + 1;
        }
/*6*/ return p;
      }
```

Independent paths by logical complexity equivalence:

p: logical predicate ;e: GFD edges; n: GFD nodes

$p+1 \Leftrightarrow e - n+2$

$1+1 \Leftrightarrow 5–5+2$

$2 \Leftrightarrow 2$

1) 1,2,3,6
2) 1,2,3,4,5,3,6

Figure 3. Use of equation 5, (A) Program *O* and (B) GFD.

In this algorithm, the comparison with the expected results and the output results between the design specification (program *O*) and the implemented program *P* is fundamental for detecting failure in the code execution.

Table 2. Application of equation 5 to Programs *O* and *P*.

Path: 1,2,3,4,5,3,6; **TC-4: base = 5, exponent = 2 (Eq. 5)**					
Program *O* and *P*	operator (sum)	entry	result	exity	Hexa
float Pow (float base, float exponent) {	66+6C+6F+61+74+50+6F+74+65+6E+63+69+61+66+6C+6F+61+74+62+61+73+65+66+6C+6F+61+74+65+78+70+6F+65+6E+74+65 = **E78**	5 + 2			E7F
float p = 1, i = 0;	66+6C+6F+61+74+70+3D+31+69+3D+30 = **3CA**				3CA
while (i < exponent) {	77+68+69+6C+65+69+3C+65+78+70+6F+65+6E+74+65 = **626**				626
p = p * base;	70+3D+70+2A+62+61+73+65= **2E2**				2E2
i = i + 1;	69+3D+69+2B+31= **16B**				16B
}	-				-

**return p;
}**
	72+65+74+75+72+6E+70 = **310**		**25**	**25**	35A

	TOTAL				**2016**

Path: 1,2,3,4,5,3,6; **TC-5: base = 5, exponent = - 2 (Eq. 5)**					
Program *O* and *P*	operator (sum)	entry	result	exity	Hexa
float Pow (float base, float exponent) {	66+6C+6F+61+74+50+6F+74+65+6E+63+69+61+66+6C+6F+61+74+62+61+73+65+66+6C+6F+61+74+65+78+70+6F+65+6E+74+65 = **E78**	5 + (- 2)			**E7B**
float p = 1, i = 0;	66+6C+6F+61+74+70+3D+31+69+3D+30 = **3CA**				3CA
while (i < exponent) {	77+68+69+6C+65+69+3C+65+78+70+6F+65+6E+74+65 = **626**				626
p = p * base;	70+3D+70+2A+62+61+73+65= **2E2**				2E2
i = i + 1;	69+3D+69+2B+31= **16B**				16B
}	-				-

**return p;
}**
	72+65+74+75+72+6E+70 = **310**		**0+0+4 = 4**	**1**	**315**

	TOTAL				**1FCA**

In this situation with the same code in *O* and *P*, but both containing a logical error, the application with different equations 1 - 4 and TC may result in outputs with the same values, confusing the tester. For example, the entries: **base= 3, exponent= 3** and **base= 3, exponent=** - **3**.

For this reason, equation 5 is applied to find out whether the writing of the *O* and *P* codes are the same, to check whether the program output is the same as the expected output, and whether there is a logical error in the writing of the program *O*.

By applying equation 5, shown in **Table 2A** and **Table 2B**, with TC-4 *(base= 5, exponent= 2)* and TC-5 *(base= 5, exponent= -2)*, it is observed that the program codes for exponential calculation did not present syntactic errors between Program *O* and Program *P*.

The results of the calculation of the parameter *operator* are equivalent in all lines of the code for *O* and *P*, with TC-4 and TC-5, showing that the codes are the same and present the same result in the 6 lines (E78; 3CA; 626; 2E2; 16B; 310 – in hexadecimal).

For the parameter *operator* added to the parameter *entry*, the result is different (*O* and *P* with TC-4 is **E7F** and *O* and *P* with TC-5 is **E7B**), which is expected because these are test cases with different inputs and, in themselves, do not indicate errors.

However, it is observed that, for *O* and *P*, with TC-4, the expected parameter *result* was **25** and the output after execution was also **25**. And with CT-5, the expected parameter *result* would be **0.04 (0+0+4 in hexa)**, but the effective output was **1**. In other words, it was detected that the codes contain an algorithmic error and do not work fully, as for any negative exponent the output will always be **1**.

In this case, the comparison is not made between Programs *O* and *P*, but between the expected results and the actual outputs of *O* and *P*, with different test cases.

The MSUT calculation demonstrates that the codes do not contain syntax or semantic errors, but present different results, indicating a logical error due to non-compliance with functional requirements. Because, for test cases with positive values, the output will be equal to the expected result. However, in test cases with negative values the output will be incorrect, different from the expected result.

This situation would not be detected using equations 1 - 4 because programs *O* and *P* do not present code changes. However, equation 5 demonstrates the equality or difference between algorithms *O* and *P* and between the expected result and the effective output.

Discussion

It is observed that the method with equations 1 – 4 makes it possible to detect errors, both in the general calculation of the metric and in the intermediate portions, whether *c-use*, *p-use*, or, in the output of results when comparing the program documented in the project with the implemented coding.

Furthermore, this metric allows for a more accurate indication of the location of semantic and logical errors in the code, compared to cyclomatic complexity, independent paths, or data flow techniques. This reduces the need to manually check the uses of variables in *def-use* for each variable and in all DFG paths, in addition to avoiding the use of mutants derived from the O program for comparison, since errors are detected directly in the output of the code with the equations presented.

Using equation 5, an error was shown in the output after executing the code and a semantic failure was detected due to algorithmic construction by comparing the expected result and the program output.

In this case, the problem was not in the syntax or data exchange. The program was tested with the TC-5 test case and every time the input was a negative exponent it always returned 1, indicating that it did not operate correctly with negative numbers, however, for every positive exponent the program worked correctly. Therefore, the inclusion of equation 5 in the MSUT was essential to compare the expected result with the generated output and indicate the existing algorithmic flaw.

Conclusion

The MSUT proved capable of identifying different types of semantic and logical errors in programs. In the case of the first example, the metric detected a specific problem in the improper exchange of a relational operator, while in the second example with the exponential calculation the error arose when comparing the output obtained with the expected result. MSUT allowed a detailed analysis of variables and flows, helping to identify problems that could otherwise go unnoticed.

The MSUT demonstrated to be a possible metric for detecting software failures, especially to supplement unit tests with cyclomatic complexity, independent paths and data flow graph techniques. And possibly avoid using mutants.

The metric helps to ensure that the code is specified in accordance with the requirements and that the results present satisfactory quality. In addition to the possibility of automated software testing tools, it suggests continuity for further research in this area.

References

Andrade, S. F. R. (2020). Signature of the Structural Test (SSt)-metric based on data flow test and mutant analysis. *Systems and Computing Magazine - RSC*, *10*(3).

ASCII (American Standard Code for Information Inter change), https://www.ascii-code.com/, Access in: maio/2025.

Barbosa, E. F., Maldonado, J. C., Vincenzi, A. M. R., & Delamaro, M. E. (2006). *Structural and mutation testing in the context of programs*.

Beizer, B. (2003). *Software testing techniques*. dreamtech Press.

Delamaro, m.; jino, m.; Maldonado, J. (2016) Introduction to Software Testing. 2. ed. Brasil: Elsevier Brasil, Campus.

Endres, A. (1975, April). An analysis of errors and their causes in system programs. In *Proceedings of the international conference on Reliable software* (pp. 327-336).

Howden, W. E. (2006). Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, (3), 208-215.

Howden, W. E. (2011, July). Error-based software testing and analysis. In *2011 IEEE 35th Annual Computer Software and Applications Conference Workshops* (pp. 161-167). IEEE.

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

Pressman, R. S. (1995). *Software engineering* (Vol. 6). São Paulo: Makron books.

Pressman, R., & Maxim, B. (2016). *Software engineering*, 8th edição. Mc. Graw Hill.

Rapps, S., & Weyuker, E. J. (1982, September). Data flow analysis techniques for test data selection. In *Proceedings of the 6th international conference on Software engineering* (pp. 272-278).

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

METRIC FOR SOFTWARE UNIT TESTING