IEEE Region 10 Conference, Tencon 92
11th - 13th November, 1992
Melbourne, Australia

A QUANTITATIVE MEASUREMENT FOR DIFFERENT TESTING METHODOLOGIES

Chi-Ming Chung and Wen C. Pai
Graduate Institute of Information Engineering
Tamkang University, Tamshui, Taiwan, R.O.C.

Abstract

Program testing is a singnificant process to assure and control software quality. Many software testing methodologies have been proposed and wildly applied[1]. However, there is few research providing scale measurement of the difference within them. This paper propose a quantitative analysis to measure the difference between ***All-P-USES criterion*** and ***All-EDGES criterion***. A theoretical basis for measuring testing efforts is presented and propose a criterion for selection of testing criteria.

Keywords : Control-Flow Analysis, Data-Flow Analysis, White box testing, Black box testing, Static analysis, Dynamic analysis.

1. Introduction

Software testing is an important process in software quarlity assurance. A stronger testing methodology maybe found more errors of the being tested program, however, the cost of the testing action is higher than a weaker one. It is a trade off between the correctness of the software and its corresponding cost.

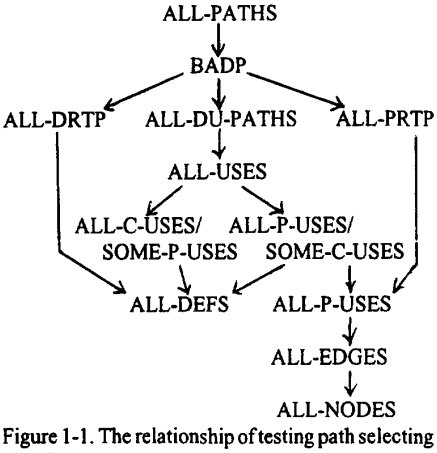

Figure 1-1. The relationship of testing path selecting criteria

A family of testing methodologies have been proposed for choice testing path[1-2]. Figure . 1-1 show the relationship of these testing criteria, where we say criterion C1 is includes criterion C2 iff for every program graph G, any set of complete paths of G that satisfies C1 also satisfies C2 (denoted by C1->C2), and that the criterion C1 and C2 are incomparable iff neither C1->C2 nor C2->C1.

The quantitative analysis for different testing criterion is important to provide a guideline for tester to select a most appropriate criterion. We have presented the quantitative measurement for ***All-edges (All-branches) criterion*** and ***All-nodes (All-statements) criterion*** [3]. This paper provides the quantitative measurement for All-edges criterion and the ***ALL-P-USE*** criterion. In the next section, testing terminologies, definitions and their associated terms are introduced. In section 3 we analyze the singnificant measurement for the all-p-use criterion and all-edes criterion. In the section 4, we rise an algorithm for analysis tested program. The conclusion and the future research suggestions are risen in the last section.

2. Structural Quantitative Analysis Concepts

We define a p-use about a variable X as there is a predicate-use of X in some edge. We denote this by **pu(x)**. The analysis between all-p-uses criterion and all-edges criterion is derived from: under the specific command type, when testing traversal satisfies all-p-uses criterion, how many p-use of some variable will be ***lost*** in all-edges criterion traversal. If the number of the p-use lost is 0 , then these two criteria are equivalence in this command type. When the number of the p-use lost is larger than 0 , then we will discuss the best-case and worst-case subsquently.

For simplifing the analysis, we compare all-p-uses criterion and all-edges criterion based on a subset of "C" language. There are eight basic instructional types in C language. These command types are showed in the figures 2-1 to 2-8.

0-7803-0849-2/92 $3.00 © 1992 IEEE

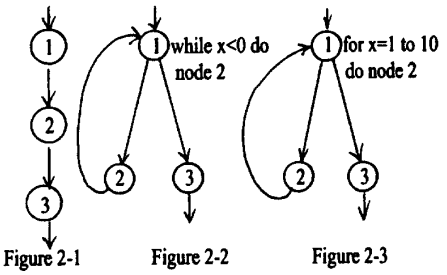

Figure 2-1 sequence command

Figure 2-2 while loop command

Figure 2-3 for loop command

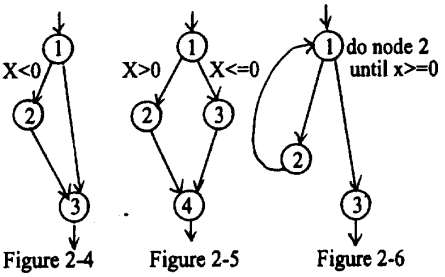

Figure 2-4 if then command

Figure 2-5 if then else command

Figure 2-6 repeat loop command

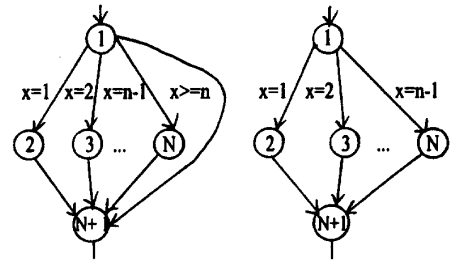

Figure 2-7 switch-case-with-default

Figure 2-8 switch-case-without-default

The following definitions need to be introduced to discuss the difference between all-p-uses and all-edges criterion.

(1). **AP**={the paths that each path is the shortest path satisfy all-p-uses criterion}

(2). **AE**={the paths that each path is the shortest path satisfy all-edges criterion}

(3). **P1&P2** : means the paths set which each path is shorest path traveled one of the P1 and then one of the P2 subsequenty.

(4). != : means *NOT EQUAL*

Definition 2.1 *Lose*

The number of the pu(x) lost in the comparison between all-p-uses and all-edges criterion; denoted by **lose(x)**.

Definition 2.2 *Unlose*

If lose(x)=0 in the comparison between all-p-uses criterion and all-edges criterion about variable X, we say that it is unlose about X; denoted by **unlose(x)**.

Definition 2.3 *Equivalence*

In a specific command type, if lose(x)=0 for every variable X in the comparison between all-p-uses criterion and all-edges criterion, we say that these two criteria are equivalence in this command type; denoted by **E(<command type>)**.

Definition 2.4 *Non-Equivalence*

In a specific command type, if E(<command type>) is not true, then these two criteria are non-equivalence; denoted by **-E(<command type)**.

Since a structural program is a sequential statements of the eight instructions mention above, testing criteria quantitative analysis of a program can be done by treating the program as a combination of these eight instructions. The combination is divided into two types, they are : *Link* and *Nest*.

Definition 2.5 *Link*

Command type T1 link command type T2 iff the combination form between T1 and T2 is (T1, T2), that is, T1 is the predecessor command type of the T2 and T2 is the successor command type of the T1; denoted by **T1+T2**.

For example, the figure 2.9 show the **sequence-command + if-then-command**.

Definition 2.6 *Nest*

Command type T1 nest command type T2 iff the combination form between T1 and T2 is the T2 contained in the T1, that is, T1 contain T2 wholly; denoted by **T1>T2**.

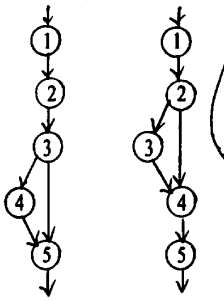

Figure 2.9 sequence+if-then

Figure 2.10 sequence>if- then

Figure 3.1.2-1 example for while-loop command

For example, the figure 2.10 show the **sequence-command > if-then-command**.

3. Quantitative Measurement Between All-p-uses And All-edges

3.1 No-link and No-nest Condition

In this section, we propose a quantitative anlysis for all-p-uses criterion and all-edges criterion for each basic command type.

3.1.1 Equivalence Case

In the equivalence cases, the set of paths that satisfies all-p-uses criterion shall be also satisfies all-edges criterion in each command type.

(1). Sequence execution commands. (see figure 2-1)

Fact: E(<sequence execution commands>)

Proof: In the testing aspect, the sequence execution commands will be tested from the *start node* to the *exit node*, consequently, lose(x)=0. All-p-uses is equivalent to all-edges.

(2). For loop commands. (see figure 2-3)

Fact: E(<for loop commands>)

Proof: In software testing, the specific block or blocks must be executed the specific times under the for-loop command. There are no differences between all-p-uses criterion and all-edges criterion. That is, lose(x)=0 and E(<for loop commands>) established.

(3). If then commands. (see figure 2-4)

Fact: E(<if then commands>)

Proof: In figure 2-4, AP = {(1, 2, 3), (1, 3)}, and AE = {(1, 2, 3), (1, 3)}, such that lose(x)=0 for all variable x p-use in edge (1, 2) and (1, 3). So E(<if then commands>) proved.

(4). If then else command. (see figure 2-5)

Fact: E(<if then else commands>)

Proof: In figure 2-5, AP = {(1, 2, 4), (1, 3, 4)}, AE = {(1, 2, 4), (1, 3, 4)}. i.e., lose(x)=0 and E(<if then else commands>).

(5). Switch case with default commands. (see figure 2-7)

Fact: E(<switch case with default commands>)

Proof: In figure 2-7, AP = {(1, 2, N+1), (1, 3, N+1), ..., (1, N, N+1), (1, N+1)} and AE = {(1, 2, N+1), (1, 3, N+1), ..., (1, N, N+1), (1, N+1)}. So lose(x)=0 and E(<switch case with default commands>).

(6). Switch case without default commands. (see figure 2-8)

Fact: E(<switch case without default commands>)

Proof: In figure 2-8, AP = {(1, 2, N+1), (1, 3, N+1), ..., (1, N, N+1)} and AE = {(1, 2, N+1), (1, 3, N+1), ..., (1, N, N+1)}, then lose(x)=0 and E(<switch case without default commands>).

3.1.2 Non-Equivalence Case

In this type of commands, the exercising testing paths are satisfies all-p-uses criterion but don't need to satisfy all-edges criterion.

(1). While loop commands. (see figure 2-2)

In figure 2-2, AE can be: {(1, 2, 1, 3)} or {(1, 2, 1, 3), (1, 2, 1, 2, 1, 3)} or {(1, 2, 1, 3), (1, 2, 1, 2, 1, 3), (1, 2, 1, 2, 1, 2, 1, 3)}, ..., etc.. It is dependent on the test cases. Subsquently, the lose(x) value (may be larger than 0) is dependent on the different test datas.

Assume the number of the while-loop-commands ***executed*** is a definite value: N, and the increment of each loop is a variable: n. (n =< N). For example, the program may be:

```
N=1
WHILE N<=10 DO
statement 1
.
.
.
statement i-1
N=N+n              * the increment *
statement i+1
.
.
.
statement m (m >= i+1)
ENDWHILE
```

then we have the fact:

Fact: -E(<while loop commands>) and define M=N/n, then:

when M is an (positive) integer:

lose(x)=0 best case

lose(x)=[N/n]-1 worst case

when M isn't an integer:

lose(x)=0 best case

lose(x)=[N/n] worst case

(where [a] means the greatest integer <= the number a)

Proof: See figure 3.1.2-1, it is obvious that the node 2 will be executed [N/n] times when the M is an (positive) integer and that will be executed [N/n]+1 times when the M isn't an integer under all-p-uses criterion. Because the AE is dependent on the test data, if M is an integer we have:

lose(x)=0 when the path contained in AE execute node 2 [N/n] times

lose(x)=1 when the path contained in AE execute node 2 [N/n]-1 times

.
.
.

lose(x)=[N/n]-1 when the path contained in AE execute node 2 one times
Similarly, when the M isn't an integer then we have:
lose(x)=0 when the path contained in AE execute node 2 [N/n]+1 times
lose(x)=1 when the path contained in AE execute node 2 [N/n] times

.
.
.

lose(x)=[N/n] when the path contained in AE execute node 2 one times
i.e., when M is an (positive) integer:
lose(x)=0 best case
lose(x)=[N/n]-1 worst case
when M isn't an integer:
lose(x)=0 best case
lose(x)=[N/n] worst case
and the proof is completed.
(2). Repeat loop commands.
This command types are similar to while-loop-commands except that the condition is evaluated at the end of the loop rather than at the beginning. Assume the number of the node 2 that must be executed (include the first times without pu(x)) is a definite value: N+1, and the increment of each loop is a variable: n, the fact is as the following and the proof will be omitted.
Fact: -E(<while loop commands>) and define M=N/n , then:
when M is an (positive) integer:
lose(x)=0 best case
lose(x)=[N/n] worst case
when M isn't an integer:
lose(x)=0 best case
lose(x)=[N/n]+1 worst case
(similarly, where [a] means the greatest integer <= the number a)

3.2 Link Condition

Fact: if lose(x) of the command type T1 is n1, the lose(x) of command type T2 is n2, then the lose(x) of any link type of two command type (i.e., T1+T2 or T2+T1) is n1+n2.
Proof: Let AE1, AP1 are the AE set and AP set of the first instruction. AE2, AP2 are the AE set and AP set of the second instruction. Therefore :
AE = { AE1 & AE2 }
AP = { AP1 & AP2 }
Since AE1=AP1 and AE2=AP2, i.e., the lose(x) is not affected by the combination. So the results of two linked instructions is n1+n2.

3.3 Nest Condition

(1). Equivalence Case And Equivalence Case
Fact : unlose(x)
Proof: If we define AP1 and AE1 are the AP set and AE set of the first command type, AP2 and AE2 are the AP set and AE set of the second command type. It is trivial that the AP set of the nest case AP(T1>T2) (or AP(T2>T1)) and the AE set of the nest case AE(T1>T2) (or AE(T2>T1)) are:
AP(T1>T2) = AP1 & AP2
AE(T1>T2) = AE1 & AE2
Since AP1=AE1 and AP2=AE2, such that AP(T1>T2)=AE(T1>T2) and the result is proved.
(2). Equivalence Case And Non-Equivalence Case
Fact: the lose(x) of the nest type = the lose(x) of the command type of the non-equivalence case.
Proof: If we define AP1 and AE1 are the AP set and AE set of the first command type, AP2 and AE2 are the AP set and AE set of the second command type, then the AP set of the nest case AP(T1>T2) (or AP(T2>T1)) and the AE set of the nest case AE(T1>T2) (or AE(T2>T1)) are:
AP(T1>T2) = AP1 & AP2
AE(T1>T2) = AE1 & AE2
Since AP1=AE1 but AP2!=AE2, so the lose(x) of the nest type is affected by the second command type only. such that
the lose(x) of this nest type
= the lose(x) of the command type of the non-equivalence case
and the proof is complete.
(3). Non-equivalence Case And Non-equivalence Case
Fact: if the lose(x) of the first command typpe is n1 for best case and m1 for worst case, the lose(x) of the second command type is n2 for best case and m2 for worst case. then the lose(x) of this nest type is:
lose(x)=n1+n2 best case
lose(x)=m1+m2 worst case
Proof:If we define AP1 and AE1 are the AP set and AE set of the first command type, AP2 and AE2 are the AP set and AE set of the second command type, then the AP set of the nest case AP(T1>T2) (or

AP(T2>T1)) and the AE set of the nest case AE(T1>T2) (or AE(T2>T1)) are:
AP(T1>T2) = AP1 & AP2
AE(T1>T2) = AE1 & AE2
But AP1 may or may not equal to AE1 and AP2 may or may not equal to AE2. From the testing aspect, the testing result of this nest type may be one of the following:
(i). AP1!=AE1 and AP2!=AE2
(ii). AP1!=AE1 and AP2=AE2
(iii). AP1=AE1 and AP2!=AE2
(iV). AP1=AE2 and AP2=AE2
Since the lose(x) of the first command typpe is n1 for best case and m1 for worst case. Besides, the lose(x) of the second command type is n2 for best case and m2 for worst case. So the lose(x) of the nest type is:
lose(x)=n1+n1 best case
lose(x)=m1+m2 worst case

Example: While-loop-command Nest While-loop-command

As we described before, the lose(x) of the while-loop-command is:
define M=N/n, then:
when M is an (positive) integer:
lose(x)=0 best case
lose(x)=[N/n]-1 worst case
when M isn't an integer:
lose(x)=0 best case
lose(x)=[N/n] worst case
(where [a] means the greatest integer <= the number a)
See figure 3.3-1, the lose(x) of the pu'(x) satisfies this result described above and pu''(x) satisfies, too. So the total lose(x) is:
lose(x)=0 best case
lose(x)=2*([N/n]-1) worst case when M is an integer
lose(x)=2*([N/n]) worst case when M is not an integer

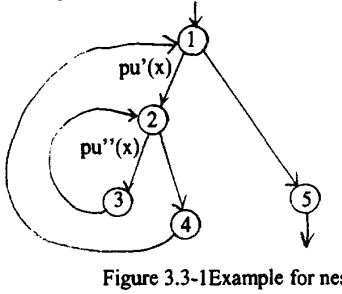

Figure 3.3-1Example for nest

4. Algorithm for analysis different criteria.

The approach for analysis a tested program can be summarized as the following:
Step 1. Decomposed the tested program to individual statement.
Step 2. The individual statement is exercised according to the one of the following:
1. If this statement is belong to the equivalence cases described in this paper, then lose(x)=0.
2. If this statement is belong to the non-equivalence cases, then caculate the lose(x).
Step 3. The trade-off between the effort of the testing and the corresponding correctness can be compared from the lose(x).

5. Conclusions

This paper provides theoretical basis for quantitative analysis within different testing methodologies. It provide the testing information and the criteria of testing methodology selection and that can be extended to analyze quantitative difference within other criteria. Our future researches may be : (1). to have a quantitative analysis within the other criteria showed in the figure.1-1 , (2). introduce a theoretical conclusion that can be referred by more methodologies.

References

[1] S.Rapps, E.J.Weyuker, "Selecting software test data using data flow information", IEEE Trans. on SE, Vol.SE-11, No.4, April 1985, pp.367-375.
[2] Chi-Ming Chung and Wen C. Pai, "A Family of Data-Flow Testing Methodologies," Int'l Journal of Mini and Microcomputers, Vol.13, No.1, 1991.
[3] Chi-Ming Chung, Ying-Hong Wang and Jiunn-Liang Wu, " A Quantitative Analysis Between All-Statements and All-Branches Criterion," Preceeding of National Computer Symposium, 1991, Taiwan, R.O.C.
[4] E. Oviedo, "Control flow, data flow and program complexity", Proceedings of the IEEE Computer Society's fourth International Computer Software and Applications Conference(COMPSAC80), pp.146-152,1980.
[5] Chi-Ming Chung, "Software development techniques -- combining testing and metrics", IEEE region 10 conference, H.K., 1990.