PTMD: Pairwise Testing Based on Module Dependency

Jangbok Kim[1], Kyunghee Choi[1] and Gihyun Jung[2]
[1] Graduate School of Information and Communication, Ajou University,
Suwon, 442-749, South Korea
[2] School of Electrics Engineering, Ajou University,
Suwon, 442-749, South Korea

Abstract: - This paper proposes a modified pairwise test case generation algorithm, named PTMD (Pairwise Testing based on Module Dependency) algorithm. The proposed algorithm produces additional test cases that may not be covered by the typical *pairwise* algorithm due to the dependency between internal function modules of software. The additional test cases effectively increase the coverage of testing without significantly increasing the number of test cases. The performance of proposed algorithm is evaluated with a part of function of *procps*[4], which is a well-known UNIX utility utilized for displaying process information.

Key-Words: - Software Testing, Testcase Generation, Pairwise

1 Introduction

To make software dependable, various software testing methods are widely used in the field. It is ideal to test software with all possible combinations of input parameters, which is not possible. The n-wise test case generation policy is an alternative. The study in [1] shows that testing *n-tuples* of parameters is enough to detect most faults embedded in various systems, when *n* is six. It means that the strategy mostly satisfies testing requirements with smaller number of test cases (compared with all possible cases). Especially, when *n*=2, the strategy is called *pairwise testing* (or *2-way testing*). Pairwise testing requires that for each pair of input parameters, every combination of valid values of these two parameters be covered by at least one test case.

Several ways have been proposed for implementing the policy. Covering array is a mechanism containing a list of test cases satisfying n-wise test case generation policy [7]. Many combinational test case generation algorithms have been studied for creating covering array. The well-known orthogonal Latin square concept was first introduced for creating covering array by Mandl[6]. Brownlie et al. and Williams et al. also used the orthogonal Latin square for creating covering array for interaction test.[9,10] In [3,8], Cohen et al. proposed *Automatic Efficient Test Generator* (*AETG*) *System*. *AETG* system adopts an algorithm to generate all possible pairs of input parameters. The system uses a greedy approach to select input parameters in a fashion that can minimize uncovered pairs. Kuo-Chung Tai et al. introduced In-Parameter-Order (IPO) algorithm in [2]. IPO algorithm generates Pairwise test cases using the first two input parameters among many input parameters and then generates other cases adding other input parameters. James Bach built a test case generation tool, named *Allpairs*, utilizing PERL [5]. *Allpairs*, that satisfies the philosophy of pairwise testing, also uses a greedy approach. But Allpairs generates test cases with input parameters that have been used least frequently.

Though pairwise testing generates a small number of effective test cases, it does not produce test cases considering the dependency between internal modules of systems. We say that there is a dependency between internal modules of a system, $\boldsymbol{S}$, when an output of a module is an input of other module of $\boldsymbol{S}$. For example, let a system $\boldsymbol{S}$ have three input parameters $\{\boldsymbol{X},\boldsymbol{Y},\boldsymbol{Z}\}$ and two modules, $\boldsymbol{op_1}$ and $\boldsymbol{op_2}$. $\boldsymbol{X}$ and $\boldsymbol{Y}$ are the inputs of $\boldsymbol{op_1}$ and $\boldsymbol{W}$ is an output of $\boldsymbol{op_1}$. That is, $\boldsymbol{W}=\boldsymbol{op_1}(\boldsymbol{X},\boldsymbol{Y})$. If $\boldsymbol{W}$ and $\boldsymbol{Z}$ are inputs of $\boldsymbol{op_2}$, then we say that there is a dependency between $\boldsymbol{op_1}$ and $\boldsymbol{op_2}$. In the case that there is a dependency between internal modules, the test cases generated by pairwise testing strategy may not include some pairs that are sometimes crucial to test systems. A system with module dependency can be modeled as a tree structure. For instance, let a system $\boldsymbol{S}$ with three Boolean input parameters, $\{\boldsymbol{X},\boldsymbol{Y},\boldsymbol{Z}\}$. $\boldsymbol{S}$ is expressed as $\boldsymbol{S = (X \text{ and } Y) \text{ and } Z}$. Fig 1 illustrates the tree structure showing the dependency between two modules. ***op*$_1$** and ***op*$_2$** are both "***and***" operation in the example. ***op*$_2$** also acts as the root of tree structure.

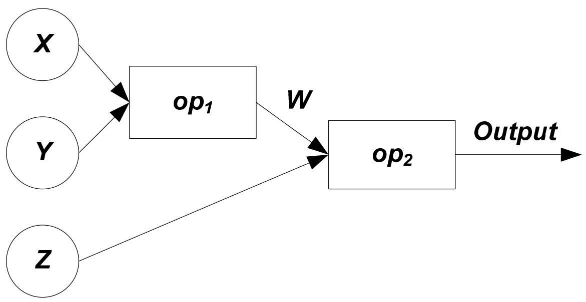

Fig. 1. Tree structure with the dependency between function modules.

X	***Y***	***W***	***Z***
T	T	T	T
T	F	F	F
F	T	F	F
F	F	F	T

Table 1. Pairwise test cases

The typical pairwise testing algorithm generates four test cases for ***S*** as depicted in Table 1. Values of ***W*** generated by ***X***, ***Y*** pairs are also shown in the Table. For testing ***op2*** module with a pairwise testing strategy, we need four test cases generated with ***W*** and ***Z***. However, one pairwise test case (***W***, ***Z***) = (T, F) is missing in the table. The case (***X***,***Y***,***Z***) = (T,T,F) has to be added to the table.

This paper proposes a modified *pairwise* testing algorithm, that generates pairwise test cases taking account of dependency between internal modules of software systems. The modified algorithm increases testing coverage without significantly increasing the number of test cases, compared with that of the typical pairwise testing algorithm. The paper also presents the outcome of empirical study to show the feasibility of proposed algorithm.

Chapter 2 presents the philosophy and details of our proposed algorithm. The way that the algorithm works is explained through an example in Chapter 3. The performance evaluation also described in Chapter 3. Finally, the paper is wrapped up in conclusion.

2 Proposed Algorithm

As shown in Alg 1, the proposed algorithm consists of three main parts: test case generation by pairwise strategy, test case generation considering module dependency and merging the two test case sets. The pseudo code of proposed algorithm applying to a system *S*, which is modeled as a tree structure, looks like Algorithm 1. Here are the definitions of notations used in the algorithm description. *nd* is a node

PTMD algorithm (*S*)
Apply a pairwise algorithm to *S* and get test case set *PT*;
Let *TC(root of S)* be a test set generated by applying *ForOneNode* to the root of *S*;
Merge *PT* and *TC(root of S);*

Procedure ForOneNode (nd)
if nd is a leaf node then
$TC(nd) = \{(nd,v_1(nd)) , (nd,v_2(nd)) , \ldots, (nd,v_{N(nd)}(nd))\}$
else
for each child node C_i *of nd do*
build $TC(C_i)$ *by calling* $ForOneNode(C_i)$ *recursively;*
for each $t_i =(C_1,v_i(C_1))\ (C_2,v_i(C_2))\ldots(C_{N(nd)},v_i(nd))$ *in* $PW(C_1,C_2,\ldots,C_{N(nd)})$ *do*
construct a test case t_{nd} *by replacing every* $(C_k,v_i(C_k))$ *in* t_i *with*
$t_{ci} \in TC(C_i)$ *such that* $output(C_i,t_{ci}) = v_i(C_k)$ *,and mark* t_{ci} *as covered;*
insert t_{nd} *into TC(nd);*
end for
while there exist uncovered test cases t_{ci} *in any* $TC(C_i)$ *do*
construct t_{nd} *by concatenating* t_{ci} *with any test cases in* $TC(C_j)$ *for* $j(\neq i)=1,2,i-1,i+1,..,N(nd)$*, and mark those test cases as covered;*
insert t_{nd} *into TC(nd)*
end while
end if

Alg 1. Pseudo code of the modified Pairwise algorithm (operator). $v_i(nd)$ is the i^{th} output value of *nd*. If *nd* is a leaf node (that is, *nd* is an input parameter of system), $v_i(nd)$ becomes the i^{th} value of itself and *N(nd)* is the number of parameters that *nd* has. Otherwise, C_i becomes the i^{th} child node of *nd* and *N(nd)* is the number of child nodes of *nd*. By a test case $t=p_1p_2p_3\ldots$, we mean a sequence of (parameter, value) pairs p_i. For example, t= (*X*,T)(*Y*,T)(*Z*,F) denotes a test case, where *X*, *Y* and *Z* are parameters, and *T* and *F* denote their values. The expression (*X*,*Y*,*Z*) = (T,T,F) in the previous example is expressed as (*X*,T)(*Y*,T)(*Z*,F) in the algorithm for convenience. output(*nd*,*t*) denotes the value to be produced when *t* is applied to *nd*. $PW(nd_1,nd_2,\ldots)$ is the set of test cases for the output values of $nd_1,nd_2,\ldots$ produced by the typical pairwise testing algorithm like AETG or IPO [2,3]. In the above example, $PW(C_1, C_2)$ = {(*W*,T)(Z,T), (*W*,T)(*Z*,F), (*W*,F)(Z,T), (*W*,F)(*Z*,F)}.

2 Performance Evaluation

3.1 A system example with module dependency

UNIX systems use *procps* utility to show process information. Since it is too complicated to describe the whole function of *procps*, we simplified the function without hurting the way it runs. *procps* activates processes depending on the selected options. There are seven allowable options, which are '*-e*', '*-a*', '*-d*', '*T*', '*a*', '*g*', '*r*'. Table 2 shows how *procps* behaves depending on different options.

Input options	Description
-e	selects all processes
-a	selects processes on a current terminal without session leader
-d	selects processes without session leader
T	selects processes on this terminal
a	selects all processes on a terminal, including those of other users
g	selects all processes with current user, including session leader
r	restricts output to running processes

Table 2. Description of input options

The options are used either separately or together. When multiple options are used, the options have priorities and some options are not allowed to be used together. Fig. 2 illustrates how the processes are selected with the options.

L1: S = NULL;
L2: Get S_T, S_g, S_a, S_{-a}, S_{-d}, S_{-e};
L3:
L4: **if** (-*a*=true \|\| -*d*=true) && (*a*=true\|\| *g*=true) **then return** error;
L5: **if** *T* = true **then** $S := S \cup S_T$;
L6: **if** *g* = true **then** $S := S \cup S_g$;
L7: **if** *a* = true **then** $S := S_a$;
L8: **if** -*a* = true **then** $S := S \cup S_{-a}$;
L9: **if** -*d* = true **then** $S := S \cup S_{-d}$;
L10: **if** -*e* = true **then** $S := S_{-e}$;
L11: **if** *r* = true **then** *S* := *running processes in*
L12: *S*;
L13:
return *S*;

Fig. 2 Process selection in *procps*

S is the set of processes selected by the options. S_T, S_g, S_a, S_{-a}, S_{-d}, S_{-e} are the processes selected by options '*T*', '*g*', '*a*', '*-a*', '*-d*', '*-e*' and '*r*', respectively. *L4* indicates that option pairs ('*-a*' and '*a*'), ('*-a*' and '*g*'), ('*a*' and '*-d*') and ('*-d*' and '*g*') cannot be applied. *L5* ~ *L11* indicate that the processes are selected by the corresponding options, regardless of other options. Option '*r*' selects the processes currently running in a system.

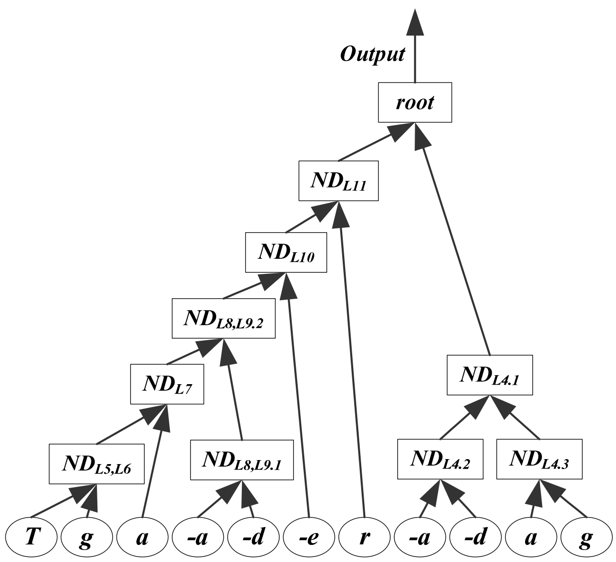

Fig. 3 Module dependency tree of *procps*

Fig 3 shows a tree structure considering the module dependencies of processes in *procps*. What we mean module dependencies here is which processes should be selected prior to other processes and which processes need to be selected without taking into account of selecting other processes. In the figure, $ND_{L4.1}$, $ND_{L4.2}$ and $ND_{L4.3}$ are 'and', 'or' and 'or' operators, respectively. They present the way to select processes in *L4* of Fig 2. $ND_{L5,L6}$ is a 'union' operator in *L5* and *L6* of Fig 2. ND_{L7}, ND_{L10} and ND_{L11} corresponds to *L7*, *L10* and *L11*, respectively. $ND_{L8,L9.1}$ and $ND_{L8,L9.2}$ are 'union' operators and correspond to *L8* and *L9*. The *root* node returns 'error' if $ND_{L4.1}$ is 'true'. Otherwise the root node returns the outcome of operator ND_{L11}. Table 3 summarizes the actions of nodes.

Node	Operation
Root	**if** ND_{L7} = TRUE **then return** error; **else return** ND_{L11};
ND_{L11}	*S*:=*running processes in* ND_{L10}; **return** S;
ND_{L10}	**if** -*e*=true **then** S:=S_{-e}; **else** S:=$ND_{L8,L9.2}$; **return** *S*;
$ND_{L8,L9.2}$	S:=$ND_{L8,L9.1} \cup ND_{L7}$; **return** *S*;
ND_{L7}	**if** *a*=true **then** S:=S_a; **else** S:=$ND_{L5,L6}$; **return** *S*;
$ND_{L5,L6}$	*S*:=NULL; **if** *t* = true **then** S:=S_t; **if** *g* = true **then** S:=$S \cup S_g$; **return** *S*;
$ND_{L8,L9.1}$	*S*:=NULL; **if** –*a* = true **then** S:=S_{-a}; **if** –*d* = true **then** S:=$S \cup S_{-d}$; **return** *S*;
$ND_{L4.1}$	**if** $ND_{L4.2}$=true **and** $ND_{L4.3}$=true **then return** true; **else return** false;
$ND_{L4.2}$	**if** -*a* = true **or** -*d* = true **then return** true; **else return** false;
$ND_{L4.3}$	**if** *a* = true **or** *g* = true **then return** true; **else return** false;

Table 3. The actions of nodes

3.2 Comparing the number of test cases

The input parameters '-*e*','-*a*','-*d*','*T*','*a*','*g*','*r*' have either 'true' or 'false' values. Therefore the number of all possible cases is 128. A Pairwise test case generation mechanism, Allpairs, generates 8 cases. And the proposed test case generation algorithm generates 48 cases. That is, the proposed algorithm generates about 40 more test cases than the pairwise test case generation algorithm, for taking care of the dependencies among modules.

3.3 Comparing the fault detection coverage

To see how much more the proposed algorithm increases the coverage of testing with the increase in the number of test cases, we intentionally inserted various faults in the *procps* model and compared the number of faults found with the test cases by the proposed algorithm and a typical Pairwise algorithm, Allpairs.

Before	After Modified
==	!=
||	&&
&&	||
S_T	"S_g", "S_a", "S_{-a}", "S_{-d}", "S_{-e}"
S_g	"S_T", "S_a", "S_{-a}", "S_{-d}", "S_{-e}"
S_a	"S_T", "S_g", "S_{-a}", "S_{-d}", "S_{-e}"
S_{-a}	"S_T", "S_g", "S_a", "S_{-d}", "S_{-e}"
S_{-d}	"S_T", "S_g", "S_a", "S_{-a}", "S_{-e}"
S_{-e}	"S_T", "S_g", "S_a", "S_{-a}", "S_{-d}"

Table 4. Modification in processes and operators

We generated three different types of faults. And we measured the number of faults the two test cases found in the *procps* module containing the generated faults. The first fault type was generated by modifying the processes or operators in *L4 ~ L11* of Fig 2. Table 4 summarizes the details of modification. This fault type mimics typos possibly occurred during coding.

The second fault was generated by exchanging the order of conditions and their corresponding actions. This fault type mimics programmer's misunderstanding the logic. With this type faults, *procps* may operate quite differently and sometimes this type of faults can down system. The last fault type was made by deleting the conditions and actions, which imitates programmers' mishandling the program. Combining the three fault types, we injected 80 faults in the modeled *procps*.

Table 5 illustrates the numbers of test cases and the faults found by the test cases generated by the proposed algorithm, a typical pairwise algorithm and all possible combinations.

	Proposed	Pairwise	All combinations
No. of test cases	48	8	128
No. of found faults	71	27	76

Table 5. Number of average test cases and found faults

Since proposed algorithm adds extra test cases to those generated by the pairwise algorithm, it is natural that proposed algorithm found more faults that the pairwise algorithm. By including about 40 tests cases more to those by the typical pairwise algorithm, it was possible to find 44 more faults that would not have been found if the extra cases were not tested. The faults found by the extra effort may be sometimes serious and critical.

Considering that executing more test cases is not a big deal at all in most recent fast systems, it may be worth to test some more cases if the extra test cases significantly improve the quality of software system. One interesting thing in the above example is that four intentionally inserted faults cannot be found even with all combinations. That is, even all possible input parameter combinations cannot find some faults. For example, the faults inserted by exchanging *L4* with *L5* produce the same outputs but they are different.

4 Conclusion

This paper proposed a test case generation algorithm that covers up a blind spot of the typical pairwise testing algorithm. The proposed algorithm generates additional test cases, considering the dependencies among function modules in software. The performance of proposed algorithm is compared with that of the typical pairwise algorithm using a simplified *procps* utility. By adding more test cases to those generated by the typical pairwise algorithm, the proposed algorithm can find some forbidden faults that may be serious sometimes.

However, one of obstacles to use the proposed algorithm is that test people need to know the dependencies between software modules. However, usually many delicate systems are tested by people who know the systems in detail. Thus, the obstacle may not be that serious in the real world.

References:

[1] D. Richard Kuhn, Dolores R. Wallace, and Albert M. Gallo Jr., Software fault interactions and implications for software testing, *IEEE Transactions on Software Engineering, Vol. 30, No. 6,* June 2004, Page(s):418 – 421.

[2] Kuo-Chung Tai and Yu Lei, A Test Generation Strategy for Pairwise Testing, *IEEE Transactions on Software Engineering, Vol. 28, No. 1,* January 2002, Page(s):109-111.

[3] D. M. Cohen, Siddhartha R. Dala, Michael L. Fredman, and Gardner C. Patton, The AETG System: An Approach to Testing Based on Combinatorial Design, *IEEE Transactions on Software Engineering, Vol. 23, No. 7*, January 1997, Page(s):437-444.

[4] Procps, *http://procps.sourceforge.net*

[5] James Bach, Allpairs, *http://www.satisfice.com*

[6] R. Mandl, Orthogonal Latin Squares, An Application of experiment design to compiler testing, *Communications of the ACM, 28(1),* October 1985, 1054-1058

[7] M.B Cohen., P.B. Gibbons, W.B. Mugridge and C.J. Colbourn, Constructing test suites for interaction testing, Software Engineering, 2003. *Proceedings. 25th International Conference*, May 2003, 38-48.

[8] D.M. Cohen, S.R. Dalal, M. L. Freedman, and G.C. Patton, Method and system for automatically generating efficient test cases for systems having interacting elements, *United States Patent,* Number 5,542,043, 1996.

[9] R. Brownlie, J. Prowse, and M. S. Padke, Robust testing of AT&T PMX/StarMAIL using OATS, *AT&T Technical Journal, 71(3)*, 41-7, 1992.

[10] A. W. Williams and R. L. Probert, A practical strategy for testing pair-wise coverage of network interfaces, *In Proc. Seventh Intl. Symp. on Software Reliability Engineering*, 1996, 246-254.