Improvement of the Baseline Method in Structural Testing

Jifeng Chen[1,*], Hao Peng[1] and Xingxing Xie[2]

[1]*Hunan International Economics University, Changsha 410205, China*

[2] *Software School, Hunan University, Changsha 410082, China*

Abstract: An algorithm based on the generation for baseline path is proposed in this paper. Making use of it, the original baseline path can be generated randomly and automatically. In respect of traversing the basic path group, the algorithm introduces the method of depth-first search to avoid storing more information of branch node and decrease the time of linear traversal. Using the column with S-node with no in-degree and the row with E-node with no out-degree, the case of nodes' value and the number of value cases are stored for reducing the waste in storage. Furthermore, giving priority to the behind path-node when generating a path, the time of judgment for loops is effectively reduced. Theoretical analysis and experimental results show that this new algorithm has an advantage over the original Baseline Method in efficiency and space complexity of the test path generation. The average space complexity is decreased from $O(n)$ to $O(\log n)$, and the time complexity is also reduced to a certain extent.

Keywords: Baseline path, Control-flow coverage, Data-flow coverage, Software testing, Test path.

1. INTRODUCTION

The nature of software testing is to determine a set of test cases for something which needs be tested and execute the test cases to find bugs in software. There are two basic methods which can be used to identify test cases, functional testing (also known as black-box testing) [1-3] and structural testing (also known as white-box testing) [4-6]. Therefore, the technology of test data generation is usually divided into function-oriented [7, 8] and structure-oriented [9, 10] test data generation technology.

The question of generating the path based on the criterion (remembered as Q_p) is defined as: For a given coverage criterion C, how to generate path P which is set to achieve the coverage requirements described as C in the structural testing.

As there is an executable path P for each test data $\overline{d}$, the criterion been used to describe the adequacy of test data also apply to test path, and the same in semantics, i.e. $C(m,D) = true \Rightarrow C(m,P) = true$.

There are two coverage criteria in structural testing: Control-flow coverage [11-13] and Data-flow coverage [14-16]. The control-flow coverage criterion is chiefly discussed in this paper.

2. RELATED WORK

The control-flow coverage criteria include state-ment coverage, branch coverage, condition coverage, condition /decision coverage, complete path coverage, basis path coverage, Z-path coverage, and so on. The complete path coverage is one of the strongest coverage criteria. Even for a small program, the number of paths may be enormous, so the criterion has no practical application most of the time. Therefore, the coverage criteria which is relatively weak are adopted. Although the statement coverage is regarded as the most weak coverage criterion, due to its poor error detection ability, the branch coverage is actually the most weak coverage criterion [17].

Among the control-flow coverage criteria mentioned above, basic path coverage criterion meets the branch coverage standard [18]. It is defined as follows:

If and only if path P includes $V(G)$ paths which are linear independence, D meets the basic path coverage criterion C, that is $\mathrm{C}(m,D) = true \Leftrightarrow \exists Q((Q \subseteq P) \wedge LI(Q) \wedge (|Q| = V(G))) = true$, $LI(X)$ means that all of the paths in X are linear independence. Basic path coverage is also called the coverage of Cyclomatic Complexity. The test path is generated based on the basic path coverage in this paper. i.e. by analyzing the cyclomatic complexity of the control flow graph, a set of basic paths (which are executable) is derived based on the program control flow graph (PFG) [19].

2.1. The Principle of the Basic Path Coverage Criterion

The basic path coverage criterion was proposed by McCabe in 1976. The basis path test is also called the cyclomatic complexity test due to the cyclomatic complexity equal to the number of independent paths. In graph theory, the cyclomatic complexity is defined as $V(G) = |E| - |N| + p$, in which the P is the number of the connected component. If the direction of PFG isn't in consideration, then $V(G) = |E| - |N| + 1$. However, PFG is not strongly connect-

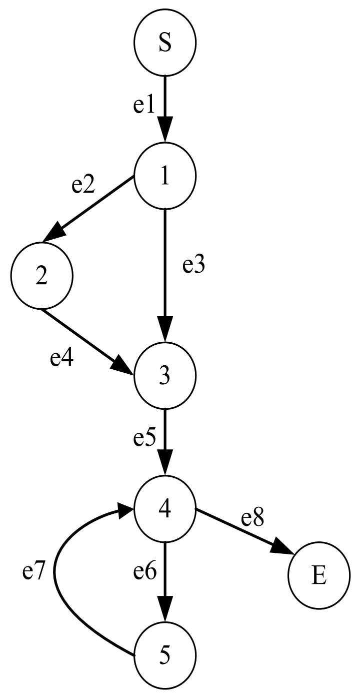

Fig. (1). A program's PFG (PFG-1).

ed, the solution is to add a edge from the Exit to the Entry, so that the cyclomatic complexity of the PFG is $V(G)=|E|-|N|+2$, just equals to the number of independent paths. Normally, the edge from the Exit to the Entry in the PFG isn't introduced, but the virtual edge is assumed to exist when calculating cyclomatic complexity [19].

The basic idea of cyclomatic complexity is similar to any vector in the vector space can be expressed by one of the baselines in this vector space in Linear Algebra. Any program's PFG can be regarded as a matrix which is called $A_{s\times|E|}$, in which s is the number of paths, $|E|$ is the number of edge. All of the paths in the PFG of Fig. (**1**) are (Simplicity, Entry expressed by *S*, Exit expressed by *E*):

$p1 =< S,1,2,3,4,E >$

$p2 =< S,1,3,4,E >$

$p3 =< S,1,2,3,4,5,4,E >$

$p4 =< S,1,3,4,5,4,E >$

$p5 =< S,1,2,3,4,5,4,5,4,E >$

$p6 =< S,1,3,4,5,4,5,4,E >$

Expressed by a matrix $A=\begin{pmatrix} 11011001 \\ 10101001 \\ 11011111 \\ 10101111 \\ 11011221 \\ 10101221 \\ \cdots \end{pmatrix}$, in where one row denotes one path, the column denotes the number of the path travel $e1, e2, \ldots, e8$, that is $A=(p0, p1, \ldots, ps)^T$. According to the knowledge of Linear Algebra [20], any matrix only has one rank and less than or equal to the number of columns. This means that, no matter how many possible paths there are, the rank of matrix will not be over the edges of program control flow graph. In fact, the rank of matrix is the cyclomatic complexity of program control flow graph. The greatest of the linearly independent vectors is a vector in the vector space, which can be expressed by all of the vectors in the vector space. Corresponding to the matrix, vectors express paths and cyclomatic complexity is the number of linearly independent paths. All the paths can be expressed by some paths whose number is cyclomatic complexity.

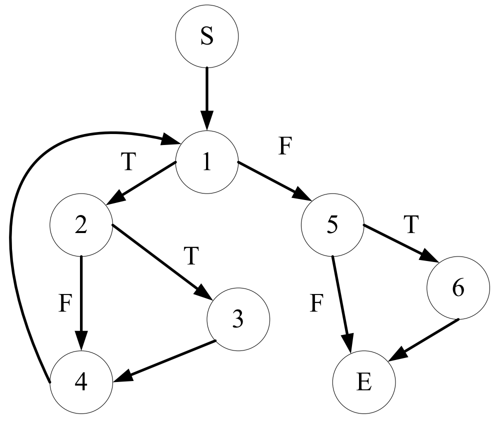

Fig. (2). Example of program flow graph for the baseline method (PFG-2).

In the case of the existence of loops, we have proved that the existence of the loop will increase the number of paths, but cyclomatic complexity will not exceed an upper limit which is the number of edge above. So the increase of paths caused by the loop will not lead to the augmentation of cyclomatic complexity.

2.2. Baseline Method

Using Gaussian Elimination, we can find out matrices' radix corresponded by the set of any given paths, i.e. the number of non-zero row vector after elimination is completed. If there is no zero-vector when elimination is completed, all the paths are linearly independent. If the vector of rank equals to it's cyclomatic complexity, the set of given paths is one of the corresponding program flow graph's radix. However, it is worth noting that, there is a corresponding vector for each path, but it is not assured that there is a corresponding path for each vector. This is very obvious, such as an addition to the first component, the rest are non-zero vector which does not exist on the path from Entry to Exit in the corresponding topology. This fact shows that we must combine with the program flow graph when the base paths are generated.

The baseline Method which is used to generate a group of the base paths has been proposed in [18]. Fig. (**2**) is taken as an example to illustrate its basic idea below.

In (Fig. **2**), The edges of branch are marked by T(True) and F(False) which show that the direction when running the program by different values. S and E express Entry and Exit respectively. Because all the sentences represented by nodes aren't known, we take any one path as the baseline. If 1 = T, 2 = T, 1 = F, 5 = F, then $<S,1,2,3,4,1,5,E>$ is the baseline path. Note: If a branch node is passed again for the loop, we don't think it is a new branch node. So the value of the node needn't change to generate a new path. There are three branch nodes in the baseline path, that is 1, 2 and 5 (the second one is not branch node, because it is passed again for the loop).

While reaching the first branch node along the baseline path, we change the value of branch node 1 to generate the second path, 1 = F, and then it returns to the baseline path, so when it reaches E along the baseline path, the second path is $<S,1,5,E>$. At this time, the first branch node has no other values.

While reaching the second branch node along the baseline path, we change the value of branch node 2 to generate the third path, 1 = T, 2 = F, and then it returns to the baseline path, the third path is $<S,1,2,4,1,5,E>$. At this time, the second branch node has no other values.

Finally, there are no other branch nodes in the baseline path. The second path is taken as the new baseline path. The branch node of baseline path is 5. Reaching 5 along the baseline path at first and changing the value of node 5, the fourth path is obtained, and node 5 has no other value. After all the branch nodes are processed, the generation for the base paths is finished. So the base paths are $<S,1,2,3,4,1,5,E>$, $<S,1,5,E>$, $<S,1,2,4,1,5,E>$, $<S,1,5,6,E>$. If the Exit along the baseline path isn't reached while returning to the baseline path, the paths may have a linear correlation. For example, there may be $<S,1,2,3,4,1,2,4,1,5,E>$ and $<S,1,2,4,1,2,3,4,1,5,E>$.

T and F are added to every branch in the PFG of the example above. But in fact it is not important to choose which branch each time. It is shown here only to illustrate expediently.

3. IMPROVEMENT OF THE BASELINE METHOD

3.1. The Basic Idea

It needs to select "typical" path that the Baseline Method's requires when selecting the baseline path. This is more suitable for manual methods, but in more complex program, determining the path of a typical baseline path manually is not easy to obtain and is prone to error. In addition, because Baseline Method begins to deal from the first branch node of baseline path, all the branch nodes in the baseline path have to store their information in the new paths which are generated by the other circumstances (such as which are the branch nodes? how many times they have been visited). Its space complexity is $O(n)$, average space comlexity is close to $O(n)$ which can be considered to be $O(n)$, in which n is the number of the branch nodes in the PFG.

Considering the first question, any one path can be chosen. It is not as typical as the path in the Baseline Method, but easy to automate. To the second question, in order to avoid storing more branch nodes' information, we can use a similar depth-first algorithm, beginning to tackle from the last branch node in the baseline path. Although its space complexity still is $O(n)$, its average space complexity is $O(\log n)$.

What's more, it can effectively avoid the repeated paths which are brought by the loop to introduce PFG. In actual testing, as a large number of nested path exist to increase the number of loops, the key to the path search algorithm is to solve the problem of loops [21]. In order to reduce the judgment of loop, the back node in the path is preferentially chosen when generating the paths (the node can be regarded as the largest number of nodes). The path which is not through the loop is generated first. Once being back to the node, the loop should be exited along the path when the path including the loop is generated.

3.2. Description of the Algorithm

According to the ideas above, at node 1, selecting 5 as a priority, and then exiting directly, so that the baseline path is $<S,1,5,E>$. Next, change the value of node 5, the second path is $<S,1,5,6,E>$. After that, change the value of node 1, selecting 5 as a priority at node 2, then return to 1. At this time returning to baseline path, to the Exit directly along the baseline path, so the third path is $<S,1,2,4,1,5,E>$. If the first path which is randomly generated by the algorithm above is $<S,1,2,3,4,1,5,E>$, it is necessary to choose a path to exit when being through the node 1 on the second time. Now, the improved algorithm is described as follows:

Input: The program flow graph is expressed by the adjacency matrix G[NUM][NUM], G[i][0] is used to mark the last value, G[NUM-1][i] is used to record the number of the value program where the column node(ie, node j) is located at, the number of node and the subscript of array corresponds with table indexNodeMap.

Output: a set of basic paths.

S1:Start from Entry node S, choose a path to Exit node E randomly, and set the number of current path currPathNum as 0. When meeting a branch node, select the largest number of all the follow-up branch nodes in the current path as the next node. At the same time, push the branch node's information into the stack(that is the node's information which G[NUM-1][j] != 0), the format is below:

```
struct NodeInfo
{
  int currNum; // The number of current paths
  int branchNodeNum; // The number of branch nodes
  NodeInfo(int first, int sec)
  {
   currNum = first;
   branchNodeNum = sec;
  }
```

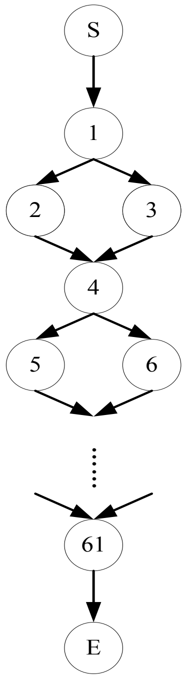

Fig. (3). Program flow graph for experiment (PFG-3).

}

S2: If the stack is empty, go to S4. Otherwise, take out the Top-elements, to assign values as below:

□ ++currPathNum; // The total number of path add 1

□ baselineNum=currNum;// Adjust the baseline path as the current path

□ nextNodeNum = G[branchNodeNum][0]

//the first non-zero element in the left

□ G[branchNodeNum][j]--; // The num of the node's value program minus 1

□ If G [branchNodeNum][j] is 0, push out the node's information of this branch from the stack;

□ If nextNodeNum is Exit, the path is generated successfully, go to S2. Otherwise, go to S3;

S3: Intercepting the list from S to node branchNodeNum from the baseline path marked by baselineNum as the beginning of the current path, and then to find a path to E starting from nextNodeNum at the base path marked by baselineNum, If the follow-up node is existing in the baseline path, intercepting the list from this node to E as the latter part of the current path. When confronting with the branch node, treat it with the method in S1. Go to S2.

S4: The generation of baseline paths is finished.

4. EXPERIMENT

Take PFG-3 (Showed in Fig. **3**) (The PFG-3 expresses a program flow graph with 20 branch nodes) as an example, to verify the feasibility of the algorithm:

$$\begin{bmatrix}
63 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & \dots & \dots & & & \\
63 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & \dots & \dots & & & \\
63 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & \dots & \dots & & & \\
63 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & & & \dots & \dots & & & \\
63 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & & & \dots & \dots & & & \\
63 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & & & \dots & \dots & & & \\
63 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & & & \dots & \dots & & & \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
63 & & & & \dots & \dots & & & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
63 & & & & \dots & \dots & & & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
63 & & & & \dots & \dots & & & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
63 & & & & \dots & \dots & & & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
63 & & & & \dots & \dots & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
63 & & & & \dots & \dots & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
63 & & & & \dots & \dots & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
63 & 1 & 2 & 1 & \dots & \dots & 1 & 1 & 1 & 2 & 1 & 1 & 2 & 1 & 1 & 2 & 1 & 1 & 1 & 0
\end{bmatrix}$$

Fig. (4). The adjacency matrix G[NUM][NUM].

1) By analyzing Fig. (**3**), the input of program flow graph expressed by the adjacency matrix G[NUM][NUM] is obtained as Fig. (**4**):

2) By importing the adjacency matrix G[NUM][NUM] as Fig. (**4**), the output on a set of basic paths is obtained as follows:

1:<*S*,1,3,4,6,7,9,10,12,13,15,16,18,19,21,22,24,25,27,28,30,
31,33,34,36,37,39,40,42,43,45,46,48,49,51,52,54,55,57,58,60,61,*E*>

2:<*S*,1,3,4,6,7,9,10,12,13,15,16,18,19,21,22,24,25,27,28,30,31,
33,34,36,37,39,40,42,43,45,46,48,49,51,52,54,55,57,58,59,61,*E*>

3:<*S*,1,3,4,6,7,9,10,12,13,15,16,18,19,21,22,24,25,27,28,30,31,
33,34,36,37,39,40,42,43,45,46,48,49,51,52,54,55,56,58,60,61,*E*>

4:<*S*,1,3,4,6,7,9,10,12,13,15,16,18,19,21,22,24,25,27,28,30,31,33,
34,36,37,39,40,42,43,45,46,48,49,51,52,53,55,57,58,60,61,*E*>

. . . . .

20:<*S*,1,3,4,5,7,9,10,12,13,15,16,18,19,21,22,24,25,27,28,30,31,
33,34,36,37,39,40,42,43,45,46,48,49,51,52,54,55,57,58,60,61,*E*>

21:<*S*,1,2,4,6,7,9,10,12,13,15,16,18,19,21,22,24,25,27,28,30,31,33,
34,36,37,39,40,42,43,45,46,48,49,51,52,54,55,57,58,60,61,*E*>

According to the description of the improved algorithm, the result which are obtained from analyzing Fig. (**3**) is fully in keeping with the situation of the output above.

3) In order to verify the improved algorithm in respect of time complexity, PFG-3 such as Fig. (**3**) is taken as an instance to measure the executive time. The result is in (Table **1**).

he experimental environment of test is: CPU, Pentium 1.60GHZ; Memory,512M; Operating System, Windows XP SP3. The test results of the executive time shows that the algorithm's average time in this paper is less than that in the baseline method.

Table 1. The Execution Time on PFG-3 (ms).

The NUM of Experiment	1	2	3	4	5	6	7	8	9	10	Average Time(ms)
Baseline Method	93	93	78	78	78	93	78	93	78	93	85.5
Algorithm in This Paper	78	78	62	75	63	62	75	62	75	62	69.2

CONCLUSION

Comparing with the baseline method, the improved algorithm has some advantages as follows:

1) When selecting the first basic path, the improved algorithm is automatic in the selection, and it can make that the generation of path simpler from the data of the experimental results. It can be considered as having generated the shortest path from S to E.

2) Using the depth-search method, the information of current and follow-up nodes is stored. The time for linearly traversing the entire basic paths which have been generated is eliminated. Experiments show that the time complexity is reduced.

3) If the case of nodes' value and the number of the value case are added into adjacency matrix, while using the column where the S-node with no in-degree is located and the row where the E-node with no out-degree is located at, not only has the program flow graph not been affected, but also the waste on storage space is reduced.

CONFLICT OF INTEREST

The authors confirm that this article content has no conflict of interest.

ACKNOWLEDGEMENTS

The research was supported by the Science and Technology Project of Hunan Province (No. 2014FJ3040) and Scientific Research Fund of Hunan Provincial Education Department (No. 14B104), China.

REFERENCES

[1] W. L. Andrade, and P. D. L. Machado," Generating test cases for real-time systems based on symbolic models", *IEEE Transactions on Software Engineering*, vol. 39, no. 9: 1216-1229, 2013.

[2] S. Baharom, and Z. Shukur. "An experimental assessment of module documentation-based testing", *Information and Software Technology*, vol. 53, no. 7, pp. 747-760, 2011.

[3] C. Nie, and H. Leung. "The minimal failure-causing schema of combinatorial testing", *ACM Transactions on Software Engineering and Methodology*, vol. 20, no. 4: 1-38, 2011.

[4] M. Alshraideh, L. Bottaci, and A. B. Mahafzah, "Using program data-state scarcity to guide automatic test data generation", *Software Quality Control.*, vol. 18, no. 1: 109-144, 2010.

[5] R. Bagnara, M. Carlier, R. Gori, and A. Gotlieb, Symbolic path-oriented test data generation for foating-point programs", *IEEE Sixth International Conference on Software Testing,* pp. 1-10, 2013.

[6] N. K. Gupta, and M. K. Rohil, "Improving GA based automated test data generation technique for object oriented software", *3rd IEEE International Advance Computing Conference(IACC)*, 249-253, 2013.

[7] L. Xu, B. Xu, And J. Jiang, "Testing web applications focusing on their specialties", *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 1, 2005.

[8] R. M. Hierons, K. Bogdanov, and J. P. Bowen, "Using formal specifications to support testing", *ACM Computing Surveys,* vol. 41, no. 2, pp.1-76, 2009.

[9] P.Y. Zhan, P. John, and A. Clark," A search-based framework for automatic testing of MATLAB/Simulink models", *Journal of Systems and Software*, vol. 82, no, 2, pp. 262-285, 2008.

[10] P.A. Gotlieb, P.T. Denmat, and P.B.Botella, "Goal-oriented test data generation for pointer programs", *Infor-mation and Software Technology*, vol. 49, no. 9, pp. 1030-1044, 2007.

[11] A. M. R. Vincenzi, M. E. Delamaro, J. C. Maldonado, and W. E. Wong, "Establishing structural testing criteria for Java bytecode", *Software-Practice & Experience*, vol. 36, no. 4, pp. 1513-1541, 2006.

[12] S. R. Shahamiri, W. M. N. W. Kadir, S. Ibrahim, and S. Z. M. Hashim, "An automated framework for software test oracle", *Information and Software Technology*, vol. 53, no. 7, pp. 774-788, 2011.

[13] F. Saglietti, N.Oster, and F. Pinte, "White and grey-box verification and validation approaches for safety- and security-critical software systems", *Information Security Technology Report*, vol. 13, no. 1, pp. 10-16, 2008.

[14] Bluemke, and A. Rembiszewski, "Dataflow approach to testing java programs", *Proceedings of the 4th International Conference on Dependability of Computer Systems*. Brunow, Poland. pp. 69-76, 2009.

[15] A. Cavarra, "A data-flow approach to test multi-agent ASMs formal aspects of computing", *Information and Software Technology*, vol. 13, no.1, pp. 21-41, 2011.

[16] P. H. Liu, Data flow analysis and testing of JSP-based web applications", *Information and Software Technology*, 2006, vol. 48, no. 12: pp. 1137-1147.

[17] Z. Zhonglin, J. Limin, and M. Lingxia, "Research of searching algorithm for path test data generation", *Proceedings of 4th International Conference on Computer Science & Education*, 25-28 July 2009.

[18] Edvardsson," A survey on automatic test data generation," *Proceedings of the 2nd Conference on Computer Science and Engineering. Linkoping University*, Oct. 1999:

[19] T. J. McCabe, "Structured testing: a testing methodology using the cyclomatic complexity metric", *Baltimore: McCabe and Associates,* 1987.

[20] W. Ganchang, "*Linear Algebra*", Baijing: China Renmin University Press, 2009.

[21] D. Zhenguo, and G. Qiang, "Research on the Program Control Based Path Coverage Testing Technique", *Electronica Science and Technology*, vol. 11, no. 22, pp. 3-56, 2008.

Received: September 22, 2014 Revised: November 04, 2014 Accepted: November 06, 2014