Generation Model of Basis-Paths Based on Variable-Dependence and Interval-Arithmetic

Du Qingfeng, Lu Qiaoying
School of Software Engineering
Tongji University
Shanghai, China
du_cloud@tongji.edu.cn, qiaoyinglu@gmail.com

***Abstract*—Basis-path coverage testing is a kind of logic-coverage testing and is based on graph theory and vector theory. Because the combination between path and business logic hasn't been considered, non-executable paths could be generated after executing the Basis-path algorithm. This generation model defines the non-executable path-segment and presents sub-model for acquisition of potential non-executable path-segments based on variable-dependence as well as sub-model for determination of non-executable path-segment based on interval-arithmetic. The combination of these two sub-models finds out all the non-executable path-segments and ensures that the basis-paths generated by the McCabe's generation model are all executable. Through experimental verification, it is proved that this generation model could effectively generate the executable basis-path suite.**

Keywords—Predicate node; Variable-dependence; Interval-arithmetic; Generation model of Basis-paths

I. INTRODUCTION

Basis-path coverage testing is a kind of logic-coverage testing, based on graph theory and vector theory. Each path in the control flow graph is seen as a vector in the vector space. Find a set of mutually independent basis-vectors which can linearly represent all other vectors. That means, find a set of mutually independent basis-paths which can linearly represent all other paths in CFG. Finally, design corresponding test cases based on the basis-paths to achieve basis-path coverage.

At present, the McCabe's generation model of basis-paths [1][2] is a commonly used method to acquire basis-path suite. This model has not associated a path in CFG with the practical business logic, so the acquired basis-path suite contains in most cases at least one non-executable path which does not match the practical business logic. In other words, the corresponding business stream is unlikely to be executed, and associated test case cannot be found. Therefore, this model can only be used as a supplementary coverage testing method.

Currently, many researchers in the field of software engineering proposed various solutions for non-executable path issue. Takeshi et al. brought forward the define-use chains [3]. Goldberg et al. built logical reasoning formula based on symbolic analysis to determine the satisfiability of path condition [4]. Bodik et al. proposed the generation algorithm of infeasible paths based on branch correlation [5]. Bueno et al. advanced the dynamic identification of infeasible paths [6]. Wan Lin et al. put forward feasible path selection method based on the minimal predicate [7]. Yates D.F et al. proposed JJ-paths model to generate valid basis-paths [8]. However all the above methods could only cover parts of the non-executable paths or have a relatively poor performance due to the complexity of the algorithm logic, and so far the non-executable paths issue hasn't been effectively solved.

To effectively solve that issue, this paper proposes a generation model of basis-paths based on variable-dependence and interval arithmetic. This model presents the sub-model for acquisition of potential non-executable path-segments and the sub-model for determination of non-executable path-segment, which identify all the non-executable path-segments, and generates the basis-paths according to the McCabe's generation model [1][2] and ensures the executability of the generated basis-paths by excluding the non-executable path-segments during generation.

II. GENERATION MODEL OF BASIS-PATHS

The two sub-models introduced in this section are applied to CFG, so we should first convert the component under test to CFG [1]. In addition, each node should be identified as ordinary node, predicate node of condition, predicate node of loop, tail node of loop body or true/false branch node.

A. Sub-Model for Acquisition of Potential Non-Executable Path-Segment Based on Variable-Dependence

1) Related definition

a) Affecting node and Self-affecting node:

There is a path-segment P_{mn} from node N_m to node N_n, N_m assigns a value to variable V_n and N_n uses it. Node N_n is variable-dependent on node N_m about variable V_n if V_n hasn't been reassigned a value to by any other node on P_{mn}. Here node N_m is called **affecting node** of node N_n about variable V_n.

Each node may contains several statements, node N_n is variable-dependent on itself about variable V_n if the i-th statement of N_n uses V_n to which the (i-k)-th statement assigns a value and V_n hasn't been reassigned a value to by any other statement between them. Here node N_n is called **self-affecting node** about variable V_n.

b) Affecting path-segment and Potential non-executable path-segment:

Let us upward traverse CFG from node N_n to search for its affecting-node N_m about variable V_n used by N_n. If N_m assigns a value to V_n by using variable(s) (including V_n itself) and variable V_m is one of those used variables, we continue to upward traverse CFG to find the affecting-node N_l of N_m about V_m. Likewise, we upward traverse CFG until node N_x assigns a value to variable V_x without using any variable.

Path-segment PS_{xn} is a path-segment from node N_x to node N_n through a series of affecting-nodes. Here PS_{xn} is called **affecting path-segment** of node N_n, and node N_x is the start node of affecting path-segment.

If node N_n is a predicate node, path-segment PS_{xn} can also be called **potential non-executable path-segment** of node N_n (**PNPS** for short), and node N_x is the start node of PNPS.

2) Logic of path-segment acquisition sub-model: This sub-model is used to acquire all the PNPSs of CFG based on the above definitions. The specific logic is as follows:

a) Initialize for each node the table of the assigned variables and that of the used variables.

b) For each predicate node, search for all its PNPSs about each of its used variable:

Suppose that node N_n is a predicate node, variable V_n is one of its used variable. Let us upward depth-traverse CFG from N_n to search for its affecting node N_m about V_n. The edge between N_m and N_n is a PNPS if node N_m assigns a value to variable V_n without using any variable. Otherwise, we recursively search for the affecting node of N_m about each of its variable until we come across the start node of a PNPS.

If we come across an unvisited tail node of loop body during traversing, we perform the above operation after marking it as visited. If we come across a visited tail node of loop body, we ignore it and continue to traverse CFG.

Let us perform the above method for each predicate node and each of its used variable to acquire all the PNPSs.

B. Sub-model for determination of non-executable path-segment based on interval-arithmetic

1) Related definition

a) Path-segment prefix and Full path-segment:

Node N_s is the start node of path-segment PS_{sn}, so the path-segment from the source node of CFG to the parent node of N_s is called the **path-segment prefix** of PS_{sn}.

The path-segment composed of the path-segment prefix of PS_{sn} and PS_{sn} itself is called the **full path-segment** of PS_{sn}.

b) Non-executable path and Non-executable path-segment:

Path P_i is a path which starts from the source node of CFG and ends with its sink node, if P_i is not the practical execution path of any test case, P_i is called **non-executable path**.

Path-segment PS_j is a segment of P_i, if each path contains PS_j is a non-executable path, PS_j is called **non-executable path-segment** (**NPS** for short).

2) Logic of path-segment determination sub-model: This sub-model is used to determine if the PNPS(s) acquired by path-segment acquisition sub-model is/are non-executable based on reference [9][10] and the above definition. The specific logic is as follows:

a) Breadth-first traverse CFG to initialize the variable table and the regular constraint table of each node [10].

Let us analyse each statement contained in each node:

For each variable declaration statement without assignment, we fill the declared variable(s) into the variable table of current node, and set its value range as the default one of its data type [9].

For each assignment statement, we analyse the arithmetic/relationship/logical expression right side of the assignment operator in accordance with operators' precedence and associativity, fill hierarchically from the innermost simple expression all the operands into the variable table of current node, includes constants, variables and the temporary variable which represents the value of the whole simple expression, and set the value ranges of variables and temporary variables as the default ones of their data type [9].

Then we fill out the regular constraint table in the following form: temp=x op y(or temp=op y or temp=x op), here op indicates operator, temp indicates value of the right expression, which can be temporary variable, and x/y indicates operand, which can be constant, variable or temporary variable. temp/x/y is represented by its index in the variable table, and the format is node index::value index.

After the analysis of the whole right expression, temporary variable $temp_{exp}$ is generated which represents its value. Then in the variable table, we replace $temp_{exp}$ with the variable left side of the assignment operator.

For each predicate statement, all its expressions are hierarchically analysed using the above analysis method, and temporary variable $temp_{exp}$ of boolean type is generated which represents the value of the whole expression. Then we fill it into the variable table, and set its value range as the default one of boolean type [9].

b) Construct full path-segment and generate its variable table and regular constraint table:

For each PNPS, we generate all its full path-segments and add them into its full path-segment set:

For each full path-segment of current PNPS, we successively start with the source node of CFG and link the variable table and the regular constraint table of each node to generate those two tables of current full path-segment.

During linking, we set the value of the corresponding variable in the variable table to true/false according to the true/false branch when we encounter a predicate node.

Each PNPS acquired by path-segment acquisition sub-model enters at most once each loop body. In practice, one path

may enter the same loop body for several times. In order to cover such cases, when we encounter the true branch node of a predicate node of loop during linking and the loop time of the corresponding loop body in current full path-segment is not greater than the default value maxLoop, we generate all the path-segments from that predicate node to the tail node of the corresponding loop body (loop-segment for short). If we encounter nested loop in this process, we just consider the following two cases: enter only once the loop body and do not enter it. For each loop-segment, a new full path-segment is generated by inserting it before that predicate node and added into the full path-segment set of current PNPS.

After linking, we traverse the variable table of current full path-segment, remove each variable which occurs not for the first time, and replace the corresponding index in the regular constraint table with the earliest one.

c) Generate use case suite [10]:

After the tables of current full path-segment have been generated, we traverse the regular constraint table and divide each operand of multiplication or division into two value ranges: one is less than zero, and the other is greater than or equal to zero, for divider, the latter one excludes zero.

After traversing, each permutation of all the operands according to their value ranges corresponds to a use case, and then a variable table is generated for each case.

d) Perform interval reduction algorithm on each use case [10]:

First successively queue all the regular constraints into the active queue, and the idle one is now empty.

Then dequeue the first constraint and reduce the value ranges of all the variables in this constraint one by one according to the rules of interval arithmetic [9]. If the value range of any variable changes and isn't empty, update current variable table and move all the regular constraints associated with this variable in the idle queue to the active queue so as to perform interval reduction further on other variables. If the value range changes and is empty, return false. Finally, queue current constraint into the idle queue.

If the active queue is empty, return true.

e) Determine the executability of current PNPS according to the reduction result:

Current PNPS is executable if return true after performing interval reduction algorithm on any use case.

Current full path-segment is non-executable if all return false after performing interval reduction algorithm on each of its use case. Current PNPS is non-executable if its full path-segments are all non-executable, so add it into the NPS set.

When all PNPSs have performed the above operations, current NPS set contains all the NPSs of CFG.

III. EXPERIMENTAL VERIFICATION

A. Experimental Subject

The program under test is passRate() function, which is used to calculate the exam pass rate. Its pseudo code and CFG are shown in Figure 1 and Figure 2.

```
public float passRate(int score[]){
int i=0;
int validCount=0;
int passCount=0;
float passRate;                                   1
while(i<score.length){                            2
 if(score[i]<=100&&score[i]>=0){                  3 4
  validCount++;                                   5
   if(score[i]>=60){                              6
    passCount++;}}                                7
 i++;}                                            8
if(validCount>0){                                 9
 passRate=passCount/validCount;                   10
}else{
 passRate=-1;}                                    11
return passRate;}                                 12
```

Fig. 1. Code of passRate()

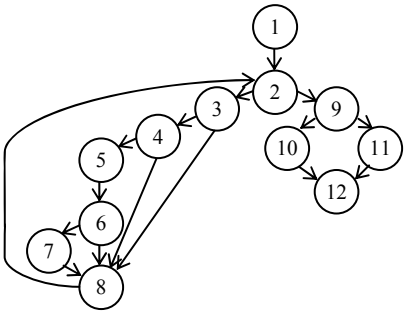

Fig. 2. CFG of passRate()

B. Experimental Result

After execution of the basis-paths generation tool based on algorithm shown in Appendix A, its basis-path suite has been generated. The specific basis-paths and the corresponding test cases are listed as follows, and the paths are illustrated with bold arrows in CFG from Figure 3 to Figure 8:

Basis-path 1: 1→2→3→4→5→6→7→8→2→9→10→12

Test Case 1: There exists at least one valid score, and the pass rate is greater than 0.0.

Basis-path 2: 1→2→9→11→12

Test Case 2: There exists no score.

Basis-path 3: 1→2→…→2→3→8→2→9→10→12

Test Case 3: There exists at least one valid score and one invalid score greater than 100.

Basis-path 4: 1→2→…→2→3→4→8→2→9→10→12

Test Case 4: There exists at least one valid score and one invalid score less than 0.

Basis-path 5: 1→2→3→4→5→6→8→2→9→10→12

Test Case 5: There exists at least one valid score, and the pass rate equals to 0.0.

Basis-path 6: 1→2→3→8→2→9→11→12

Test Case 6: There exists no valid score but at least one invalid score greater than 100.

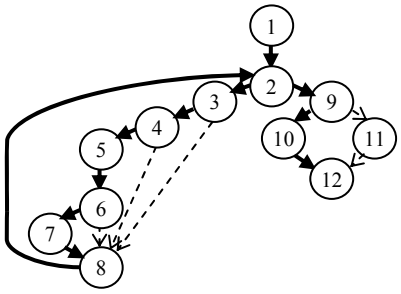

Fig. 3. Basis-path 1

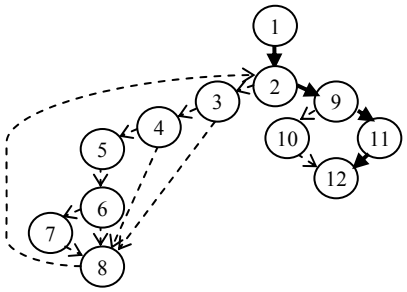

Fig. 4. Basis-path 2

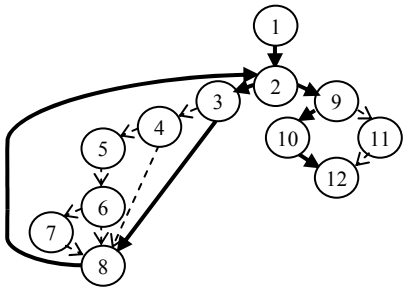

Fig. 5. Basis-path 3

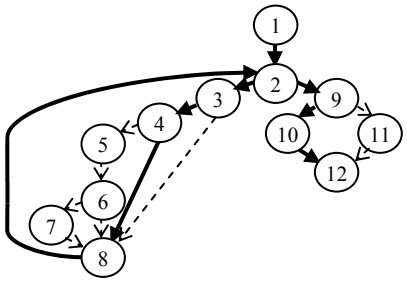

Fig. 6. Basis-path 4

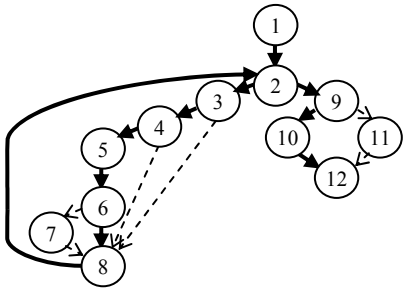

Fig. 7. Basis-path 5

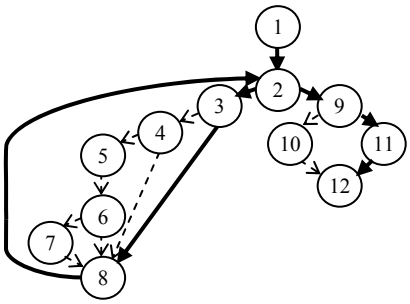

Fig. 8. Basis-path 6

All these six paths are basis-paths because they are generated according to the McCabe's generation model of basis-paths, and the count of basis-paths meets the count calculated by the McCabe's cyclomatic complexity formula.

IV. Conclusions

This paper bases the basis-path-coverage testing on the combination between path and business logic. First acquire all the PNPSs so as to reduce the input set of the interval arithmetic method. Then figure out the NPSs through the interval arithmetic method. Finally, exclude the paths contain any NPS during the generation of basis-paths. Thus, ensure the executability of all the generated basis-paths the most degree.

In conclusion, the generation model of basis-paths based on variable-dependence and interval-arithmetic proposed in this paper effectively solve the path-executability issue exists in basis-paths coverage testing, and break the impasse that it can only be regarded as a supplementary testing method.

Appendix A
Algorithm Implementation

Figure 9 shows the data structure of CFG which consists of an array nodeArray and an adjacency matrix nodeMatrix. Algorithms shown in Figure 10 to Figure 12 implement the whole basis-paths generation model and its sub-models:

```
1  bool[][] nodeMatrix;//stores arc information
2  Node[] nodeArray;//stores nodes data
3  class Node{
4     String nodeType;
5     List<AssignVariable> assignVarList;
6     List<String> useVarList;
7     bool isVisited;
8     List<Variable> varList;
9     List<Reg> regList;}
10 class AssignVariable{
11    String variable;
12    List<String> useVarList;}
13 class Variable{
14    String varID, varType;
15    List<Range> varRange;}
16 class Range{
17    String min, max;
18    double stepLength;}
19 class Reg{
20    VarIndex leftOperand, rightOperand;
21    String operator;}
22 class VarIndex{
23    int nodeIndex, varIndex;}
```

Fig. 9. Data structure of CFG

```
Procedure getPNPSList
  Input nodeArray,nodeMatrix
  Output PNPSList
Begin
  initiate the assignVarList and useVarList of each node;
  for each predicate-node i in nodeArray{
    nodeStack.push(i);
    for each variable v in nodeArray[i].useVarList{
      /* analyze parent-nodes of nodeArray[i]*/
      for each parent-node p of nodeArray[i]{
        nodeStack.push(p);
        /*ignore the visited tail-node of loop-body*/
        if (!nodeArray[p].isTailNodeOfLoopBody||!nodeArray[p].isVisited){
          nodeArray[p].isVisited=true;
          /*nodeArray[p] is the affect-node of nodeArray[i]*/
          if nodeArray[p].assignVarList contains v{
            /*nodeArray[p] is the start-node of PNPS*/
            if the assignment of v only use constant
              {add the path in nodeStack to PNPSList;}
              /*upwards analyze the inter-node of nodeArray[p]*/
            else{
              for each variable w in useVarList of v
                {recursively analyze parent-nodes of nodeArray[p];}}}
          else{recursively analyze parent-nodes of nodeArray[p];}}
        /*pop the node which is already analyzed*/
        p=nodeStack.pop();
        nodeArray[p].isVisited=false;}}}
  return PNPSList;
End
```

Fig. 10. Algorithm of path-segment acquisition sub-model

```
Procedure genNPSList
  Input nodeArray,nodeMatrix,PNPSList
  Output NPSList
Begin
  /*initiate the varList and RegList of each node*/
  traverse CFG
    for each statement of nodeArray[i]{
      if assignment-statement{
        analyze the right expression and fill the varList and RegList;
        replace temp_exp by the left variable;}
      else if declaration-statement{
        add variable to varList;}
      else if predicate-statement{
        analyze the expression and fill the varList and RegList;
        add temp_exp to varList;}}
  End traverse
  judge:for each PNPS{
    /*determine if PNPS is non-executable*/
    generate fullPathSegList;
    for each full-path-segment{
      /*generate varList and RegList of this full-path-segment */
      traverse the path-segment
        if nodeArray[i] is predicate-node{
          set the last variable of varList to true/false;}
        append varList and RegList of nodeArray[i];
        if nodeArray[i] is true-branch-node of predicate-node of Loop{
          generate full-path-segments and add to fullPathSegList;}
      End traverse
      traverse its varList where var repeatedly turns up
        delete this element;
        update the corresponding index in its RegList;
      End traverse
      generate the caseList; //read reference[10]
      for each case{
        /*PNPS is executable*/
        if range-narrowing returns true{//read reference[10]
          /*go to determine the next PNPS*/
          continue judge;}}}
    /*PNPS is non-executable*/
    add this PNPS to NPSList;}
  return NPSList;
End
```

Fig. 11. Algorithm of path-segment determination sub-model

```
Procedure genBPList
  Input nodeArray,nodeMatrix
  Output BPList
Begin
  calculate the number of the independent paths V_G;
  List PNPSList=getPNPSList(nodeArray,nodeMatrix);
  /*standardize the PNPSList*/
  update the PNPSList by adding true/false-branch-node to each PNPS;
  delete PNPS which has duplicated edges;//except the compound predicate
  List NPSList=genNPSList(nodeArray,nodeMatrix,PNPSList);
  traverse CFG to get the first base-path which contains no NPS;
  add it to BPList;
  /*generate other base path */
  take first base-path as baselinePath;
  While((BPList.length<V_G)||(!baselinePath)){
    traverse baselinePath where(out-degree[i]>=2)&&(edge[i,j] is unvisited
    in current path)
      take nodeArray[j] as child-node of nodeArray[i] and continue
      traversing CFG to generate new base-path which contains no NPS;
      add it to BPList;
    End traverse
    take the next base-path as baselinePath;}
  return BPList;
End
```

Fig. 12. Algorithm of basis-paths suite generation model

ACKNOWLEDGMENT

This paper is supported by the National Natural Science Foundation of China under Grant No.41171303.

REFERENCES

[1] Du Qingfeng. “Advanced Software Testing Technology” [M]. Beijing: Tsinghua University Press, 2011:104-112.

[2] Du Qingfeng, Dong Xiao. “An Improved Algorithm for Basis Path Testing”[C], 2011 International Conference on Business Management and Electronic Information, 2011:175-178.

[3] Takeshi, Nakajo. “A test-path determination method based on define-use chains: Test conditions and program fault overlooks”[J], Systems and Computers in Japan, Vol.24, No.5, 1993:14-29

[4] Goldberg, A. Wang, T.C, Zimmerman.D. “Applications of feasible path analysis to program testing”[C], Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA), 1994:80-94.

[5] Bodik R, Gupta R, Soffa M L. “Refining Data Flow Information Using Infeasible Paths Using Infeasible Paths”[C], ACM SIGSOFT Software Engineering Notes, 1997:361-377.

[6] Bueno, P.M.S, Jino.M. “Identification of potentially infeasible program paths by monitoring the search for test data”[C], Proceedings ASE 2000, 15th IEEE International Conference on Automated Software Engineering, 2000:209-218.

[7] Wan Lin, Xiao Qing, Gong Yunzhan. “Feasible Path Selection in Structure Test” [J], Computer Engineering, 2003, Vol.29, No.2: 42-46.

[8] Yates D.F, Malevris N. “Inclusion subsumption, JJ-paths and structured path testing: a Redress”[J], Software Testing, Verification and Reliability, Vol.19, No.3, 2009:199-213

[9] Wang Yawen, Gong Yunzhan, Xiao Qing, Yang Zhaohong. “Variable Range Analysis Based on Interval Computation” [J], Journal of Beijing University of Posts and Telecommunications, 2009, Vol.32, No.3: 38-41.

[10] Wang Zhiyan Liu Chunnian. “The Application of Interval Computation in Software Testing” [J], Journal of Software, 1998, Vol.9, No.6: 438-443.