A System on Visualization of Program Executive Path and Extraction of Path Sets

Cai Zhimin, Rong Guoping, Zhou Peng and Pan Jingui

State Key Laboratory for Novel Software Technology, Computer Science and Technology Dept., Nanjing University, China

This article goes into the analysis of program executive path at length, performs a system of program executive path visualization, compares several aspects of different path coverage criteria and discusses the implementation of two specific, yet important, path combinations. In the end, it discusses briefly about the significance based on this system and further study.

Keywords: program executive path, route-covering combination, divergence-covering combination, directional graph, independent statement block, program semantics

1. Introduction

Program executive path is an important aspect of describing the syntax and semantics of a program. It not only reflects a program's static semantics, its whole status space and problem solving space determined by the set of program executive paths, but also reflects a program's dynamic semantics, its transforming flow of status space through the executing flow. Obviously, program executive path also has an essential role in the proof of program validation and program equivalence. At the same time, it is the basis of software testing, especially white box testing.

Therefore, analysis of program executive path is of special importance. Unfortunately, analyzing all the paths in a program is often impossible, since programs with loops may contain an infinite number of paths. Therefore, main energy should be spent on how to extract the best, if possible, subset of program paths, of which the features will be studied, as the substitute of whole path set. So that, how to select a considerate subset from whole path set, or in other words, how to decide a considerate **path selection criteria**, to achieve a high coverage and low cost, becomes a key point of the question.

Much work has been done in this field, and a number of criteria have been proposed during the years [5-8], including statement coverage, branch coverage, loop coverage and so on. And, they were compared in [3]. Also, Sipser pointed out that most of exhausting search belongs to NP problem in [12]. Specially, Chu raised several visual model of program executive path [10].

This paper is trying to introduce a system based on program directional graph (called directional graph or graph in the following, if no conflicts). In this system, program executive path can be visualized in a clear way, and so path subsets, featuring different properties and fulfilling different demands, can be generated easily. The advantage of this system includes: its clear visualization of program executive path, robustness of its algorithm and high independency of platform. Anyway, the implementing system is working on k&r C language.

This paper also puts forwards a useful path selection criterion, and to make clear the advantage of our criterion, several criteria are compared briefly in Section 3. The conclusion and further research can be found in Section 5.

2. Construction of Directional Graph

Obviously, construction of directional graph is the basis of the system. Thus, we will go into it a little further. A directional graph is the visualization of a program. So, it must remain consistent in syntax and semantics of a program, or in clearer word, render original program's syntax and semantics without any distortion. For this system, the consistency can be described as following. If constructing process is seen as a mapping:

$$f : \text{program} \rightarrow \text{directional graph}$$

f must be reversible, that is, we can construct an equivalent program from the directional graph, although program literal may be a little different due to the implementing method. This is the fundamental rule of constructing directional graph and the premises of the algorithm's validity. Secondly, the graph itself should be intuitively easy to be understood, able to reveal the program meanings hidden in program text. This will promote the algorithm efficiency.

First, let's investigate a little energy in the observation of relations between neighboring statements. Wong divided a program into the sequence of basic blocks in [1]; Weyuker imported the term of disjoint block in [4]. Similarly, we first introduce the term of *Independent Statement Block*, ISB for short, and then try to explain how to construct a directional graph from a program, based on ISBs.

Independent statement block means that:

1. a statement or declaration or condition is a minor independent block; and
2. if the executing order of two minor independent blocks is of strict consequence, these two minor independent blocks can be combined into one minor independent block, where strict consequence means that in a program executive path, for neighboring minor independent blocks A and B, the status of A appearing before B and that of B appearing before A cannot be found at the same time; and,
3. if none of such minor independent blocks can be combined, each of them is called an *independent statement block*.

According to the above, we can easily divide a program into the sequence of ISBs. Let's go a little further into the executing order among ISBs. We may find that there are altogether four types of executing order, respectively sequential, selective, iterative and transferring order, corresponding to sequential structure, selective structure (including *if structure* and *case structure*), iterative structure (including *while, do while* and *for structure*) and transferring structure (including *goto, break* and *continue structure*).

The above structures' representation in directional graph is shown separately in Figure 1.

Now, let's move to the description of directional graph. To simplify the discussion, we here assume that a program is either a main program or a single module and has only one entry and one exit point. The directional graph of a program P is $G(P) = (N, E, n_s, n_f)$, where N is a finite set of nodes, $E \subseteq N \times N$, is the set of directional edges, $n_s \in N$ is called the *start node*, and $n_f \in N$ is called the *end node*. Each node in the graph, except the start node and the end node, represents an ISB in P; for distinct nodes m and $n \in N$, $(m, n) \in E$ called an edge, is legal if there is a possible executing transfer from the ISB represented by m, to the ISB represented by n, also we call node m a *predecessor* of n, and n

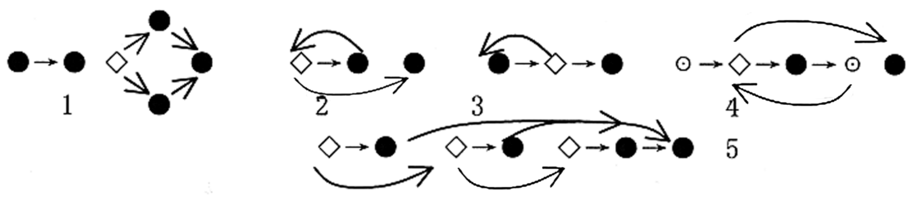

Fig. 1. The representation of different structures in a directional graph; respectively is sequential structure if structure, while structure, do while structure, for structure and case structure. Transferring structure is neglected here since its representation is similar to sequential structure. In the figure, • represents a statement, ◇, a condition, ⊙, a iterator, and arrow point represents control flow.

is a *successor* of *m*. If node has no successors, it is called a *dead node*. Dead nodes often mean an abnormal or bad program. We assume that there is no $(n, n) \in E$. Specially, there exist edges (n_s, i) and (o, n_f), where node i and o called entry node and exit node, separately.

The graph defines the paths in a program. In $G(P)$, a subpath is a sequence of nodes $p = (n_1, n_2, n_3, \ldots, n_p)$ such that for all i, $1 \leq i \leq p$, $(n_i, n_{i+1}) \in E$. A *program executive path* or a *path* is a subpath whose first node is *start node* and last node is *end node*. A *circle* or a *loop* is a subpath whose first node is same as last node, representing to an iterative or transferring structure in the program.

As mentioned above, graph must represent the original program consistently and efficiently. So, in the implementing system, we assume: compound conditions not decomposed, declarations and statements not distinguished, function call seen as normal statement or expression, function definitions analyzed respectively in a file if it contains more than one function definitions, and included head files not parsed. Also, some blank nodes may be included during construction; yet, they have no impact on the control flow of a program. Two properties of a directional graph are stated below:

1. each node, except the dead node, in the graph normally has 1 or 2 successor nodes, each node has at least one predecessor node, but no maximal value; and
2. iterative and transferring structures cause the loops in the graph, which often mean backtracking in the search. Therefore the solving of loops is of great importance in the algorithm.

```
step0  initialize including adding a START node;
step1  split the program of current layer through
       statement/declaration/condition;
step2  identify the type of current statement;
step3  IF it can be combined in a single node, combine
       it, goto step4; ELSE IF it is a simple statement,
       generate it into a sub graph, goto step4; ELSE IF it is a
       compound structure, enter inner layer, goto step1;
step4  IF current layer ends, check whether there is outer
       layer to it; IF true, goto outer layer, goto step2; ELSE
       goto step5;
step5  add END node, END.
```

Fig. 2. The constructing algorithm of directional graph.

```
1   int total, valid nim, max;
2   int i, sum, average;
3   int value[100];
4   total=0; valid=0; i=1; sum=0; min=MIN; max=MAX;
5   while (value[i]<>−999 and total < 100){
6   total++;
7   if ( (value[i]>=min)&&(value[i]) ){
8   total++;
9   sum=sum+value[i];
    }
10  else continue;
11  i++;
    }
12  if(total>0)
13  average=sum/total;
14  else average = −999;
```

Mapping: 1-4:A, 5:B, 6:C, 7:D, 8-9:E, 10:F, 11:G, 12: H, 13:I, 14:J.

Fig. 3. An example to demonstrate the algorithm.

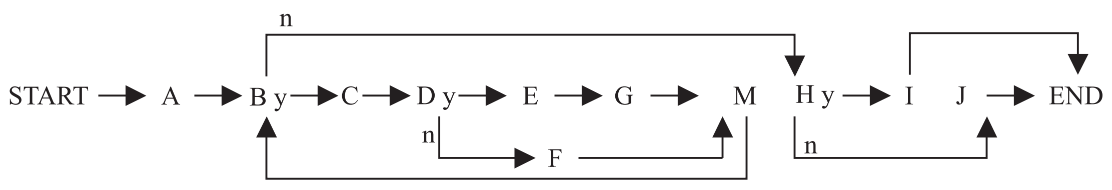

Fig. 4. The directional graph derived from the above program fragment, in which M is a blank node.

Before we move to discussing the details of the algorithm, one thing must be pointed out. One must confess that *goto* statement is not always a trouble causer in software, especially for programming of embedded systems. Sometimes, it is an efficient tool in programming. So, we deal with simple and easily understood *goto* statements, otherwise, we will give a warning message that it is difficult to construct such a directional graph.

Now, it is time to focus on the algorithm itself. The basic idea is: seeking from the beginning of a program, judge the relations of neighboring statements. If they can be combined, combine them, otherwise, create a new node and set its links to other nodes, constructing corresponding subgraph. Seek and add nodes repeatedly, until it reaches the end of the program. Because programs are normally nested, graph generation is not done step by step along the program, but partitioned, needing backtracking. Or, in other words, because the syntax of a program is recursive, the algorithm is recursive. In Figure 2 the algorithm is shown in detail.

The example given in Figure 3 demonstrates the algorithm, which is also frequently used in the following discussion. This program fragment is originally used to compute the average value of an array of numbers. The corresponding mapping between nodes and statements is shown in the figure. The graph derived from Figure 3 is shown in Figure 4.

3. Discussion of Three Selection Criteria

3.1. Comparison of Three Criteria

Definition: **Independent Path** is a path that contains at least one uncovered ISB, compared to the paths that have existed or have been derived. *Independent path criterion* requires that each path extracted is an independent path during extracting procedure. In nature, it is equivalent to statement coverage criteria. It needs that every statement of a program be executed at once in the test. It is well accepted that one may not feel confident about the program's behavior if there are some statements that are still not executed any time. After all, one should not be so glad because it is clear that even if all the statements are executed in the test, a program is very likely to contain errors. So, this criterion is regarded as the weakest one. It is mostly used for comparison purpose, and so is in this paper. In fact, the path set derived from this criterion can only reveal some literal errors and simple logical errors in a program; it does not reflect real transforming flow of program status space, especially the transforming procedure of program dynamic semantics, due to iterative and transferring structures. So, the defect of the independent path criterion (called criterion 1 in the following, if no conflicts) is apparently distinct. For the example in section 2, a possible path set is {(A, B, H, I), (A, B, H, J), (A, B, C, D, F, M, B, H, I), (A, B, C, D, F, M, B, H, J), (A, B, C, D, E, G, M, B, H, I), (A, B, C, D, E, G, M, B, H, J)}, whose path number is 6.

Loop coverage criterion is such a criterion that the path set must cover all the possible loops combinations at least once. In formal words, if a program has n loops, a path set P satisfies loop coverage criterion, if paths with t loops are executed at least once, where $1 \leq t \leq n$. This criterion subsumes branch coverage criterion, which requires that each branch be covered in the set (if there is no loops/iterating or transferring structure, they are identical). Under this criterion, path p (A, B, C, D, E, G, M, B, C, D, F, M, B, H, I) needs to be tested, while path q (A, B, C, D, F, M, B, C, D, E, G, M, B, H, I) can be excluded if path p has been tested. Consider path p and q, they both have loops m (D, F, M) and n (D, E, G, M); the difference is the executing order of two loops. Such that, this criterion (called criterion 2 in the following, if no conflicts) regards that they reflect similar program dynamic semantics, it is unnecessary to test them repeatedly (Strictly, the transforming of program status space is wholly different through different paths). Criterion 2 can cover all the statements and all branches, and also cover all iterative and transferring structures. It reflects more effectively the influence on program semantics, due to the variation of different number of executing loops. A significant number of errors will be revealed by such a test. Of course, the path number increases rapidly, compared with criterion 1. And backtracking is inevitable in extracting paths. For the example in section 2, a possible path number is 8.

Strong loop coverage criterion, based on loop coverage criterion, it also requires that all the possible loop permutations should be additionally executed at least once. Under this criterion (called criterion 3 in the following, if no conflict), path p and path q are different. It argues that although loop m and loop n appear in both paths, the executing order is not the same; the transforming of program status space along each path is not same. They should be regarded as paths with different program semantics, and thus should both be tested. Criterion 3 takes into account the executing order of different loop, and considers of its influencing on program dynamic semantics, then reflects more details of a program's semantics transforming along different paths. After all, one obvious disadvantage is that the path number increases prominently, compared with criterion 2, even causing combination exploding (In fact, it partly indicates a real state that analyzing all paths in a program belongs to NP problems). And, backtracking is an important part of the extracting algorithm. For the example in section 2, a possible path number is 10.

One thing must be mentioned in above discussion, a single loop is assumed to execute once or none, which seems weak to reflect the real state of program executing. After all, on one hand, such a test concerning the executing times of a loop can be found in Woodard's LCSAJ test in [9], which is beyond this paper, also we know that the variation in executing a single loop with different times is only the value of variables, which probably has less influence on program test; on the other hand, one should balance between coverage and cost, since too many **infeasible paths** will be extracted if too much attention is paid to counting the executing times [2]. So, we are only interested in whether a loop is executed or not and in the order in which certain loops are executed, while the executing times is not taken into account.

Now, let's estimate path numbers and computing complexity of three criteria. Since most of the time is spent on visiting nodes, only such time is concerned in the algorithm. Supposed that there are n nodes in a program P's directional graph, and the time of visiting each node is constant. Because at most n nodes are visited in a single path, time of visiting a single path is assumed $O(n)$.

Under criterion 1, McCabe pointed out that the path number equals to the cyclomatic complexity of a program in [7], which is, of course, less than n, assumed $O(n)$. So the whole time consumption is $O(n^*n)$. Under criterion 2, path number has nothing to do with n, but with number of loops. Assumed there are t loops in program P, the number of paths that have t loops is C_t^t, that have $t-1$ loops is $C_t^{t-1}, \ldots,$ that have one loop is C_t^1, that have no loop is C_t^0. Altogether, path number is $C_t^t + C_t^{t-1} + \ldots + C_t^1 + C_t^0 = 2^t$. The whole time consumption is $O(n^*2^t)$. Since t is usually less than 20, and is also far less than n, such a scale is compatible. Similarly, under criterion 3, path number is $P_t^t + P_t^{t-1} + \ldots + P_t^1 + P_t^0$. Obviously if t is moderately large, it is beyond a computer's ability.

In the end, summary of the comparison is shown in Table 1.

3.2. Selection of Our Implementing System

The users of this system are designers of embedded systems, who hold a high view of program correctness, thus expecting to make the most of different test paths from different angles. So, we introduce strong loop coverage criterion, whose merits can be clearly seen from the comparison. Another reason is that, several estimable path sets can be extracted, based on strong loop

Criterion	Statement coverage	Branch coverage	Loop coverage	Reflecting semantics	Path number	Backtracking in parsing	Algorithm complexity
Cri. 1	able	disable	disable	bad	Small	None	Low
Cri. 2	able	able	able	good	Large	A little	High
Cri. 3	able	able	able	better	Larger	Much	being NP

Table 1. Comparison of different path selection criteria.

coverage path set. With regard to combination exploding, the following should be done: for programs whose cyclomatic complexity is too high, for example, more than 20, a warning message will be given that it is possible to cause combination exploding; else if there are still too many potential paths, the user may specify a maximal path number or other limits to reduce the path number.

4. Implementation of the System

For comparison purpose, we implemented two path sets, based on two selection criteria. One set is *Route Covering Combination*, short for RCC, which is a path set, based on strong loop coverage criterion; the other one is *Divergence Covering Combination*, short for DCC, which is the smallest path set that covers all branches in a program. In fact, it is based on branch coverage criterion. In the system, we obtain paths of RCC first, and then select certain paths from RCC to build DRC.

4.1. Extraction of RCC

Brief of the algorithm is: from *start node*, search through the graph, using DFS (Depth First Search), storing unvisited links of nodes to **node stack**, until reaching *end node* (thus a path is obtained) or a *dead node*, pop an unvisited node from **node stack**, and go on searching until **node stack** is empty, which means RCC is complete, or path number reaches certain maximal number, which means the program has to end abnormally.

Data structures related to algorithm is explained below:

Node: containing program text corresponding to a node. Each node has two links. If it has two successors, the left link points to the successor of true branch, the right one to that of false branch. Else, if it has only one successor, the left link works, by default. Otherwise, both links are set null.

Stack: Path stack stores current searching path. **Node stack** stores unvisited nodes during the search. **Temp stack** stores temporary subpath.

Path list stores extracted paths and finally writes them to a file. **Loop list** controls loops, recognizing, recording and comparing loops in current searching path, also in charge of finding and delete duplicated circles.

Backtracking of seeking is necessary due to the requirement of searching different branches, avoiding duplicated loops and returning from *dead nodes*.

A complete extracting algorithm is shown below:

```
Initialize the surroundings of seeking; include ad|ding a
nul node to node stack; initialize the graph;
WHILE node stack not empty do
  WHILE current node is not END do
    IF right link is not empty, push it to node stack;
   temp node := left link;
   IF temp node is null //back tracking
    repeatedly pop nodes from node stack and
      modify path stack and c|heck path until there
      are no same circles in path stack; store the last
      popped node to temp node;
      current node=temp node;
      IF current node is END, BREAK;
    ELSE
      Push temp node to path stack;
      Like above, do popping, modifying and
      c|hecking and storing;
      current node=temp node;
      IF current node is END, BREAK;
  ENDIF
 ENDWHILE // find a single path;
 IF node stack is empty, BREAK;
    Like above, do popping, modifying and c|hecking
    and storing;
    print the path from path stack; path number plus 1;
    /*start another route seeking;*/
    IF path number larger than MAX, BREAK;
ENDWHILE // find all pos|sible routes
```

4.2. Extraction of DCC

There is some practical difficulty in implementing DCC strictly. It is not so easy to find such a minimal set that has least paths, because such a set is possibly multiple, and in case of combination exploding, it is impossible to find a minimal set because RCC is not even obtained correctly. Also, in practical test, minimal path number usually does not mean the best test effect. In fact, users expect from DCC to get a good test quality with relatively small path set. Then, in the system, we just try to find first satisfying path set as DCC.

Thus, the algorithm of DCC is described as following: we first count the number of the conditions in the program, supposed is n, and add a coverage array to each path to record covering results of different branches. Also, we initialize a balance array to record covering status in current DCC. Of course, the size of coverage array and balance array is both 2^*n, since a condition has a true branch and a false branch, altogether. Initially the value of item in both arrays is 0. At certain time, if certain branch of a condition is covered, the corresponding item will be set 1. In the beginning, the coverage array of each path in RCC is assigned according to its coverage of all the branches. Pick a path from RCC one by one, compare it to balance array by "eXclusive OR" operation to get its current covering quality, which is represented as covering degree, number of array items, whose XOR result is 1) and redundant degree, number of array items, whose XOR result is 0), select the path that has the highest covering degree and the lowest redundant degree and add to DCC, then modify the balance array according to current status of DCC. Repeat these operations until all the items of balance array are 1, thus DCC is generated.

The details of the algorithm are as following.

```
initialize the seeking; END := false;
WHILE not END
  WHILE path list not empty
    get a path from path list; compare with Balance
    Array to calculate the current value of this path;
  ENDWHILE
 rest the path list;
  Best Path := null;
 WHILE path list not empty
     get a path from path list;
     IF this path is better than Best Path, replace it;
 ENDWHILE
 ad|d this Best Path to teh Set and modify the Balance
   Array;
  c|heck whether Balance Array is FULL,
   if so, END := true;
  reset the value of eac|h path;
ENDWHILE
```

5. Conclusion and Further Research

5.1. Conclusion Drew from the Experiment

We implemented our system with C^{++} Builder 3.0 on PC P100/40M and tested 80 programs, trying to find the relations among path number, nest number, and cyclomatic complexity.

As regards RCC, 12 cases exceed the upper limit (10000 paths), 8 cases have paths more than 1000, 38 cases have paths more than 10, the others have less than 10. But as regards DCC, all the cases have paths less than 10. We suppose that RCC may have a good test quality in most cases, and that DCC can be used to test branches and loops in a program in quick time.

The relations among path number, nest number and cyclomatic complexity is not so simple. For programs that do not contain loops, path number increases with nest number and cyclomatic complexity. But for programs that contain loops, the relationship is not an increasing function. Path number is more related to the nature of loops. This will be the subject of our further research.

5.2. Further Research

As is known from the above study, RCC is based on loop coverage, with an aim to study the transforming procedure of program dynamic semantics through checking whether a certain loop is executed or not and through relative executing order of loops in an executive path. DCC is based on branch coverage, with an aim to study the transforming procedure of program dynamic semantic through checking whether a certain branch is executed or not and relative executing order of branches in an executive path.

This system can help us do some research on special-aimed path extracting techniques. Now we are considering the techniques including following aspects: path set in which specified statements or branches are covered, and path set in which definitions or references of specified variables are covered. In addition, we are also focusing on the following areas.

Analyzing the characteristics of program syntax and semantics embodied in a program executive path, which is one of our primary research interests.

Studying relations between program complexity and path number, to find the delicate determinism of a program's complexity on the path set.

And, a useful research is to get practicing data of relations between path number and latent bugs number of programs. PG-Relief system, developed in cooperation with Fujitsu Company, can find latent bugs in a program. Based on these data, we hope to find statistical relations between them through testing large numbers of programs, thus using it as a useful guide in programming.

Also, we try to test programs from different angles, in different test environments, from which we can study the validation and maturity of program test, hence we will have a deeper understanding of program correctness.

References

[1] W. ERIC WONG, JOSEPH R. HORGAN, SAUL LONDON, AND ADITYA P. MATHUR, "Effect of Test Minimization on Fault Detection Effectiveness" , *Proc.* 17th *International Conf. On Software Engineering*, April 23–30, 1995, Seattle, Washington, USA

[2] D. HEDLEY AND M. A. HENNELL, "The Cause and Effects of Infeasible Paths in Computer Programs", *Proc.* 8th *International Conf. On Software Eng.*, Aug 28–30, 1985, London, UK

[3] LORI A. CLARKE, ANDY PODGURSKI, DEBRA J. RICHARDSON, AND STEVEN J. ZEIL, "A Formal Evaluation of Data Flow Path Selection Criteria", *IEEE Trans. On Software, Eng.*, Vol. 15, No. 11, Nov. 1988

[4] ELAINE J. WEYUKER, "Evaluating Software Complexity Measures", *IEEE Trans. On Software Eng.*, Vol. 14, No. 9 Sept. 1988

[5] SANDRA RAPPS, ELAINE J. WEYUKER, "Selecting Software Test Data Using Data Flow Information", *IEEE Trans. On Software Eng.*, Vol. Se–11, No. 4, April, 1985

[6] PHYLLIS G. TRANKL AND ELAINE J. WEYUKER, "An Applicable Family of Data Flow Testing Criteria", *IEEE Trans. On Software Eng*, Vol. 14, No. 10, Oct. 1988

[7] THOMAS J. MCCABE, "A Complexity Measure", *IEEE Trans. On Software Eng.*, Vol. Se–2, No. 4, Dec. 1976

[8] S. C. NTAFOS, "On Required Element Testing", *IEEE Trans. On Software Eng.*, Vol. Se–10, No. 6, Nov. 1984

[9] M. R. WOODARD, DAVID HEDLEY, AND MICHAEL A. HENNELL, "Experience with Path Analysis and Testing of Programs", *IEEE Trans. On Software Eng.*, Vol. Se–6, No. 3, May, 1980

[10] CHU JUNJIE, "Developing higher level views of execution paths", *Chinese J. Computers*, Vol. 21, No. 3, Mar. 1998

[11] ROGER S. PRESSMAN, Software Engineering, *A practitioner's Approach*, Fourth Edition 1997, McGraw–Hill

[12] MICHAEL SIPSER, *Introduction to the Theory of Computation*, 1997, PWS

[13] KENNETH C. LOUDEN, Compiler Construction, *Principles and Practice*, 1997, PWS

[14] ZHENG RENJIE, *Computer software test techniques*, 1992, Tsinghua University Press

[15] XU JIAFU, Corpus of Xu jiafu, 228–248, 1992, Nanjing University Press

Received: June, 2001
Accepted: September, 2001

Contact address:

Cai Zhimin
State Key Laboratory for Novel Software Technology
Computer Science and Technology Dept.
Nanjing University, 210093, China
e-mail: `czhimin@mes.nju.edu.cn`
`zhimin_cai@hotmail.com`

Rong Guoping
State Key Laboratory for Novel Software
Technology, Computer Science and Technology Dept.
Nanjing University, 210093, China
e-mail: `jacky@mes.nju.edu.cn`

Zhou Peng
State Key Laboratory for Novel Software Technology
Computer Science and Technology Dept.
Nanjing University, 210093, China
e-mail: `duckbill@dislab.nju.edu.cn`

Pan Jingui
State Key Laboratory for Novel Software Technology
Computer Science and Technology Dept.
Nanjing University, 210093, China
Phone: 86-25-3260023
Fax: 86-25-3317685
e-mail: `panjg@nju.edu.cn`
`panjg@nanjing-fnst.com`

CAI ZHIMIN born in 1977, obtained his B.Sc. degree from Nanjing University, China, in 1999, and now is a graduate student of the graduate school, Nanjing University. He is doing research in the field of software testing and is also involved in a joint research project of Nanjing University and Fujitsu Company. He focused on information retrieving and analyzing before he moved to software engineering two years ago. His research interest is broad and mainly restricted in software engineering, including software testing and maintenance, software metrics and software quality.

RONG GUOPING born in 1977, obtained his B.Sc. degree from Nanjing University, China, in 2000, and now is a graduate student of the graduate school, Nanjing University. His research interest includes XML based systems and formal language analysis.

ZHOU PENG born in 1978, obtained his B.Sc. degree from Nanjing University, China, in 1999. After one year's teaching career, he is now a graduate student of Nanjing University. His research interest is mainly focused on parallel and distributed system and system visualization.

PAN JINGUI born in 1952, has been in research staff of Computer Science & Technology Dept., of Nanjing University since he graduated from Nanjing University in 1975. His research interest includes knowledge engineering and application and multimedia distance education. He is author or co-author of several books on software application and education and has published over 30 papers in journals or conferences since 1990. In recent years he has focused on middle ware, information retrieval and distance learning. He is a member of IASTED, also the editor of Computer Research and Development and Microcomputer Systems, China.