Appl. Math. Inf. Sci. 6 No. 1S pp. 1S-8S (2012)

Applied Mathematics & Information Sciences
An International Journal

@ 2012 NSP
Natural Sciences Publishing Cor.

A Code Automatic Generation Algorithm Based on Structured Flowchart

WU Xiang-Hu, QU Ming-Cheng, Liu Zhi-Qiang and Li Jian-Zhong
School of Computer Science and Technology, Harbin Institute of Technology, Harbin 150001, China;
Email Address: wuxianghu@hit.edu.cn; qumingcheng@126.com

Received June 22, 2010; Revised March 21, 2011; Accepted 11 June 2011
Published online: 1 January 2012

Abstract: In order to automatically convert structured flowchart (SFC) to problem analysis diagram (PAD) for generation of codes, by analyzing the characteristics of PAD and SFC, a structure identification algorithm is proposed for the structured flowchart. Taking the identified structured flowchart as input, a transformation algorithm is proposed to transform the structured flowchart into a semantically equivalent PAD. Then a specific language code is generated using recursive algorithm based on PAD. The effectiveness of the proposed algorithms of structure identification, transformation from flowchart to PAD and code automatic generation is verified through example test.

Keywords: Code automatic generation; Structured flowchart; Problem analysis diagram.

1 Introduction

Flow chart describes the control logic of a program by top-down process. For PAD (problem analysis diagram), it has the capability of top-down and left-right. So we can say if flowchart is a one-dimension chart, then PAD is a two-dimensional chart [1]. So, the transformation from flowchart to PAD can further enhance the readability of an algorithm, reduce the difficulty of a system design and improve the reliability and robustness of software [2].

Flowchart plays an important role in system requirement analysis, preliminary design and detailed design aspect [3]. Recently, there are some reports about the automatic generation of code from flowchart. However, these researches all have certain deficiencies, and the core algorithm and technologies are not public, so the accuracy and validity are hard to be convinced. More researches, such as “AthTek Code to FlowChart”, “Code to Chart”, “AutoFlowchart” etc, are just its reverse engineering, that is automatic generation of flowchart from code.

Hemlata Dakhore presented a strategy based on XML parser to generate code [4]. But the paper did not discuss how to identify the semantic of a specific flowchart. That is, the identification method of selection and loop are not discussed. According to the method, it must first determine whether a judgment node is a loop or selection, this information must be specified in advance by the modeler. If so it will lose the flexibility and convenience of a flowchart model, and also lack of automation and intelligence. And the paper only gives a sequence-selection simple example, for the algorithms of converting flowchart to XML and automatically generating code are not discussed. Martin C. Carlisle proposed a modeling and simulation system RAPTOR [5], which provides selection and loop primitives. This means that the modelers must know what kinds of structures they should draw in advance. While in standard flowchart there is only a judgment node, loop and selection nodes should be determined according to the semantic of a specific flowchart. So the RAPTOR is a specialized and non-standard graphical language. And this article only describes the functions of a system. Tia Watts gave a flowchart modeling tool SFC, which can be used to automatically generate code [6]. But its operation is mechanical, can only inserted pre-standard graphical elements from fixed points, the flexibility is very low, operation is not convenient, lack of scalability, do not support the component model. Most importantly, it does not support nested flowchart (processing nodes can be implemented as sub-flow chart).

By analyzing the characteristics of PAD and flowchart, a coding strategy is proposed, based on it we put forward a structure identification and coding algorithm, after then take the flowchart identified and coded in previous step as input, a algorithm which can convert structured flowchart to PAD is detailedly presented, at last we proposed a algorithm to generate code from PAD automatically.

2 PAD VS. Structured flowchart

Any complex algorithms can be composed of three basic structures, sequence, selection and loop. These basic structures can be coordinates, they can include each other, but they cannot cross and directly jump to another structure from the internal of a structure. As the whole algorithm is constructed by these three structures, just like composed by modules, therefore, it has the characteristics of clear structure, easily verifying accuracy, easy error correction [7-8].

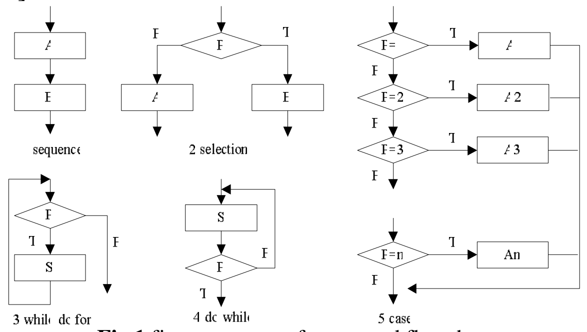

Fig.1 five structures of structured flowchart

Flowchart is independent of any programming language. Structured flowchart can be further divided into five kinds of structures: sequence, selection, more selection, pre-check loop and post-check loop, as shown in Figure 1. Any complex flow chart can be built by the combination or the nesting of the five basic control structures. Now there are many tools which support flowchart modeling, such as Visio, Word, Rose and so on.

PAD is the acronym for Problem Analysis Diagram. It is made by Japan Hitachi, evolved by flowchart. It has now been approved by ISO. Its advantage is clear, intuitive, and the order and hierarchy of program can be a good show. We can say that if the flow chart is a one-dimensional, then PAD is two-dimensional. A lot of people use PAD for system modeling at present in China and other countries. As shown in Figure 2, PAD has also set up five basic control structure primitives.

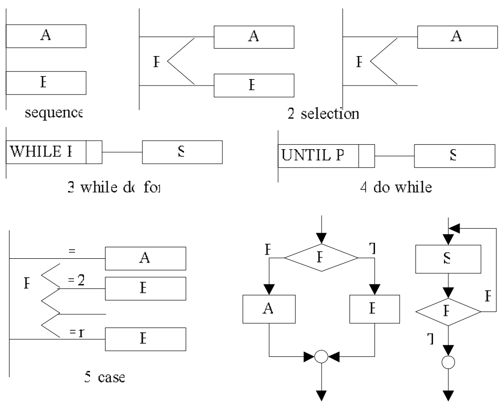

Fig.2 five structures of PAD

In order to make flowchart model more clear and intuitive and unambiguously, as shown in Figure 1, in addition to the order structure, the remaining four structures all use a judge node, when the executions exit their structures, the page reference primitive ("o") must be used. It is called "on page reference" in visio, in this paper is called convergence, as shown in Figure 2.

In this paper we use the most commonly used five kinds of primitives for flowchart to automatic generating of code, and they are: “Begin”, “End”, “Process”, “Judgment” and “Convergence”.

3 Structure identification

3.1 Identification method

(1) Identification of basic structure

For the three basic structures shown in Figure 3 the loop structure must be a cycle path, while the sequence and selection structures must not be. Figures 3-A and 3-B both have a cycle path. For a basic structure, if a cycle path occurs in a Process node for the first time, its current father (comes from) must be a Judgment, if not, the flowchart must be wrong. We can identify the Judgment as a do-while structure. If a cycle path occurs in a Judgment node, we can also identify its current father (Judgment node) as a do-while structure. If all the sons of a Judgment have been processed (return from their Convergence node), and the Judgment has not been identified, we can identify it as a Selection structure.

It can be seen from Figure 3 that the identification of while/for structure depends on its Judgment only;

and the identification of do-while must depend on the first node (see Figure 3, Judgment can exist in the nesting structure, as shown in Figure 4). The first node in a "do-while" structure, the Judgment of a "while" structure and the Convergence of a selection structure are all called **key nodes**.

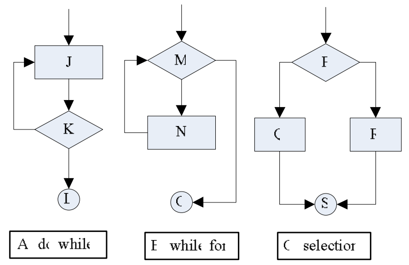

Fig.3 Three basic structures

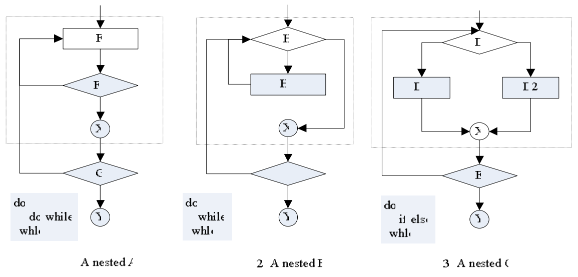

Fig.4 nesting structure of do-while

(2) Identification of nesting structure

According to the execution process of flowchart, the structure first executes to end must be the internal and basic structure. In Figure 4, nesting structures (1) (2) and (3) are constructed by the basic structures shown in Figure 3. As each basic structure completes (jump to their Convergence), the out layer structures are executed one by one. So if nesting structures exist, the internal structures must be identified firstly, and then the out layer.

As the identification of a While structure only depend the Judgment node itself (begins and finishes at itself), so if a cycle path appears in the Process node and its current father (comes from) is Judgment, then we can identify the father as a do-while structure. If the Process is the first node (key node) in multi-do-while, we should record the nesting level in the Process node and build a link between the Process node and its current father.

Similarly, if a Judgment node (JN) has been identified as a while/for or selection structure, and a cycle path again appears in the Judgment node, and its current father is Judgment node, then we can identify the father as a do-while structure. If the Judgment node (JN) is the first node (key node) in a multi-do-while structure, then we should record the nesting level in the Judgment node (JN) and build a link between the Judgment node (JM) and its current father.

As shown in Figure 4, the three figures are all nesting do-while structures. The white nodes in Figure 4 are all key nodes. In Figure 4-(1) there are two cycle paths in node F, and its current father F1 or G is Judgment, so F1 and G are both identified as do-while structures; as shown in Figure 4-(2), H is a key node of while structure, meanwhile it is a key node of outer layer do-while structure; as shown in Figure 4-(3), D is identified as Selection structure, then a cycle path appears in D, so D is the key node of the outer do-while.

In order to recursively traverse, every Judgment node must be able to have a direct access to its Convergence node, so it can jump current structure to traverse the outer nodes recursively. As a Judgment node and its Convergence are matched, when a Judgment has been traversed, its Convergence must be the subsequent one. So we can use a stack to match them. Define a stack as *StackofJudgement*, when a Judgment node is first in, we put it into *StackofJudgement*, when the execution arrive at a Convergence (as *currentConvergence*), pop the first node (as *currentJudgment*), and build a link between *currentJudgment* and *currentConvergence,* i.e., *currentJudgment.Convergence=currentConvergenc e*.

If the basic structures shown in Figure 3 are nesting by do-while, we can get the structures shown in Figure 4. While D, F1, H will be identified first, then cycle paths will again appear in F, H, D nodes, so we can know the outer structure must be do-while. Then E, G, I are identified as do-while structure. We should build links between them and G, I, E. Meanwhile the nesting level (as *doWhileCounter*) of G, I, E should do *doWhileCounter*++. The program can access G, I, E from D, F, H by the combinative conditions: get the *father* of (D,F,H) and *father.doWhileNode =(D,F,H)* and *father.doWhile- Counter=(D,F,H). doWhile-Counter.*

3.2 Algorithm description

We used a depth-first search algorithm based on recursion. The return conditions of recursion: no need return from sequence; when arrive at a Convergence or End return; when a Judgment has been Identified return, and jump the Convergence of Judgment to process the follow-up nodes.

We process all the sub-nodes recursively when the program arrives at a Judgment. When a Judgment is identified, return to recursive call point.

```
Stack StackofJudgement(Judgment); /* the elements of stack is Judgment, used to match Judgment and its corresponding
Convergnece */
Node root;   /*root is Begin node, so the code of first node is root.son*/
StructureIdentify(root, root.son);   /*start recursion*/
StructureIdentify(Father, Node)
{
   If(Node is Process)  [1]
      {
         If(Node has not be traversed)  [2]
           {
              StructureIdentify(Node, Node.Son);  [2-1]
           }
         else if(Father is Judgment)  [3] /* Include multiple do-while nesting */
           {
              Father.type←do-while;   /* recognized as do-while structure;*/
                 Node.doWhileCounter++; /*the original value is 0*/
                 Father. doWhileCounter= Node.doWhileCounter;
                 Father.doWhileNode= Node; /* build a link between the Judgement and the first Process of a do-while structure
                         */
                  }
   }
      If(Node is Judgment) [4]
      {
         If(Node has not be traversed) [5] /*first in*/
         {
            Stack.push(StackofJudgement, Judgment) /*push Judgment into StackofJudgement */
            for every son of Node do StructureIdentify(Node, Node.Son); [5-1]

               If(Node is not recognized) [6] /*loop structures have been recognized, the left is selections*/
               {
              /* according to the condition of judgment, the detailed structures of if-else/if/case can be recognized also.*/
                  Node.type←selection; /* recognized as selection structures; */
               }
               Node= Node. directJudgmentNode; /*Continue to process the nodes behind Convergence. */
               StructureIdentify(Node, Node.Son); [5-2] /*continue to code the other node after Convergence */
      }
      Else [8] /* traversed */
           {
              If(Node is not recognized) [9]  /*the first round trip*/
              {
                 Node.type←while or for structure /* recognized as while or for structures;*/
              }
              else [10]
              {
                 Father.type←do-while;   /* recognized as do-while structure;*/
                    Node.doWhileCounter++; /*the original value is 0*/
                    Father. doWhileCounter= Node.doWhileCounter;
                    Father.doWhileNode= Node; /* build a link between the Judgement and the first Process of a do-while structure
                         */
                 }
           }
      }
      If(Node is Convergence) [11]
      {
         If(Node has not been traversed) [12] /*match a judgment node and a convergence node*/
         {
            tempJudgeNode=Stack.Pop(StackofJudgement); /*use it when process the nodes behind Convergence */
            Node.directJudgmentNode= tempJudgeNode;
            tempJudgeNode.directJudgmentConvergence= Node;
            Node.code= tempJudgeNode.code;
           }
           Return;
      }
      If(Node is End) return;
}
```

3.3 Effectiveness verification of algorithm

As the algorithm is based on recursion, so we can use exhaustive method to verify its effectiveness, including the recursive entry and return. For the three basic structures shown in Figure 3, they nest with each other or their own can generate nine nesting structures We use these twelve structures to verify the effectiveness of the algorithm.

Take (1) in Figure 4 as an example: F goes into [1], then [2], execute [2-1] (recursion 1); F1 goes into [4], [5], and is push into stack, execute [5-1] (recursion 2); continue to process X or F (no effect), suppose F first enters [3], and F1 is identified as do-while structure, build the link between F and F1, recursive level (doWhileCounter) of node F is increased by 1, then return to [5-1] (recursion 2); node X goes into [12], F1 is popped from stack, construct the link between F and X, return to [5-1] (recursion 2), jump [6], execute [5-2] (recursion 3); process node G, G goes into [5], is pushed into stack, execute [5-1] (recursion 4); Y goes into [12], G is popped from stack, the link between G and Y is constructed, return to [5-1] (recursion 4); then F enters into [3], G is identified as do-while structure, construct the link between F and G, recursive level (doWhileCounter) of node F is increased by 1, return to [5-1] (recursion 4), jump [6], process the successor nodes of node Y.

It can be seen from the above process: First, the basic structure within the dashed box is identified as do-while, then the outer layer. Similarly we can check Figures 4-(2) and 4-(3), also the results are correct.

4 SFC to PAD

4.1 Algorithm description

Define a data structure as *BLOCK<Type, sequencePtr, levelPtr, code>*. Type denotes node type [such as: Sequence, Selection, while/for/do-while (loop)]; *sequencePtr* denotes sequence pointer; *levelPtr* denotes level pointer (only Judgment node), *code* denotes the code contained in a flowchart node.

As every new level of PAD only starts from a Judgment node, so we can use a structure contained two pointers to build a tree to depict PAD structure. As shown in Figure 5, black circles denote pointers, the down pointer is sequence pointer, and the right pointer is level pointer. The left figure is a SFC and the right one is its semantically equivalent data structure of PAD.

Node B in Figure 5 is a loop node, so its *levelPtr* should points to next level; and node D is a selection node, so its *levelPtr* should points to a pointer list, and each element of the pointer list is used to point each branch of selection structure. Obviously all the Judgment (loop, selection) must be recursively processed for their inner structures.

The algorithm takes the identified SFC as input, and use recursion to traverse. During the traverse process we can build a tree for PAD like the right figure in Figure 5.

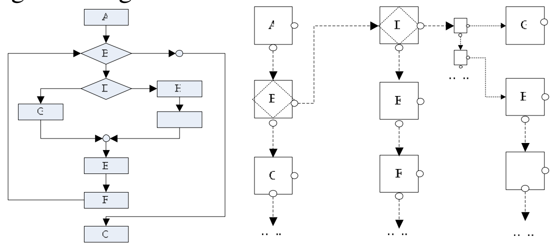

Fig.5 example of two-pointer constructing PAD

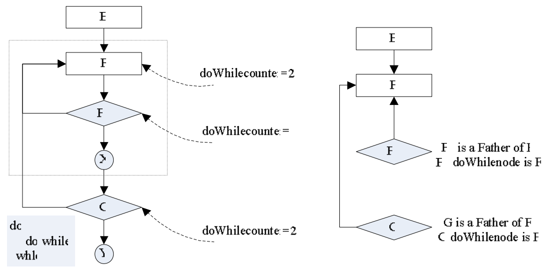

Fig.6 Combinative conditions to determine do-while nodes

The core algorithm is as follows:

```
Block firstBlk;
ConvertToPAD(Begin.son, firstBlk);
ConvertToPAD(Node CurrentNode, Block CurrentBlock, Block FatherBlock)
{
        If CurrentNode.doWhilecounter is not zero{    [1]
            Get the father of Node as tfather, and met:
                tfather.doWhilenode is CurrentNode, and tfather.doWhilecounter== CurrentNode.doWhilecounter,
                CurrentNode.doWhilecounter--;
                ConvertToPAD(tfather, CurrentBlock);  [1-1] /**/
        }
        If CurrentNode is Process {    [2]
              New block as newBlock;
              CurrentBlock.type=Sequence, CurrentBlock.code= CurrentNode.code;
              CurrentBlock.sequencePtr points to newBlock;
              ConvertToPAD(CurrentNode.son, newBlock, CurrentNode);    [2-1]
        }
        If CurrentNode.type is loop {    [3]
              [3-0]
              If CurrentNode has been processed {
                  free(CurrentBlock), FatherBlock.sequencePtr=null;
              }
              else{
                  New Block as levelBlock;
                  CurrentBlock.type= CurrentNode.type, CurrentBlock.code= CurrentNode.code;
                  CurrentBlock.levelPtr points to levelBlock;
                  CurrentNode = the son of CurrentNode who is not Convergence;
                  ConvertToPAD(CurrentNode, levelBlock);  [3-1]
              }
                New Block as SequenceBlock;   /*loop body has been processed*/
              CurrentBlock.sequencelPtr points to SequenceBlock;
              ConvertToPAD(Node.Convergence.son, SequenceBlock, CurrentBlock); [3-2]
              /*jump Convergence to process the other nodes*/
        }
        If CurrentNode.type is selection {    [4]
              For every branch of CurrentNode    [4-1]
              {
                  define a new block as BranchNewBlock(i),
                  BranchNewBlock.type= Branch;
              }
              Let CurrentBlock.levelPtr points to BranchNewBlock(1), CurrentBlock.code= CurrentNode.code,  [4-2]
              and BranchNewBlock(i). sequencePtr= BranchNewBlock(i+1);
              for every branch(i) of Node   [4-3]
              {
                  define a new block as LevelBlock (i);
                  Let levelPtr of BranchNewBlock(i) points to LevelBlock (i);
                  ConvertToPAD(branch(i), LevelBlock (i));  [4-3-1]
              }

                If All the sons of CurrentNode have been processed {   [4-4]
                    define a new block as SequenceBlock;
                    Let CurrentBlock.sequencePtr points to SequenceBlock;
                    ConvertToPAD(Node.Convergence.son, SequenceBlock, CurrentBlock); [4-4-1]
                }
        }
        If CurrentNode.type is End then Block.type=End, return;  [5]
        If CurrentNode.type is Convergence {
              If FatherBlock is Proces free(CurrentBlock), FatherBlock.sequencePtr=null;
              If FatherBlock is Judgment free(CurrentBlock), FatherBlock.LevelPtr=null;
              return;  [6]
        }
}
```

4.2 Effectiveness verification of algorithm

Take nested do-while structure in Figure 6 as an example to explain.

For do-while structure, its Judgment can be reached during the end of traverse, so when arrive at the first node of a do-while (a node whose doWhilecounter is not zero), we can use its dowhilecounter and its father's doWhilecounter to get the Judgment. As shown in Figure 6, node F is a key node (first node, see section 4.1) of two do-while structures. By the processing of structure identification algorithm in section 4, we can get: *F.doWhilecounter*=2, *F1.doWhilecounter*=1, *G.doWhilecounter*=2. And also we can use a method to get the fathers of F.

(1) First E goes into [2] (see section 5.1 row 16), the Block of E points to newblock, execute recursion [2-1], the result can be seen from Figure 7-(3).

(2) F goes into [1], by the condition of *F.doWhilecounter*=2, we can know F is a key node of a do-while. Then get the Father of F as tfather, **and** meet tfather.doWhilecounter is F.doWhilecounter **and** tfather.doWhilenode is F,

by the combinative conditions we can get the outermost Judgment node G, then do F.doWhilecounter--,doWhilecounter=1. Execute recursion [1-1], the newblock defined in step (1) is passed as a parameter, and no new block is defined here.

(3) G goes into [3], use the parameters of G to assign *newBlock* defined in step (1), define a new block as *levelBlock*, and make the *levelPtr* of *block* point to *levelBlock*, then get node F, execute recursion [3-1]. The result can be seen from Figure 7-(3).

(4) F enters [1], by *F.doWhilecounter*=1, we can know F is a key node of a do-while. Then get the Father of F as tfather, **and** meet tfather.doWhilecounter is F.doWhilecounter **and** tfather.doWhilenode is F, by the combinative conditions we can get the Judgment node F1, do F.doWhilecounter--, doWhilecounter=0. Execute recursion [1-1], the *newblock* defined in step (3) is passed as a parameter, and no new block is defined here.

(5) F1 goes into [3], use the parameters of F1 to assign *newBlock* defined in step (3), define a new block as *levelBlock*, and make the *levelPtr* of *block* point to *levelBlock*, then get node F, execute recursion [3-1]. The result can be seen from Figure 7-(5).

(6) F enters [2], use the parameters of *F* to assign *CurrentBlock*, define a new block as *newblock*, and make the *sequencePtr* of *CurrentBlock point* to *newblock*. Get *F1*, execute recursion [2-1]. The result can be seen from Figure 7-(6).

(7) F1 goes into [3], execute [3-0], free *CurrentBlock*, make the *sequencePtr* of *fatherblcok* be null, create a new block as *SequenceBlock*, and let the *sequencePtr* of *CurrentBlock* point to *SequenceBlock*, execute recusion [3-2]. The result can be seen from Figure 7-(7).

(8) G goes into [3], execute [3-0], free *CurrentBlock*, make the *sequencePtr* of *fatherblcok* be null, create a new block as *SequenceBlock*, and let the *sequencePtr* of *CurrentBlock* point to *SequenceBlock*, execute recusion [3-2], process the successor nodes of node Y. The result can be seen from Figure 7-(8).

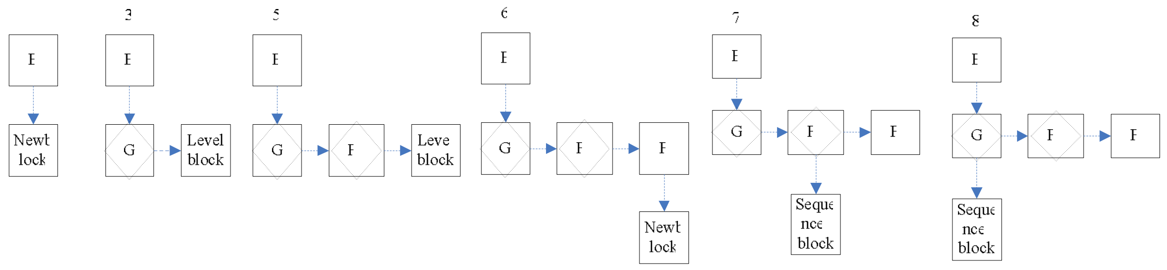

Fig.7 Result

5 Generation of code from PAD

From the characteristic of PAD, if depth-first search strategy is adopted, then the program chain we get will be equivalent to the execution order of code. So we can give the algorithm of generating code from a PAD easily.

The depth-first strategy is also used here. First define string *TempCode*, then pass it into *CodeGenerate()* fucntion, if encounter a process block, then append the program segment to *TempCode*; if encounter selection or loop blocks, then define a new string to store the program inside, recursively traverse the sub-nodes inside, and append that string to *TempCode* when return. So in the end, *TempCode* will contain the all program code.

```
String TempCode; /*When pass it into a function, all the operation to it whith that function will effect the original value, as it is address pass*/
CodeGenerate(beginBlock, TempCode);
PrintAndFormatCode(TempCode);   /*output and format the code*/
CodeGenerate(Block CurrentBlock, String CurrentCode)
{
      if(CurrentBlock.NodeType is Process)  CurrentCode.Append(CurrentBlock.codeBlock);
      else if(CurrentBlock.NodeType is Selection)
      {

         Generate branch code as SelectionCode;
         for every SubNode in CurrentBlock.SubNodeList do
            {
               Generate branch code branchCode;
```

```
                String branchBody;
                   CodeGenerate(SubNode, branchBody);
                   Insert branchBody into branchCode;
                   SelectionCode.Append(branchCode);
      }
      CurrentBlock.Append(SelectionCode);
      }
      else if(CurrentBlock.NodeType is HeadPtr)  /*If type is HeadPtr, then process the sub-nodes*/
      {
         for every SubNode in CurrentBlock.SubNodeList do CodeGenerate(SubNode, CurrentCode);
      }
      else if(CurrentBlock.NodeType is Loop)
      {
         Generate loop code as loopCode;
         String loopBody;
         for every SubNode in CurrentBlock.SubNodeList do CodeGenerate(SubNode, loopBody);
         insert loopBody into loopCode;
         CurrentBlock.Append(loopCode);
      }
      else return;
}
```

6 Conclusion

We proposed a structure identification algorithm for structured flowchart. The effectiveness of the proposed algorithm is checked using exhaustive method, i.e., twelve structures can be identified, then an algorithm can be used to convert a flowchart identified to PAD, and generate code from PAD using recursion algorithm. The technologies and algorithms are used in an integrated development platform, we develop a weapon system based on the platform to verify the effectiveness of the proposed algorithm.

References

[1] S. J. Le, W. L. Zhang and L. X. Dong. An Approach to Generate Scenario Test Cases Based on UML Sequence Diagrams. Computer Science, 31,2004:1-6

[2] J. T. Ostrand and M. J. Balcer. The Category-Partition Method for Specifying and Generating Functional Tests. Communication of ACM. 2006: 31-67

[3] S. Sendall and W. Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. IEEE Software, 9, 2003:42–45

[4] H. Dakhore and A. Mahajan. Generation of C-Code Using XML Parser, Proceedings of ISCET 2010, (2010) March 19-20; Punjab, India:1-6

[5] M. C. Carlisle, T. A. Wilson and J. W. Humphries. Raptor: introducing programming to non-majors with flowcharts. Journal of Computing Sciences in Colleges. 19,2004, 1-6

[6] T. Watts. The SFC editor: a graphical tool for algorithm development. J. Comput. Small Coll. 20, 2004, 73-85

[7] I. Nassi and B. Shneiderman. Flowchart techniques for structured programming. ACM SIGPLAN Notices, 8, 1973:12-26

[8] J. F. Gimpel. Contour: a method of preparing structured flowcharts. ACM SIGPLAN Notices. 15,1980:35-41

Xiang-Hu Wu is a professor in school of computer science and technology of Harbin institute of technology (HIT). He is a advanced member of CCF. His research interests include embedded computing, cyber-physical system, operating system and software engineering.

Ming-Cheng Qu is a Ph.D. in school of computer science and technology of Harbin institute of technology (HIT). He received his BS and MS degree from HIT. His research interests include embedded computing and software engineering etc.