Automatic Conversion of Structured Flowcharts into Problem Analysis Diagram for Generation of Codes

Xiang-Hu Wu
School of Computer Science and Technology, Harbin Institute of Technology, Harbin, 150001, China;
Email: Wuxianghu@hit.edu.cn

Ming-Cheng Qu, Zhi-Qiang Liu, Jian-Zhong Li
School of Computer Science and Technology, Harbin Institute of Technology, Harbin, China;
Email: Qumingcheng@126.com; Liuzhiqiangmar@sina.com; Lijz@hit.edu.cn

Abstract—**Compared with flowchart, problem analysis diagram (PAD) can not only be used to describe the sequence of program but also the hierarchy structure. It is of great significance to automatically convert flowchart to PAD for generation of codes. There are some deficiencies in existing researches, and their key algorithms and technologies are not elaborated. By analyzing the characteristics of PAD and structured flowchart, a coding strategy is proposed, and a structure identification and coding algorithm are put forward for structured flow diagram and PAD. Based on the coding strategy a transformation algorithm which can transform flowchart into a semantically equivalent PAD is proposed. Then by using recursive algorithm the specific language code are generated from PAD. Finally a integrated development platform is developed using such algorithms, including flowchart modeling, code automatic generation, and CDT\GCC\GDB. The correctness and effectiveness of coding strategy and algorithm, the structural transformation from flowchart to PAD, and automatic generation of codes based on PAD have been verified through practical operations.**

Index Terms—**Automatic generation of codes; structured flowchart; identification of structure; problem analysis diagram; integrated development platform**

I. Introduction

Software development ideas based on Model Driven Architecture (MDA) have attracted much attention from the research community in recent years[1,2]. MDA is first proposed by OMG. It is a methodology and standard system by which software systems are built on the basis of a variety of models, through model transformation to drive system development[3]. The development of Model-driven software is a hot issue in the current field of software engineering, and it has become a new software development paradigm to improve the quality and efficiency of software development[4].

Standard flowchart plays an important role during requirement analysis, overall design and detailed design of software development[5]. However the use of flowchart is limited to display, communication and description only to make communication and document formation easy. Direct conversion of flowchart into a specific language code is more in line with the objective of MDA[6,7].

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[8]. 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[9].

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.

II. Related Work

Currently, automatic code generation is a mainstream technology for real-time embedded system application. There is a number of code generation technologies based on object-oriented and state diagram reported in recent years[10-13]. Now Rose, Rhapsody, Matlab and other automated development tools already have these features. The concept of real-time framework is adopted by many commercial IDEs. Such as the "ROOM virtual machine" of "Real-time Object-Oriented Modeling", the "visualSTATE engine" of "IAR visualSTATE" and the "Object Execution Framework (OXF)" of "I-Logix Rhapsody". But for strong real-time weapon systems, the code should be refined as much as possible, so it will lead to bad efficiency for the software system based on real-time framework.

Hemlata Dakhore presented a strategy that bases on XML parser to generate code. But the paper did not

doi:10.4304/jsw.7.5.1109-1120

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[14].

Martin C. Carlisle proposed a modeling and simulation system RAPTOR, 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[15].

Tia Watts gave a flowchart modeling tool SFC, which can be used to automatically generate a code. But its operation is mechanical, can only be inserted into pre-standard graphical elements at fixed points, the flexibility is very low, operation is not convenient, lack of scalability, and it does not support the component model. Most importantly, it does not support nested flowchart (processing node can be implemented as a sub-flow chart)[16]. Kanis Charntaweekhun simply introduced the method of how to use flow chart for programing and its advantages, and said that the developed system can transform flowchart into code. But, the conversion algorithm, key technologies and data structures are not mentioned, and the examples given are very simple[17].

III. MAIN WORK AND CONTRIBUTION

Main contribution By analyzing the characteristics of PAD and flowchart, a coding strategy is proposed, we put forward a structure identification and coding algorithm based on it, after taking the flowchart identified and coded in previous step as input, a algorithm which can convert structured flowchart to PAD is presented in detail, at last we proposed a algorithm for automatic generation codes from PAD.

We designed and implemented an integrated development platform based on Eclipse and algorithms presented above. The platform uses a structured flowchart to describe program logic and convert flowchart model into standard ANSI-C code. Code editor (CDT), compilation tools (GCC) and debugging tools (GDB) are all integrated into this platform.

(1) Build the system with flowchart: Tasks and interrupt service can be modeled, platform can generate an instance of the task or interrupt, and support nested flowchart model and code generation.

(2) Variables and head-file management: Management of global variables, local variables, macros, and various header files.

IV. MAIN WORK AND CONTRIBUTION

A. PAD vs structured flowchart

Any complex algorithm can be composed of three basic structures, i.e., sequence, selection and loop. These basic structures can be coordinated, so that they can include each other, but they can not 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 modules, and so, it has the characteristics of clear structure, easily verifying accuracy, easy error correction[18,19].

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 nest of the five basic control structures. Now there are many tools which support flowchart modeling, such as Visio, Word, Rose and so on.

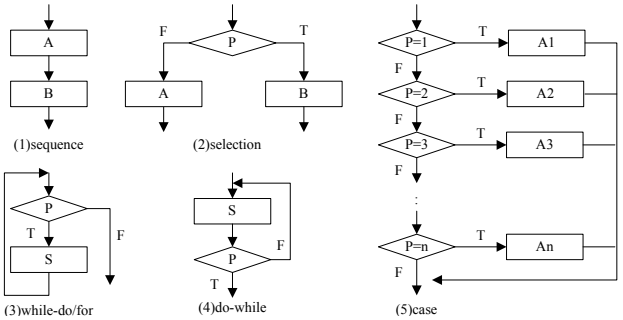

Figure 1. Five structures of structured flowchart

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, PAD is then 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.

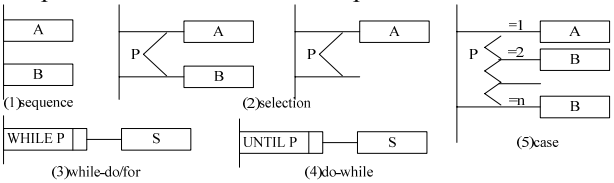

Figure 2. Five structures of PAD

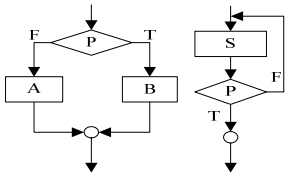

Figure 3. Examples of convergence

In order to make flowchart model more clear and intuitive and unambiguously, as shown in Figure 3, 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, and in this paper it is called convergence, as shown in Figure 5.

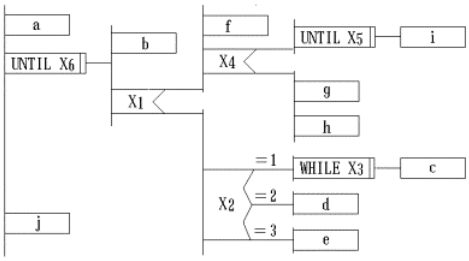

Figure 4. Flow of program expressed by PAD

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

B. Idea for automatic generation of codes

The semantics of PAD and flowchart can be transformed to and from each other. But in comparison with flowchart, PAD can more clearly depict the order and hierarchy of program. A PAD will start from the top of the left vertical line, and execute in Top-down order. When it encounters judgment or loop, it will go into the next level from left to right, and will continue from the top of the vertical line at the next level until it reaches the bottom of that vertical line, and then return to the exit position of the previous level. So it continues until the execution arrives at the bottom of main vertical line (the left one). If we can effectively travel onto every node in a PAD, and when the depth-first traversal strategy is used, the traversal process is just consistent with the execution order of PAD. An example from PAD to flowchart is shown in Figure 7.

The flowchart equivalent to Figure 4 is shown in Figure 5. In Figure 7 each selection and loop structure end at their convergences is named by the judgment condition of judgment node. The analysis of semantic is as follows: first execute code a, then a loop “UNTIL X6”, this loop corresponds to a "do-while (X6)" structure. Code block b will execute first in structure "UNTIL X6", then X1(if-else structure), when "UNTIL X6" finishes, code block j will executes. The execution order and hierarchy of program can be seen clearly in Figure 4, but it is not obvious and intuitive in Figure 5.

Main ideas: Firstly, convert flowchart into PAD in consistent semantics, and then adopt depth-first search strategy to generate code from PAD.

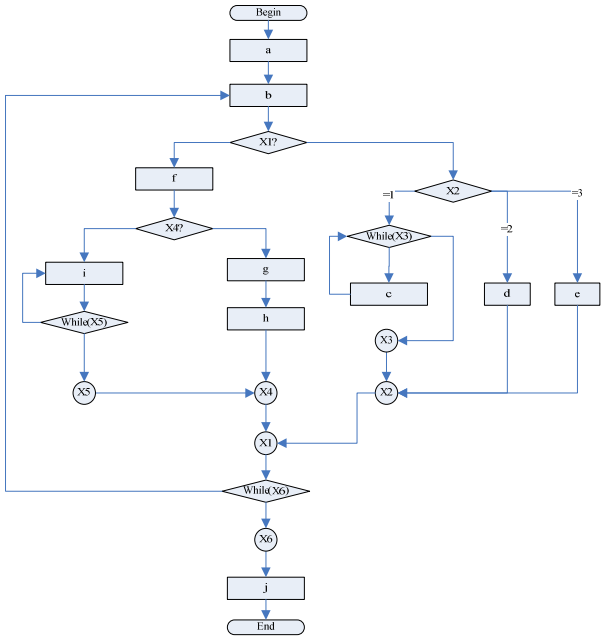

Figure 5. Flowchart semantically equivalent to Figure 4

C. Node coding strategy

It can be seen from figure 6 the order of program can be represented by longitudinal line and the level can be represented by the transverse layer. So each primitive node can be located by sequence coding and level coding. From PAD every layer expands with judgment node. After expanding, structures can be loop (while/for/do-while) or judgment (if-else/if/case). If each node of a flowchart has a unique code to express the sequence and level position, we can convert flowchart into PAD using the unique code.

Coding strategy: (1) The basic unit of coding for every node is a number in the form of XXYY (length is 4); (2) where XX is the coding number of branches for judgment node, and start counting from 00; (3) YY is a sequence code which increases one by one in the same layer, and starts counting from 00; (4) if the level increases one(the follow-up node of judgment), the length of code will increase by 4; (5) The node code of a new layer inherits from the code of its direct judgment node and append four numbers at its tail; (6) The code of every convergence is the same as the one of its corresponding judgment node.

The code for the nodes in figure 6 are：Node1=0000, Node2=0001, Node3=0003, Node4=00010000, Node5= 00010001, Node6= 000100010000, Node7= 00010001-0001 and Node8=000100010100.

From the coding strategy: (1) The hierarchy of a PAD can be learned from the code length; (2) the productions of levels all start from judgment node; (3) The code length of judgment node is four longer than the ones in the direct follow-up layer.

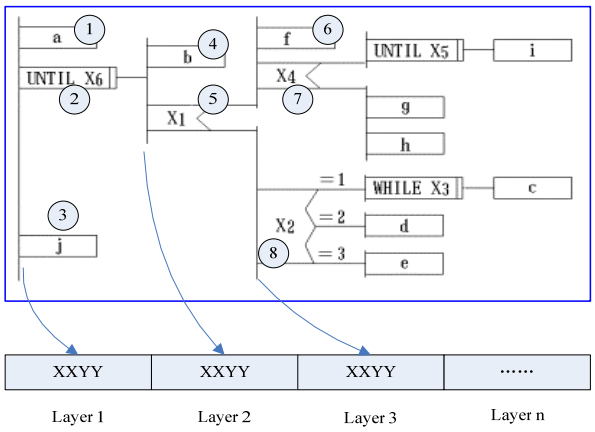

Figure 6. Correspondence between code and PAD

The codes of each node in Figure 5 are: Node1=0000, Node2=0001, Node3=0003, Node4= 00010000, Node5=00010001, Node6=000100010000, Node7= 000100010001, and Node8=000100010100.

It is known from the strategy that it is suitable for PAD and flowchart, because all the layers start from the judgment node.

Special regulation: the code of Convergence must be the same as the one of its corresponding Judgment.

V. STRUCTURE IDENTIFICATION AND NODE CODING

A. The minimum transmission time model (A)

(1) Identification of basic structure

For the three basic structures shown in figure 7, the loop structure must be a cycle path, while the sequence and selection structures must not be. Figures A and 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 (where it 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 7 that the identification of while/for structure depends on its Judgment only; and the identification of do-while must depend on the first node (Process or Judgment: node J in A of figure 7, Judgment can exist in the nested structure, as shown in figure 8). 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**.

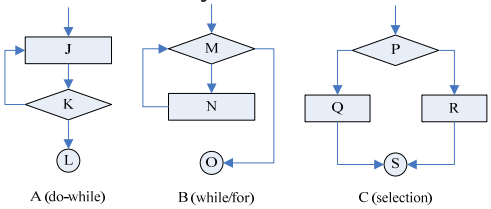

Figure 7. Three basic structures

(2) Identification of nested structure

According to the execution process of flowchart, the structure first executes to end must be the internal and basic structure. In figure 8, nested structures (1) (2) (3) are constructed by the basic structures shown in figure 7. As each basic structure completes (jump to their Convergence), the out layer structures are executed one by one. So if nested 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 (where it 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 nested 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 nested level in the Judgment node (JN) and build a link between the Judgment node (JM) and its current father.

As shown in In figure 8, the three figures are all nested do-while structures. The white nodes in figure 8 are all key nodes. In figure (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 (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 (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.

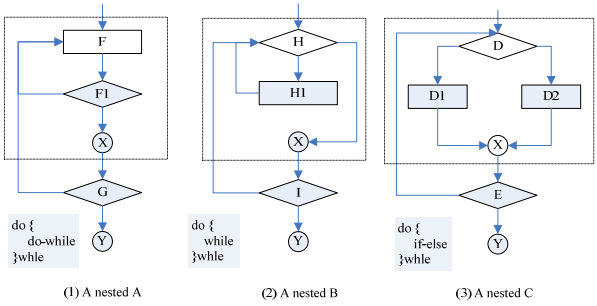

Figure 8. Nested structure of do-while

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 is 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 arrives at a Convergence (as *currentConvergence*), pop the first node (as *currentJudgment*), and build a link

between *currentJudgment* and *currentConvergence,* i.e.*, currentJudgment.Convergence= currentConvergence*.

If the basic structures shown in figure 3 are nested by do-while, we can get the structures shown in figure 8. 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 nested 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.doWhileCounter =(D,F,H).doWhileCounter.*

B. Structure recognization and coding algorithm

General description

We used a depth-first search algorithm based on recursion, which consists of two parts: one is structure identification and coding (called ***CodeAlgorithm***), the other one is recode for do-while (called *Recode*). The nodes are coded during identification process.

Here a problem appears, as the Judgment of do-while structure appears in the end of a structure, so if we code from top-down during the traverse, the key node of do-while structure will get the highest level coding and Judgment node will get the lowest level coding. This is different from the coding strategy above. To keep consistentency, we should recode all the nodes in a do-while structure when the Judgment is identified as do-while. So we give a recode algorithm (called ***Recode***) for do-while structures which is called in ***CodeAlgorithm*** when a do-while is identified.

As a layer starts from a Judgment, we should process all the sub-nodes recursively when the program arrives at a Judgment.

The return conditions of recursion: no need return from sequence; when arrive at a Convergence return; when a Judgment has been Identified return, and jump the Convergence of Judgment to process the follow-up nodes.

```
(1)  :  /*************************************************************************
(2)  :  Function：Generate a code for every node according to the coding strategy.
(3)  :  Input：All the nodes of flowchart.
(4)  :  Output：The flowchart coded
(5)  :  *************************************************************************/
(6)  :  Stack StackofJudgement(Judgment); /* the elements of this stack is a Judgment, used to match Judgment and its convergnece */
(7)  :  Node root;   /*root is Begin node, its code is 0000, so the code of first node is 0001*/
(8)  :  Let the code of node be a string;
(9)  :  CodeAlgorithm(root, root.son, 0000)   /*start up recursion*/ [0]
(10) :  CodeAlgorithm(Node Father, Node CurrentNode, String CurrentrCode)
(11) :  {
(12) :        If(CurrentNode is Process) [1]
(13) :        {
(14) :           CurrentNode.code= CurrentCode   /*Generate the code for CurrentNode;*/   [1-1]
(15) :           Increase YY of CurrentCode by one→ CodeofSon
(16) :           If(CurrentNode has not been traversed) CodeAlgorithm(CurrentNode, CurrentNode.Son, CodeofSon);
(17) :            [2]
(18) :         Else
(19) :           {
(20) :              /*Father is recognized as a do-while structure, mark the Judgment and link it with his Father, include do-while and nested
(21) :                        do-while */
(22) :              if(Father is Judgment and Father has not been recognized)    [3]
(23) :                {
(24) :                      Father.Type←do-while;
(25) :                      CurrentNode.doWhileCounter++;  /*original value is zero*/
(26) :                      Father.doWhileNode=CurrentNode;
(27) :                      Father. doWhileCounter= CurrentNode.doWhileCounter;
(28) :                      CurrentNode.doWhileRecodeCounter=CurrentNode.doWhileCounter;/*used for recode*/
(29) :                 }
(30) :          if(CurrentNode.doWhileRecodeCounter>0)  Recode(CurrentNode);   [4]
(31) :       }
(32) :    }
(33) :        If(CurrentNode is Judgment) [5]
(34) :        {
(35) :
(36) :          If(CurrentNode has not been traversed)  /*first in*/  [5-2]
(37) :          {
(38) :             CurrentNode.code= CurrentCode   /*Generate the code for CurrentNode;*/   [5-1]
(39) :             Stack.push(StackofJudgement, Judgment)  /*push Judgment into StackofJudgement */   [6]
(40) :
(41) :                For every son j of CurrentNode (except for convergence) do  /*j starts from 0*/   [7]
(42) :                {
(43) :                   CurrentCode =currentCode+0j00；     /*nex layer*/
(44) :                   CodeAlgorithm(CurrentNode, CurrentNode.Son, CurrentCode);   [7-1]
(45) :                }
(46) :                If CurrentNode has a son of convergence( as convergenceSon )   [7-2]
(47) :                              CodeAlgorithm(CurrentNode, convergenceSon, null);
(48) :
(49) :                [8]
(50) :                If(CurrentNode is not recognized)  /*loop structures have been recognized, the left is selections*/
```

```
(51) :                  {
(52) :                 /* according to the condition of judgment, the detailed structures of if-else/if/case can be recognized also.*/
(53) :                     CurrentNode.type←selection;  /* Recognized as selection structures; */
(54) :                  }
(55) :                  /*Continue to process the nodes behind Convergence. */
(56) :                  CurrentNode=CurrentNode.directJudgmentNode;
(57) :                  Increase YY of CurrentCode.Code by one→ CodeofSon
(58) :                  CodeAlgorithm(CurrentNode, CurrentNode.Son, CodeofSon);  [9]
(59) :        }
(60) :        Else  /*been traversed */
(61) :            {
(62) :              If(CurrentNode is not recognized)  /*the first round trip*/  [10]
(63) :              {
(64) :                 CurrentNode.type←while or for  /* recognized as while or for structures;*/
(65) :              }
(66) :                 Else  /*been traversed and recognized*/
(67) :                 {
(68) :                  if(Father is Judgment and Father has not been recognized)  [11]
(69) :                      {
(70) :                          Father.Type←do-while;
(71) :                          CurrentNode.doWhileCounter++;  /*original value is zero*/
(72) :                          Father.doWhileNode=CurrentNode;
(73) :                          Father. doWhileCounter= CurrentNode.doWhileCounter;
(74) :                          /*used for recode*/
(75) :                          CurrentNode.doWhileRecodeCounter=CurrentNode.doWhileCounter;
(76) :                      }
(77) :                 if(CurrentNode.doWhileRecodeCounter>0)  Recode(CurrentNode,null);  [12]
(78) :                 }
(79) :            }
(80) :         }
(81) :         If(CurrentNode is Convergence)
(82) :         {
(83) :           If(CurrentNode has not been traversed)  /*match a judgment node and a convergence node*/ [13]
(84) :           {
(85) :             /*as a judgment and a convergence is exist geminate, so the top judgment in stack must be able to
(86) :                        match the current convergence */
(87) :                tempJudgeNode=Stack.Pop(StackofJudgement);
(88) :             CurrentNode.directJudgmentNode= tempJudgeNode;
(89) :             tempJudgeNode.directJudgmentConvergence= CurrentNode;
(90) :             CurrentNode.code= tempJudgeNode.code;
(91) :           } else Return;  [14]
(92) :         }
(93) :         If(CurrentNode is End) return;  [15]
(94) : }
(95) :
(96) : /*************************************************************************
(97) : Function: Recode do-while.
(98) : Input: When first enter, the first parameter is the first node in do-while and the second is null.
(99) : Output: Recoded flowchart.
(100) : *************************************************************************/
(101) : Stack LoopReturnStack(Judgment);  /*used by recursion return codition */
(102) : Recode(Node CurrentNode, String CurrentCode)
(103) : {
(104) :      If (is not first in)  [R1]
(105) :      {
(106) :           if(CurrentNode.type is loop)
(107) :           {
(108) :              if (LoopReturnStack.top!= currentNode) Push currentNode into LoopReturnStack;  [R2]
(109) :            else  [R3]
(110) :              {
(111) :                 LoopReturnStack.pop;
(112) :                   return;
(113) :           }
(114) :        }
(115) :      }
(116) :      If (CurrentCode!=null) CurrentNode.Code= CurrentCode;  [R4]
(117) :
(118) :   if(CurrentNode.doWhileRecodeCounter>0)
(119) :      {
(120) :         [R5]
(121) :           Get a father (as tfather) of CurrentNode , and meet:
(122) :                    (tfather.doWhileCounter= doWhileRecodeCounter
(123) :                    and tfather.doWhileNode=CurrentNode);
(124) :           CurrentNode.doWhileRecodeCounter--;
(125) :
(126) :           Recode(tfather, CurrentNode.Code); /*Let the code of tfather be the code of CurrentNode*/ [R6]
(127) :           if(CurrentNode.doWhileRecodeCounter==0)  [R7]
```

```
(128) :                    CurrentNode.doWhileRecodeCounter= CurrentNode.doWhileCounter;
(129) :    }
(130) :
(131) :       If(CurrentNode.type is Process)  [R8]
(132) :       {
(133) :          CurrentCode←CurrentCode.YY+1;  /*the sequence part of code increase one*/
(134) :       Recode(CurrentNode.son, CurrentCode,);   [R8-1]
(135) :
(136) :       }
(137) :       Else if(CurrentNode.type is selection)
(138) :       {
(139) :          For every son j of CurrentNode (except for convergence) do  /*j starts from 0*/  [R9]
(140) :          {
(141) :                   CurrentCode =currentCode+0j00;     /*nex layer*/
(142) :                   Recode(son, CurrentCode);
(143) :          }
(144) :          Recode(CurrentNode.directJudgmentConvergence.son, CurrentCode);  [R10]
(145) :       }
(146) :       Else if(CurrentNode.type is loop)  [R11]
(147) :       {
(148) :          CurrentCode =currentCode+0000;    /*nex layer*/
(149) :          Recode(CurrentNode.son, CurrentCode);   /* CurrentNode.son is not the convergence */ [R11-1]
(150) :          If(LoopReturnStack.top==null) return;  /*return to CodeAlgorithm*/   [R13]
(151) :             CurrentCode←CurrentCode.YY+1;
(152) :             Recode(CurrentNode.directJudgmentConvergence.son, CurrentCode);   [R11-2]
(153) :       }
(154) :       Else return;   [R12]
(155) :  }
(156) :
(157)
```

C. Time complexity analysis

If no do-while exists, the algorithm will traverse all the nodes one by one, so the time complexity is $O(n)$. If there are do-while structures, the algorithm must recode all the nodes inside. Let the nested do-while be $L_1, L_2,\ldots, L_{(i-1)}, L_i$ from inside to outside. As when coding for laye-i, the inside layer (i-1),…, layer-0 must be recoded, so from L_1 to $L_{(i+1)}$, the code times are $L_1=i, L_2=i-1,\ldots, L_{(i-1)}=2, L_i=1$. Let the sum of nodes in L_1 to $L_{(i+1)}$ are $X_1, X_2,\ldots, X_{(i-1)}, X_i$. Then the total execution times $SUM=iX_1+ (i-1)X_2+,\ldots,+ 2X_{(i-1)}+ X_i$.

In the worst case, let the most nodes kN lie in the innermost do-while structure, where N is the sum of nodes, and 0<k<1. So it can be approximately regarded that: $X_1=X_2= \ldots =X_i=kN$, $SUM = kN\frac{i(i-1)}{2}$. The traverse time for the other nodes outside is less than N, so the time complexity will be $O(i^2N)$. Obviously the execution time will increase as nested level of do-while increases.

D. Effectiveness verification of algorithm

As the algorithm is based on recursion, so we can use an exhaustive method to verify its effectiveness, including the recursive entry and return. For the three basic structures shown in figures 6, they nest with each other or their own can generate nine nested structures, as shown in figures 8, 9 and 10. We use these twelve structures to verify the effectiveness of the algorithm.

Take figure (2) in Figure 8 as an example, let ***CurrentrCode*** an input of ***CodeAlgorithm*** be 0001. H enters into [5-2], and executes [5-1], H is coded by 0001, then execute [6], H is put into stack, execute [7], generate code 00010000. Here H has two sons, take H1, then execute [7-1](recursion-1), enter [1], execute [1-1], H1 is coded by 00010000, then generates new code 00010001, continue to process H, execute[2](recursion 2), enter [5], enter [10], H is identified as while/for. Return to [2](recursion 2), return to [7-1](recursion 1), all the sons of H have been processed. Execute [7-2](recursion 3), enter [13], H is poped and matched with X, and X is coded by 0001 (the same with H). Then return to [7-1](recursion 3), jump [8], execute [7], generate code 00020000, execute [7-1](recursion 5), and process H. Then H enter [11], node I is identified as do-while, H and I are linked and the nested level is recorded. Continue to execute [12], pass H into ***Recode*** function to recode it. When return from ***Recode***, directly return to [7-1](recursion 5), execute [7-2], and continue to process the sequent nodes behind Y.

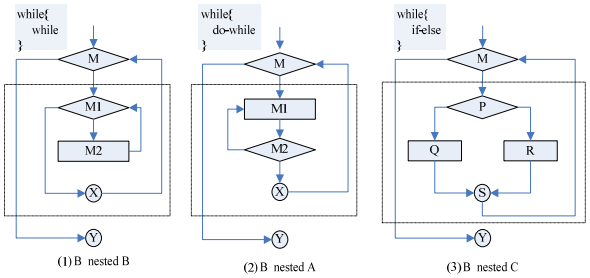

Figure 9. Nest of while

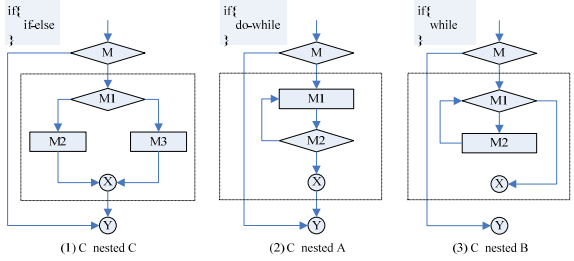

Figure 10. Nest of selection

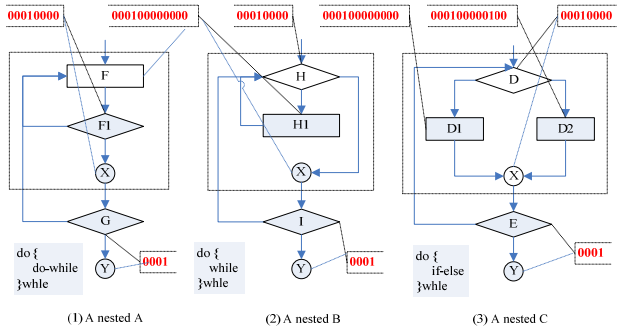

Figure 11. Figure 8 coded

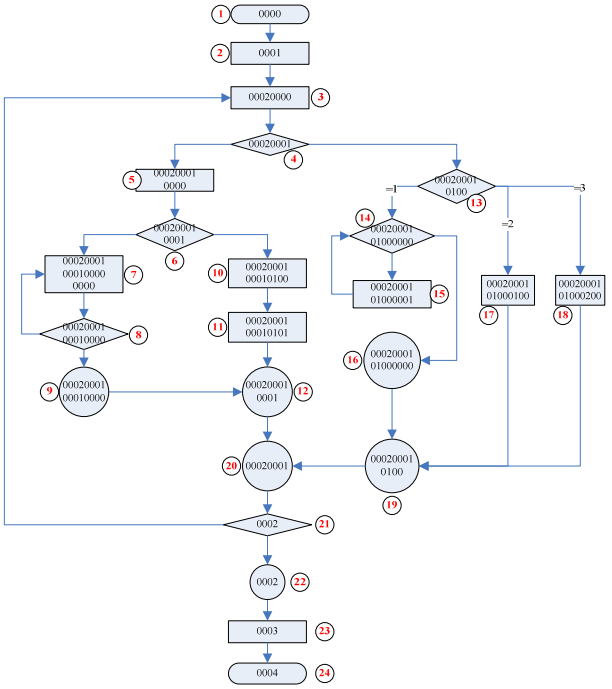

Figure 12. Coded flowchart of figure 5

Enter ***Recode*** function, if first in, directly execute [R5], by the link between H and I created previously, node I will be taken as ***tfather***. Eexcute [R6](recursion C1), take code 0001 of H and node I (***tfather***) as inputs. Enter [R1], execute branch [R2], put H into stack, then execute [R4], I is coded by 0001, enter [R11], generate code 00010000. Continue to process, execute [R11-1] (recursion C2). H goes into [R1], execute branch [R2], H is put into stack, execute [R4], H is coded by 00010000. Enter [R11], generate code 000100000000, execute [R11-1] (recursion C3), and process H1. H1 goes into [R4], and is coded by 000100000000, and then go into [R8]. Generate code 000100000001, execute [R8-1] (recursion C4), and process H. Then H goes into [R1], [R3], return to [R8-1] C4, return to [R11-1] C3, and generate code 00010001. Jump [R13], execute [R11-2] (recursion C5), jump X, continue to process I, input code 00010001, I goes into [R1], [R3], then return to [R11-2] (C5), and return to [R11-1] (C2). Now the program has returned to I, and *LoopReturnStack* is empty, execute R[13], return to [R6] (recursion C1), execute [R7], reset *doWhileRecodeCounter*(the outer may be nested by do-while, so the code must be reseted). Then return to ***CodeAlgorithm***.

According to the algorithm above, make code for Figure 5, output the coded flowchart as shown in Figure 12.

VI. TRANSFORMATION FROM FLOWCHART TO PAD

A. Description of algorithm

The depth-first strategy is used. Here the code of node is defined as string, so opterator "+" means "append".

Define hashtable *List<code, node>,* and let code be *key*.

Define function *node List.GetNode(code)*, and search the node by key.

Construct six instances of hash List, as *List4*, *List8, List12, List16, List20, List24,* they will be used to store nodes with code length from four to twenty four respectively.

Define a data structure *Block<NodeType, codeblock, SubNodeList >*. *NodeType*: the type of node; *codeblock*: the program segment of current node; *SubNodeList* is used to store all the nodes linked to it at the same level, and the storage order is the execution order of a PAD.

Convertion Process: recursion technology is used, fisrt scan the coded flowchart, push the nodes into responding hashtables, define a new *Block* as *newBlock*. The startup code is 0001, get the coresponding node of 0001 from hashtable as *currentNode*. If *currentNode* is *Process*, then *newBlock.NodeType=Process, newBlock.codeblock =currentNode.Codeblock,* and *newBlock.SubNodeList= null*; then *currentCode*←(the sequence part YY of currentCode increase one). If *NodeType* is selection, then get the sum of its sons as i, generate the next layer code: *currentCode+0100, currentCode+0000… currentCode+ 0(i-1)00*, get the corresponding nodes from hashtable and put them into *currentNode.SubNodeList*, then recursively process the their sub-nodes. If *currentNode* is *Loop*, then directively generate code *currentCode*+0000, and recursively process its sub-nodes.

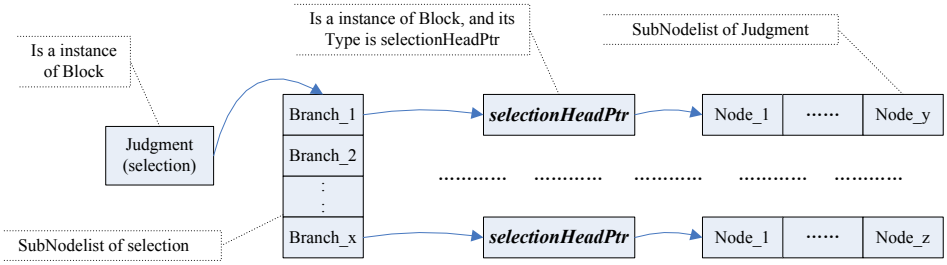

Figure 13. Data structure of a selection node

The following is the process method of the sub-nodes for judgment node: define a new Block *selectionHeadPtr* for every branche of selection structure, put every *selectionHeadPtr* into *SubNodeList* of current judgment node, and let *selectionHeadPtr*.Type=*HeadPtr*, then recursively put all the nodes of every branche into the *SubNodeList* of *selectionHeadPtr*, and what we can get is a data structure as shown in figure seven.

```
(1)  :  /************************************************************
(2)  :  Function: Convert the coded flowchart into PAD.
(3)  :  Input: Six hashtables, the coded flowchart.
(4)  :  Output: A PAD.
(5)  :  ************************************************************/
(6)  :  Define currentCode as string;
(7)  :  Node GetFromHashTable(currentCode)  /*Get the node from hashtable by GetNode() method*/
(8)  :  Put the nodes(except for end and convergnece nodes) into HashTable according to their code length;
(9)  :  Block beginBlock;
(10) :  beginBlock.NodeType=Begin;
(11) :  ConvertToPAD(0001, beginBlock);   /*the code of Begin is 0000, so original code is 0001*/
(12) :  ConvertToPAD(currentCode, Blcok fatherBlock)
(13) :  {
(14) :      CurrentNode= GetFromHashTable(currentCode);
(15) :      If (CurrentNode is null))  return;   /* recursion return condition*/
(16) :      Block newBlock;
(17) :      If (CurrentNode is Process)
(18) :      {
(19) :        currentCode←currentCode.YY+1;  /*the sequence part of code increase one*/
(20) :        newBlock.NodeType=Process;
(21) :        newBlock.codeblock=currentNode.Codeblock;
(22) :          newBlock.SubNodeList= null;
(23) :          fatherBlock. SubNodeList.append(newBlock);
(24) :          ConvertToPAD(currentCode, fatherBlock);   /*as in the same layer, so the fatherBlock is the same*/
(25) :      }
(26) :      Else If (CurrentNode is Judgment)
(27) :      {
(28) :        If(CurrentNode.type is selection)
(29) :        {
(30) :          i←the sum of sons of CurrentNode;
(31) :              newBlock.NodeType= selectionType;  /*here selectionType represents the types of selection*/
(32) :              newBlock.codeblock=currentNode.Codeblock;
(33) :              fatherBlock.SubNodeList.append(newBlock);
(34) :
(35) :          for(j=0;j<i;j++)
(36) :          {
(37) :              Block selectionHeadPtr;   /*every branch has its owe one*/
(38) :              selectionHeadPtr.NodeType=HeadPtr;
(39) :              newBlock.SubNodeList.append(selectionHeadPtr);
(40) :              TempCode =currentCode+0j00；    /*nex layer*/
(41) :                      ConvertToPAD(currentCode, selectionHeadPtr); /*begin a new layer, so the fatherBlock is
(42) :                                                                    newBlock */
(43) :              }
(44) :              /*as all the sub-nodes have been coded, so process the nodes at the same level */
(45) :               currentCode←currentCode.YY+1;
(46) :              ConvertToPAD(currentCode, fatherBlock);
(47) :        }
(48) :        If(CurrentNode.type is loop)
(49) :        {
(50) :              currentCode =currentCode+0000；/*into next layer*/
(51) :              newBlock.NodeType= loopType;  /*here loopType represents the types of selection*/
(52) :              newBlock.codeblock=currentNode.Codeblock;
(53) :              ConvertToPAD(currentCode, newBlock);   /*begin a new layer, so the fatherBlock is newBlock */
(54) :              /*as all the sub-nodes have been coded, so process the nodes at the same level */
(55) :              currentCode←currentCode.YY+1;
(56) :              ConvertToPAD(currentCode, fatherBlock);
(57) :        }
(58) :      }
(59) :  }
```

B. Effectiveness verification of algorithm

We also take an exhaustive method to check the correctness. Take the twelve structures built previously as input, take Figure 11-(1) as an example:

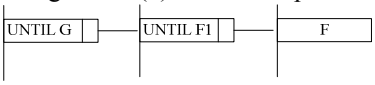

Figure 14. A PAD generating from figure

Execute code line-8, construct a hashtable. Here node "End" and "Convergence" will not be put into. Then execute code line-9, create a *Begin* block, take code 0001 and begin block as inputs to start recursion *ConvertToPAD*. Then execute [C1], by code 0001 we can get node object G into *CurrentNode*, execute [C1-2] to create a new block, then G goes into [C3], [C8], execute [C8-1] to generate the first sub-node code as 00010000, set the type loop to current block, then execute [C8-2], append G at *Begin* node, execute

[C9](recursion 1). Continue to execute [C1], by code 00010000 we can get node object F1, F1 goes into [C3], [C8], then execute [C8-1], generate the first sub-node code 000100000000, execute [C8-2], append F1 at *G,* then execute [C9](recursion 2). Continue to execute [C1], get F, goes into [C2], generate code 000100000001, execute [C2-1], and append F at the list of its father F1. Execute [C2-2](recursion 3), execute [C1], then return to [C2-2](recursion 3), and continue return to C[9](recursion 2). Execute [C8-3] to generate code 0002, execute [C10](recursion 5), and then return to [C10](recursion 5). The PAD generated is as shown in figure 14.

When the execution is completed, we can get a PAD from Figure 5, which has the same semantics with figure 4.

After testing, the other 11 kinds of structures can be converted into PAD with the semantic equivalence, each recursion can be called and returned correctly.

VII. Generation of code from PAD

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.

```
(1)  : /*****************************************************
(2)  : Function: Generate program code from PAD.
(3)  : Input: PAD.
(4)  : Output: Program code.
(5)  : *****************************************************/
(6)  : String TempCode;  /*When pass it into a function, all the operation to it whith that function will effect the original value, as it is
(7)  : address pass*/
(8)  : CodeGenerate(beginBlock, TempCode);
(9)  : PrintAndFormatCode(TempCode);   /*output and format the code*/
(10) : CodeGenerate(Block CurrentBlock, String CurrentCode)
(11) : {
(12) :       if(CurrentBlock.NodeType is Process)  CurrentCode.Append(CurrentBlock.codeBlock);
(13) :       else if(CurrentBlock.NodeType is Selection)
(14) :       {
(15) :
(16) :          Generate branch code as SelectionCode;
(17) :          for every SubNode in CurrentBlock.SubNodeList do
(18) :            {
(19) :               Generate branch code branchCode;
(20) :               String branchBody;
(21) :                  CodeGenerate(SubNode, branchBody);
(22) :                  Insert branchBody into branchCode;
(23) :                  SelectionCode.Append(branchCode);
(24) :       }
(25) :       CurrentBlock.Append(SelectionCode);
(26) :       }
(27) :       else if(CurrentBlock.NodeType is HeadPtr)  /*If type is HeadPtr, then process the sub-nodes*/
(28) :       {
(29) :          for every SubNode in CurrentBlock.SubNodeList do CodeGenerate(SubNode, CurrentCode);
(30) :       }
(31) :       else if(CurrentBlock.NodeType is Loop)
(32) :       {
(33) :          Generate loop code as loopCode;
(34) :          String loopBody;
(35) :          for every SubNode in CurrentBlock.SubNodeList do CodeGenerate(SubNode, loopBody);
(36) :          insert loopBody into loopCode;
(37) :          CurrentBlock.Append(loopCode);
(38) :       }
(39) :       else return;
(40) : }
```

As the program code in *TempCode* is unformatted, so a third part tool should be called to format that file when they are written into a text file, here we used *Codeblocks* to do that.

VIII. Resource selection model

We developed a integrated develop platform based on Eclipse platform and Graphical Editor Framework (GEF), including flowchart modeling and automatic generation of code, to verify the effectiveness of the proposed algorithms. The overall system interface is as shown in Figure 15.

We construct a model to test various complex nested structures, including the case of sub-flowchart nest. As shown in figures 15, there are totally five structures: switch, for, while, do-while, and if-else. The outer is case, in its first branch we create two sub-flowchart, the type of automatic code generation is *source-block* (all the code generated will be put in the original place) and *function-call* (all the code generated will be put in function, in the original place there will be put a function call statement).

The left figure in figure 16 is a sub-flowchart of *source-block* type (while nests while), the right one is *function-call* (do-while nests do-while).

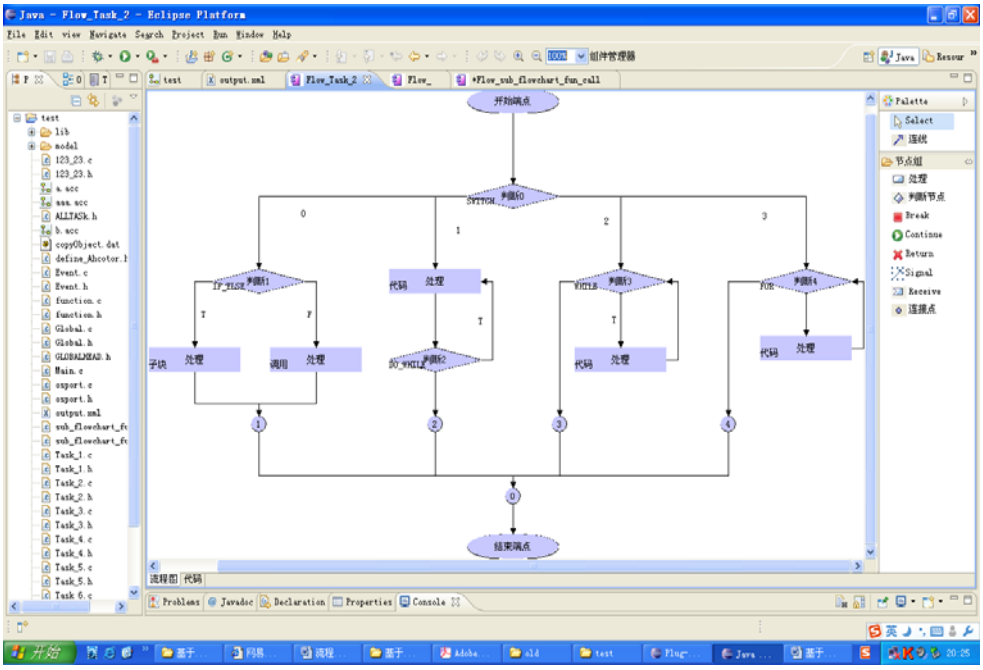

Figure 15. Overall interface of integrated development platform

The code is automatically generated as shown below.

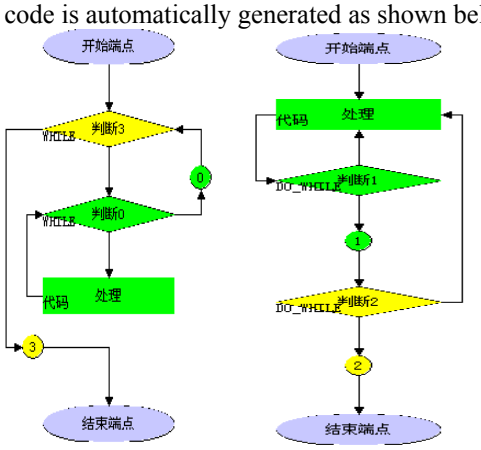

Figure 16. Sub-flowcharts nested

```
/*Task_1.c*/
#include "Task_2.h"
#include "sub_flowchart_fun_call.h"
void Task_2()
{
   unsigned int inner_var;
       unsigned int inner_var2 = 44;
       int i=10;
   switch ( inner_var )
   {
   case 3 :
   {
      for ( inner_var = 0 ; inner_var < 10 ; inner_var ++ )
      {
         printf("this is a for condition");
      }
      break;
   }
   case 2 :
   {
      while ( inner_var < 0 )
      {
         printf("this is a while condition");
      }
      break;
   }
   case 1 :
   {
      do
      {
         printf("this is a do_while condition");
      } while ( inner_var < 0 );
      break;
   }
   case 0 :
   {
      if ( inner_var2 > 0 )
      {
         while (1) /* Generated by sub-flowchart, the type is source-block */
         {
            while (i--)
            {
               printf("this is a sub-flowchart");
            }
         }
      }
      else
      {
         sub_flowchart_fun_call();/*Generated by sub-flowchart, the type is function-call */
      }
      break;
   }
   }
                                    }
```

```
/*sub_flowchart_fun_call.h*/
#ifndef sub_flowchart_fun_call_h
#define sub_flowchart_fun_call_h
#include "GLOBALHEAD.h"
void sub_flowchart_fun_call ( );
#endif
```

```
/* sub_flowchart_fun_call.c*/
#include "sub_flowchart_fun_call.h"
void sub_flowchart_fun_call ( )
{
   int i=10;
   do
   {
      do
      {
         printf("this is sub_flowchart function call");
      } while (i--);
   } while (1);
}
```

IX. CONCLUSIONS

We proposed a structure identification algorithm for structured flowchart. The effectiveness of the proposed

algorithm is checked using an exhaustive method, i.e., twelve structures can be identified, then a 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 a integrated development platform, we develop a weapon system based on the platform to verify the effectiveness of the proposed algorithm.

REFERENCES

[1] Zhang Tian, Zhang Yan, Yu Xiao-Feng etal. MDA Based Design Patterns Modeling and Model Transformation. Journal of Software. 2008,19(9):2203-2217

[2] L V Rui-feng, WANG Gang, WEN Xiao-xian etal. Process model ing method of calculation independent model level based on MDA. Computer Integrated Manufacturing Systems, 2008,14(5):868-874

[3] S. Needham. OMG Unified Modeling Language Specification. Object Management Group. 2003: 275-293

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

[5] Zhao Zhikun, Sheng Qiujian, Shi Zhongzhi. An execution semantics of UML activity view for workflow modeling. Journal of Computer Research and Development. 2005: 300-307

[6] R. Eshuis, R. Wieringal. A formal semantics for UML activity diagrams. University of Twente, Tech Rep. 2001: 201-204

[7] Jiang Hui, Lin Dong, Xie Xiren. The formal semantics of UML state machine. Journal of Software. 2002: 2244-2250

[8] Y. Futamura, T. Kawai, H. Horikoshi etal. Development of computer programs by problem analysis Diagram(PAD). International Conference on Software Engineering archive Proceedings of the 5th international conference on Software engineering. San Diego, California, United States, 1981: 325-332

[9] Kei Kato, Toyohide Watanabe. Structure-Based Categorization of Programs to Enable Awareness About Programming Skills. Lecture Notes in Computer Science,2006, Volume 4253/2006:827-834

[10] D. Harel. Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming. 2007

[11] SHEN Jian-Le, WANG Lin-Zhang, LI Xuan-Dong etal. An Approach to Generate Scenario Test Cases Based on UML Sequence Diagrams. Computer Science, 2004,31(8):1-6

[12] Thomas J. Ostrand, Marc J. Balcer. The Category-Partition Method for Specifying and Generating Functional Tests. Communication of ACM. 2006: 31-67

[13] S. Raman, N. Sivashankar, W. Stuart. HIL Simulators for Powertrain Control System Software Development. American Controls Conference. 2009 23-32

[14] Hemlata Dakhore, Anjali Mahajan. Generation of C-Code Using XML Parser: http://www.rimtengg.com/iscet/ proceedings/ pdfs/advcomp/149.pdf

[15] Martin C. Carlisle, Terry A. Wilson, Jeffrey W. Humphries, etal. Raptor: introducing programming to non-majors with flowcharts. Journal of Computing Sciences in Colleges. 2004,19(4) 1-6

[16] Tia Watts . The SFC editor: a graphical tool for algorithm development. J. Comput. Small Coll.2004, 20(2), 73-85

[17] Kanis Charntaweekhun, Somkiat Wangsiripitak. Visual Programming using Flowchart. ISCIT1 2006, IEEE Computer World: 1-4

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

[19] James F. Gimpel. Contour: a method of preparing structured flowcharts. ACM SIGPLAN Notices. 1980,15(10):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 grid computing and embedded computing.

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 grid computing etc