International Journal of Signal Processing, Image Processing and Pattern Recognition
Vol.9, No.6 (2016), pp.133-148
http://dx.doi.org/10.14257/ijsip.2016.9.6.12

Research on Integration Algorithm of Global Function Call Path Based on Module Path

Pan Lu, Mu Yong-min and Yang Zhi-jia

Open Computer System Laboratory, Beijing Information Science & Technology University, Beijing 100101, China

Abstract

The coverage test method based on function call path is according to the relationship between functions, not only effectively reduce the number of test case, but also ensure the adequacy of the test coverage, how to accurately extract these paths is the key problem of this test method. Paper proposed that for the single function of C program, analysis and extraction the control flow graph and local function call information stored on a particular data structure, design a algorithm to analysis the local function call information, starting from the main function, expanse the function call information layer by layer, and then get the global function call relationship of the program. Experimental results show that this method can accurately obtain the local function call information and global function call path, accurately restore the function call relationship of the program.

***Keywords**: test coverage; function call path; C program; control flow graph*

1. Introduction

Software testing is an important stage in the software development process. It is the key to ensure the software quality and improve the reliability of the software [1]. Considering whether care about the internal structure and concrete realization of the software, software testing can be divided into white box testing, black box testing and white box test. The white box testing can efficiently find and solve human causes errors in the software [2]. Basis path testing is first proposed by Tom McCabe, which is a white box testing technique, the method is mainly based on control flow graph of program, through the analysis of the control structure's annular complexity, so as to extract the basic path test set, design the corresponding test cases [3-4]. Compared with the basic path testing method, the path testing method which based on function calls is expanse the analysis size to the function, a single function as a processing unit, based on control flow information to analysis the logical relationship between functions in the source code, thus obtaining a function call path set as the test path set [5]. Under the premise of single function completed the unit testing, this method can not only optimize the test paths effectively, but also can ensure the test completely. The function call relationship reflects the dependence relationship between functions in the software system, in understanding and analysis procedures, test and maintenance of software, compiler optimization and many other field of software engineering has a wide range of applications [6-8]. At the same time, the thought of based on function call path is also provides a new idea for the research of defect location [9], the defect location which take the function as the processing unit can be first located in function, in this way, to a great extent narrowing the scope of detecting, have a great help to improve the locating efficiency.

As a basic test method, the key of path testing method is to obtain the test path set accurately. According to the different extraction methods, the function call path is divided into two categories, namely static path [10] and dynamic path [10]. The static path refers to the function call path obtained by static analysis of source code. With the wide application of the function call path testing method, for different programing languages there have a complete method for static path extraction. For example, software testing tools Regression Test For C/C++ is already available through static analysis of C/C++ source code, to obtain static path and the function call graph which contain control flow [11-15]. Using the support of Soot which provide optimization and analysis that within process or between processes, the control flow graph of each function in Java source code can be obtained easily, then the static path also can be extracted [16-17]. Dynamic path refers to obtain the function execution sequence in the dynamic execution of program. The more commonly used processing method is dynamic instrumentation [18-19]. The method is based on guaranteeing the program logic integrity, insert a detection program (also known as probe function) in the specific parts of the tested program [18]. In order to determine the instrument position, ensure the accuracy of the instrument and does not generate redundant information, the lexical analysis and syntax analysis of source code is needed, insert instrument at function call processing. The method can effectively obtain the dynamic path of program. But the integrity of the obtained path depends heavily on test cases. The tools of Valgrind and Gprof [20-21] are based on dynamic path obtained method.

Yang Zhijia in literature [22] proposed a new idea in the year of 2014, namely extract the function call path based on the control flow graph. This method can accurately obtain the intermediate code with gcc compiler, then obtain the control flow information based on the control flow information which contained in intermediate code, finally get the function call information and extract the function call path. But as the key research point of this paper is not in the function call path, therefore this paper only extracted a single function's local call information, did not get the global function call path.

Based on the research of literature [22], this paper proposes a method of extracting global function call path based on the local function call path. The premise of using this method has two, one is that the local function call relationship can be obtained accurately, the other is there have a simple and feasible algorithm to extract the global function call relationship from local function call relationship. In the above premise, the global function call path of source code can be obtained accurately.

2. Related Concepts

The definition of function calls are as follows:

Definition 2.1 local function call path set, namely a function name sequence which obtained by a function's internal logic and function call information. Expressed as $G_{f\text{-}i}$={F_0, F_1, F_2,... F_n}, among them, $G_{f\text{-}i}$ refers to this function call path set is belongs to the function f, F_j is a function name, function f call the function F_j, the adjacent relationship between F_j and F_{j+1} only expressed their order of execution, don't denote F_j call the function F_{j+1}.

Definition 2.2 function call path set: also known as global function call path set. It refers to a function name sequence from the entrance point to the exit point which according to the source code and obtained by function calls. Expressed as G_i={ F_0, F_1, F_2,... F_n}, among them, F_j is a function name, the adjacent relation of F_j and F_{j+1}, expressed F_j calls F_{j+1} or F_j and F_{j+1} are executed in sequence.

Definition 2.3 local function call relationship graph: the call relationship inside a single function can be described by a directed graph, the local call graph of function f can be expressed as G_f=<V,E>. V=[f1,f2,…,fn] is nodes set, each node f_j represents a function, E={(x,y) | x,y∈V} is a set of arcs, expressed the correlation between function x and y, if x is the function f, it’s indicated that function x calls function y; if x is not, it's indicates that y will be executed sequentially after function x.

Definition 2.4 function call relationship graph: also known as global function call graph. The call relation between functions can also use a directed graph to described, expressed as G=<V,E>[22]. V=[f1,f2,…,fn] is nodes set, each node f_j represents a function, E={(x,y) | x,y∈V} is a set of arcs, expressed the correlation between function x and y, which is that x call y or x and y executed sequentially.

Program fragments shown as follows in Figure 1:

```
1.   int add(int a,int b)
2.   {
3.      return(a+b);
4.   }
5.   int minus(int a,int b)
6.   {
7.      return (a-b);
8.   }
9.   int operate(int a,int b)
10.  {
11.     if(a<b)
12.          add(a,b);
13.     else
14.          minus(a,b);
15.  }
16.  void fun()
17.  {
18.     printf("operate is ok.");
19.  }
20.  void main ()
21.  {
22.     int a=0;
23.     int b=5;
24.     scanf("%d",&a);
25.     operate(a,b);
26.     fun();
27.  }
```

Figure 1. Examples of Program Fragments

There are five functions in the program fragments: *main,operate,fun,add,minus*. (The standard library functions will not consider.) According to the call relation between functions, the local function call path set of function *main* is G_{main-1}={*operate,fun*}, of function *operate* is $G_{operate-1}$={*add*}, $G_{operate-2}$={*minus*}, which in the internal of function *fun,add* and *minus* do not contain function call information. The global function call path set of this program is G_1={*main,operate,add,fun*},G_2={*main,operate,minus,fun*}. A path set corresponding to a function call path, combining with the above definitions, the local function call path of function *main* is P_1: *main*→*operate*→*fun*. The local function call path of Function *operate* are P_1: *operate*→*add*,P_2: *operate*→*minus*. The global function call path of program are P_1: *main*→*operate*→*add*→*fun*,P_2: *main*→*operate*→*minus*→*fun*. When the input test data of program is a<5, executing the 12th lines' code, calling the function *add* to get the function call path P_1. If Entering a>=5, executing the 14th lines' code, calling the function *minus* to get the function call path P_2. The local function call relationship graph of function *main* is G_{main}=<[*main,operate,fun*], {(*main,operate*), (*operate,fun*)}>. The local function call relationship graph of function *operate* is $G_{operate}$=<[*operate,add,minus*], {(*operate,add*), (*operate,minus*)}>. Global function call graph of the program is the integration of local function call graph, it can be expressed as

G=<[*main, operate, add, minus, fun*], {(*main, operate*), (*operate, add*), (*operate,minus*), (*add,fun*), (*minus,fun*)}>. Adding node <*begin*> and <*end*> when drawing the function call relationship graph, as shown in Figure 2. The node <*begin*> and <*end*>, which in local function call relationship graph respectively represent the beginning and end execution of this function. In global function call relationship graph the <*begin*> and <*end*> node respectively represent the beginning and end execution of *main* function.

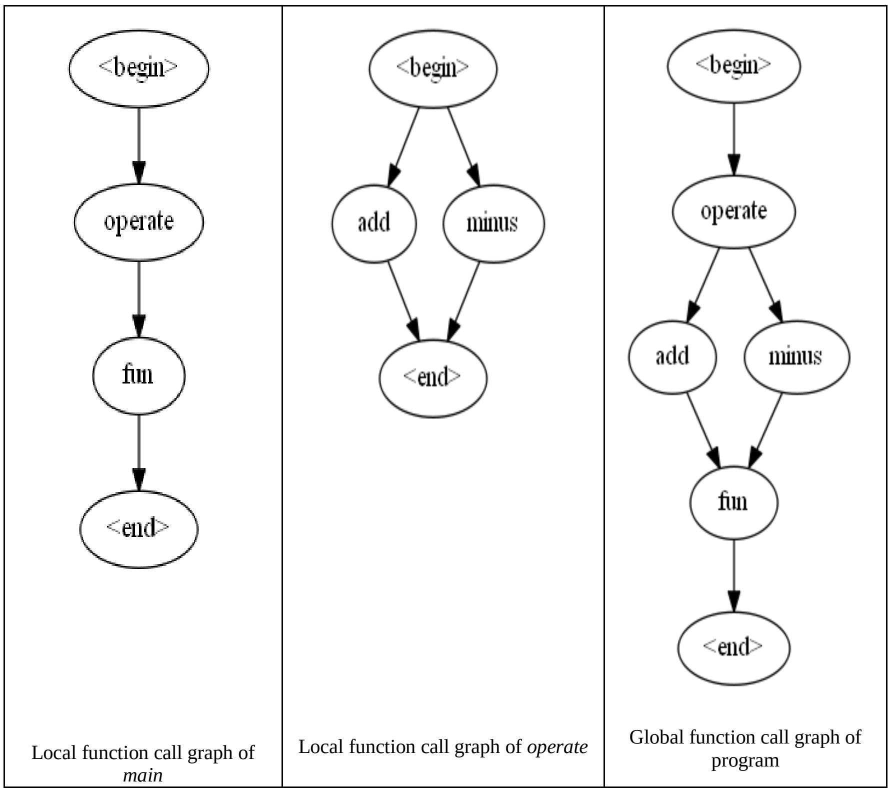

<begin>
operate
fun
<end>
Local function call graph of *main*
<begin>
add
minus
<end>
Local function call graph of *operate*
<begin>
operate
add
minus
fun
<end>
Global function call graph of program

Figure 2. Function Call Path Graph

3. Local Function Call Information Extraction

3.1 Control Flow Graph Extraction

Control flow graph can be described as a tuple<N, E, B, D>, which N denotes the nodes set, each node represents a basic code block, E∈N×N denotes the edges set, each edge represents a basic block of control flow, B∈N represents the beginning node of each edge, D∈N represents the end node of each edge [23]. The extraction of control flow graph is mainly based on gcc. gcc is a powerful C language editor, which contains a number of options for controlling the compile and link process, among them, *–fdump -tree* option can get gcc pre-coding information of the source code, select the appropriate sub-options can generate intermediate code which like control flow graph [22]. Using the way of pattern: action to static analyze the intermediate code, combining with the format of code, designing algorithms to match function declaration, basic code block, jump statement and function call. During the extraction control flow graph, regard function as the basic processing unit, therefore, when matching the function declaration is means to open a processing unit. Control flow information extracted from the intermediate code are stored in the Json file with a specific format. Json is a lightweight data interchange format, It's written in the format of key: value pairs, where value can be numeric, string or array [22]. When key value is *functions*, the corresponding value is stored an array of all functions' control flow information, a single element *function* in the array is a processing unit which contains two elements *function_name* and *tokens*. Json file uses key:value to store the control flow information of the function in the layer by layer, its file structure tree is shown in Figure 3, the whole tree structure represents a processing unit, the leaf node in the tree is the explanation of parent node.

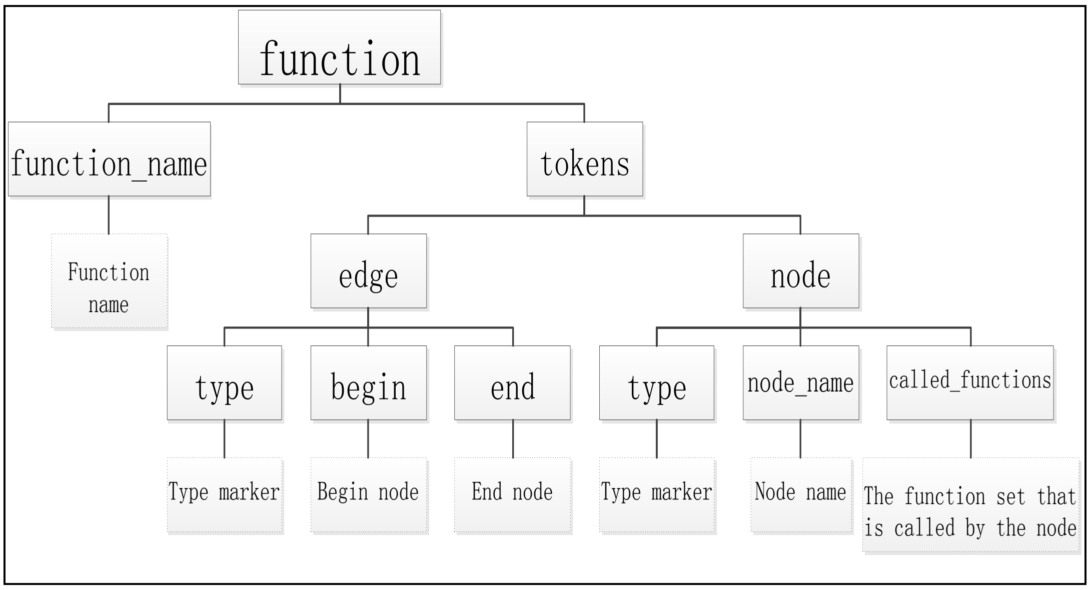
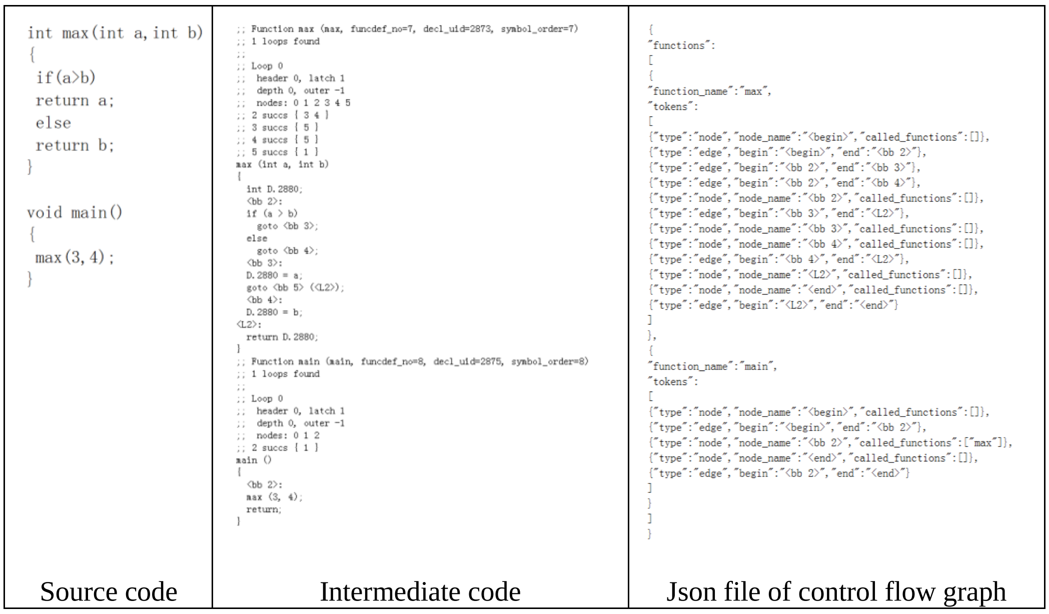


Figure 3. The File Structure Tree of Control Flow Graph

Combined with the definition of the Json file structure, the static analysis results of the intermediate code are in turn to fill in the corresponding position in the Json file. An analysis example that contains a number of functions is shown in Figure 4.



```
int max(int a, int b)
{
 if(a>b)
 return a;
 else
 return b;
}

void main()
{
 max(3,4);
}
```

Source code

```
;; Function max (max, funcdef_no=7, decl_uid=2873, symbol_order=7)
;; 1 loops found
;;
;; Loop 0
;;  header 0, latch 1
;;  depth 0, outer -1
;;  nodes: 0 1 2 3 4 5
;; 2 succs { 3 4 }
;; 3 succs { 5 }
;; 4 succs { 5 }
;; 5 succs { 1 }
max (int a, int b)
{
  int D.2880;
  <bb 2>:
  if (a > b)
    goto <bb 3>;
  else
    goto <bb 4>;
  <bb 3>:
  D.2880 = a;
  goto <bb 5> (<L2>);
  <bb 4>:
  D.2880 = b;
<L2>:
  return D.2880;
}
;; Function main (main, funcdef_no=8, decl_uid=2875, symbol_order=8)
;; 1 loops found
;;
;; Loop 0
;;  header 0, latch 1
;;  depth 0, outer -1
;;  nodes: 0 1 2
;; 2 succs { 1 }
main ()
{
  <bb 2>:
  max (3, 4);
  return;
}
```

Intermediate code

```
{
"functions":
[
{
"function_name":"max",
"tokens":
[
{"type":"node","node_name":"<begin>","called_functions":[]},
{"type":"edge","begin":"<begin>","end":"<bb 2>"},
{"type":"edge","begin":"<bb 2>","end":"<bb 3>"},
{"type":"edge","begin":"<bb 2>","end":"<bb 4>"},
{"type":"node","node_name":"<bb 2>","called_functions":[]},
{"type":"edge","begin":"<bb 3>","end":"<L2>"},
{"type":"node","node_name":"<bb 3>","called_functions":[]},
{"type":"node","node_name":"<bb 4>","called_functions":[]},
{"type":"edge","begin":"<bb 4>","end":"<L2>"},
{"type":"node","node_name":"<L2>","called_functions":[]},
{"type":"node","node_name":"<end>","called_functions":[]},
{"type":"edge","begin":"<L2>","end":"<end>"}
]
},
{
"function_name":"main",
"tokens":
[
{"type":"node","node_name":"<begin>","called_functions":[]},
{"type":"edge","begin":"<begin>","end":"<bb 2>"},
{"type":"node","node_name":"<bb 2>","called_functions":["max"]},
{"type":"node","node_name":"<end>","called_functions":[]},
{"type":"edge","begin":"<bb 2>","end":"<end>"}
]
}
]
}
```

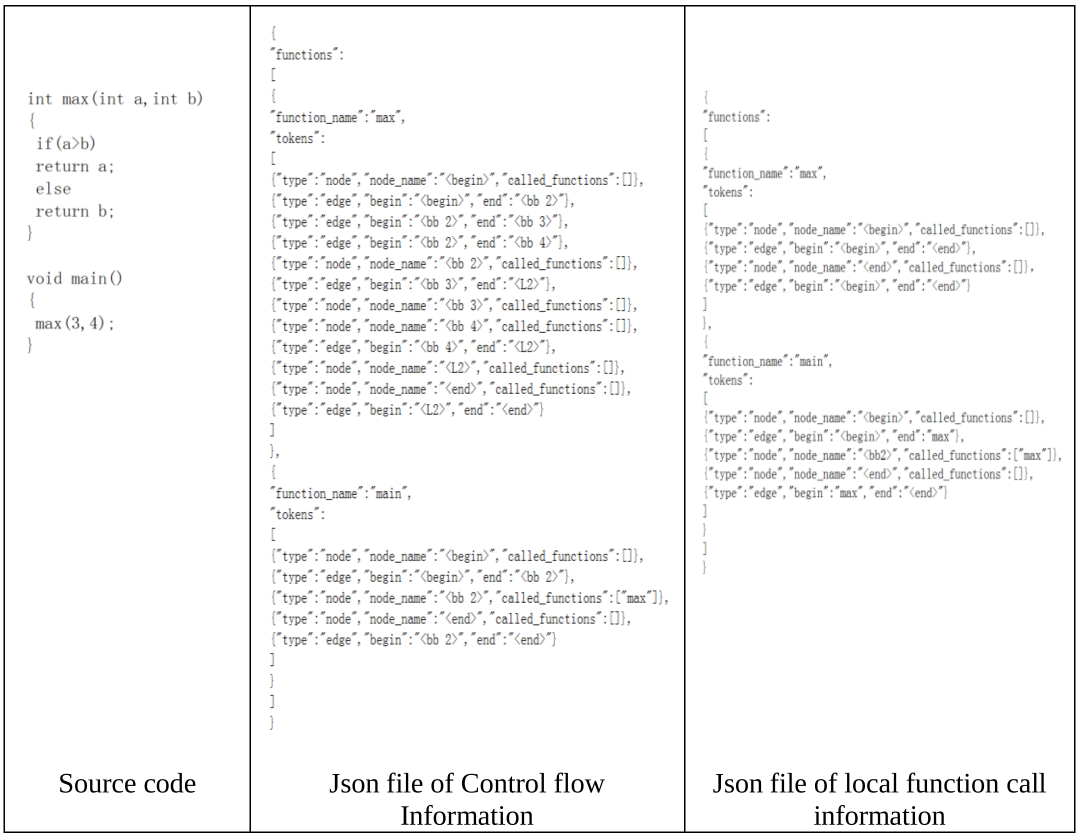
Json file of control flow graph

Figure 4. Storage Format of Control Flow Graph

3.2 Local Function Call Relation Extraction

The control flow graph of function shows the information of function's internal control logic, the local function call graph displays the call information of the internal function. So the meaning of the node in control flow graph and function call graph are also different. A node in the control flow graph represents a basic block of code, but a node represents a function in the function call graph. Therefore, it is important to accurately analyze the function call information in each basic block of code.

In the control flow graph, for each basic block of code, according to the number of internal including function calls can be divided into three categories: no function calls, exactly one function call, the function call number is greater than one [22]. Different situations using different processing methods. If there is no function call information in the basic code block, according to the situation analysis of the node's in degree and out degree, remove the node, and then perform the corresponding "merge" [22] operation. If there is only one function call in the base code block, the name of the node can be replaced by the name of the function. In the third case, we can use the "split"[22] operation. By processing the Json file which stored the control flow information, get the Json file which containing the function call information, the specific analysis of the sample as shown in Figure 5.



```
int max(int a, int b)
{
 if(a>b)
 return a;
 else
 return b;
}

void main()
{
 max(3,4);
}
```

Source code

```
{
"functions":
[
{
"function_name":"max",
"tokens":
[
{"type":"node","node_name":"<begin>","called_functions":[]},
{"type":"edge","begin":"<begin>","end":"<bb 2>"},
{"type":"edge","begin":"<bb 2>","end":"<bb 3>"},
{"type":"edge","begin":"<bb 2>","end":"<bb 4>"},
{"type":"node","node_name":"<bb 2>","called_functions":[]},
{"type":"edge","begin":"<bb 3>","end":"<L2>"},
{"type":"node","node_name":"<bb 3>","called_functions":[]},
{"type":"node","node_name":"<bb 4>","called_functions":[]},
{"type":"edge","begin":"<bb 4>","end":"<L2>"},
{"type":"node","node_name":"<L2>","called_functions":[]},
{"type":"node","node_name":"<end>","called_functions":[]},
{"type":"edge","begin":"<L2>","end":"<end>"}
]
},
{
"function_name":"main",
"tokens":
[
{"type":"node","node_name":"<begin>","called_functions":[]},
{"type":"edge","begin":"<begin>","end":"<bb 2>"},
{"type":"node","node_name":"<bb 2>","called_functions":["max"]},
{"type":"node","node_name":"<end>","called_functions":[]},
{"type":"edge","begin":"<bb 2>","end":"<end>"}
]
}
]
}
```

Json file of Control flow Information

```
{
"functions":
[
{
"function_name":"max",
"tokens":
[
{"type":"node","node_name":"<begin>","called_functions":[]},
{"type":"edge","begin":"<begin>","end":"<end>"},
{"type":"node","node_name":"<end>","called_functions":[]},
{"type":"edge","begin":"<begin>","end":"<end>"}
]
},
{
"function_name":"main",
"tokens":
[
{"type":"node","node_name":"<begin>","called_functions":[]},
{"type":"edge","begin":"<begin>","end":"max"},
{"type":"node","node_name":"<bb2>","called_functions":["max"]},
{"type":"node","node_name":"<end>","called_functions":[]},
{"type":"edge","begin":"max","end":"<end>"}
]
}
]
}
```

Json file of local function call information

Figure 5. Storage Format of the Local Function Call Information

4. Global Function Call Relation Extraction

In the C language program, *main* called the main function, is the entrance of the program. The remaining functions are called by the *main* function or other general functions. Therefore, in order to obtain the global function call graph, function *main* can be regarded as a breakthrough, then gradually acquire the global call graph step by step.

Local function call graph of function *main* can be regarded as the zeroth level expanded view of the function call graph, and then extract total functions which called by the function *main*, analyze this functions one by one, and divide them into two categories according to whether they contain any other function call information. For the function which not contain other function call information, don't need to deal with. If the function contain other function call information, it is necessary to further processing this kind of functions, namely adding these functions' local call function information to the global function call information, getting the one level expanded view of function call graph. Using this approach step by step, finally get the global function call graph. The complexity of the program is different, the function call information is also different, so the expansion of the series is not the same, in order to be flexible master, it is necessary to make some adjustments to the Json file which stored the local function call information, add an *open* attribute to the node's property, which is used to label the function call information of this node whether opened. The tree structure of a single processing unit in the modified Json file as shown in Figure 6.

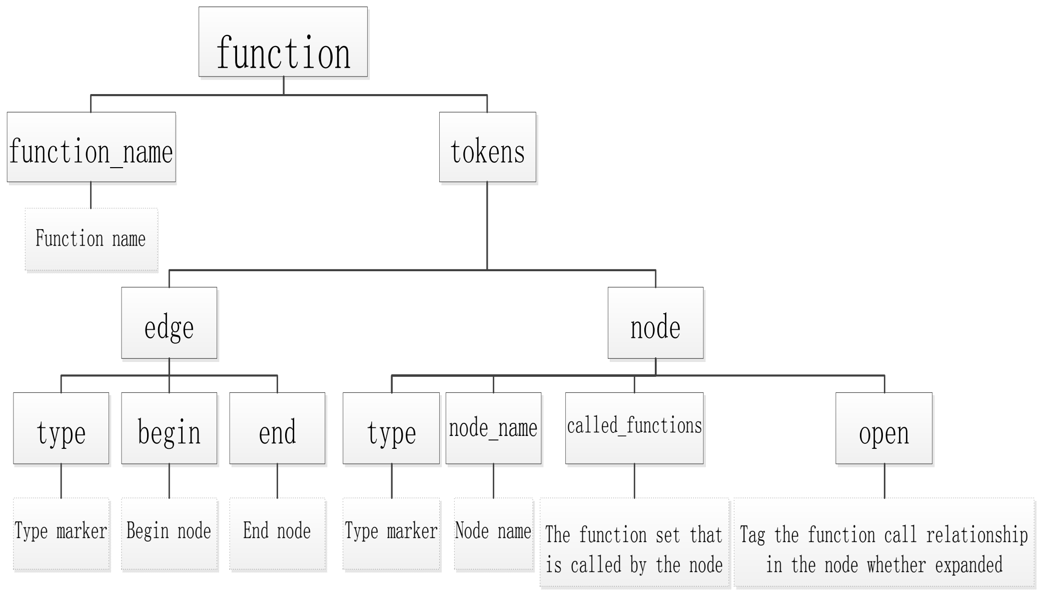
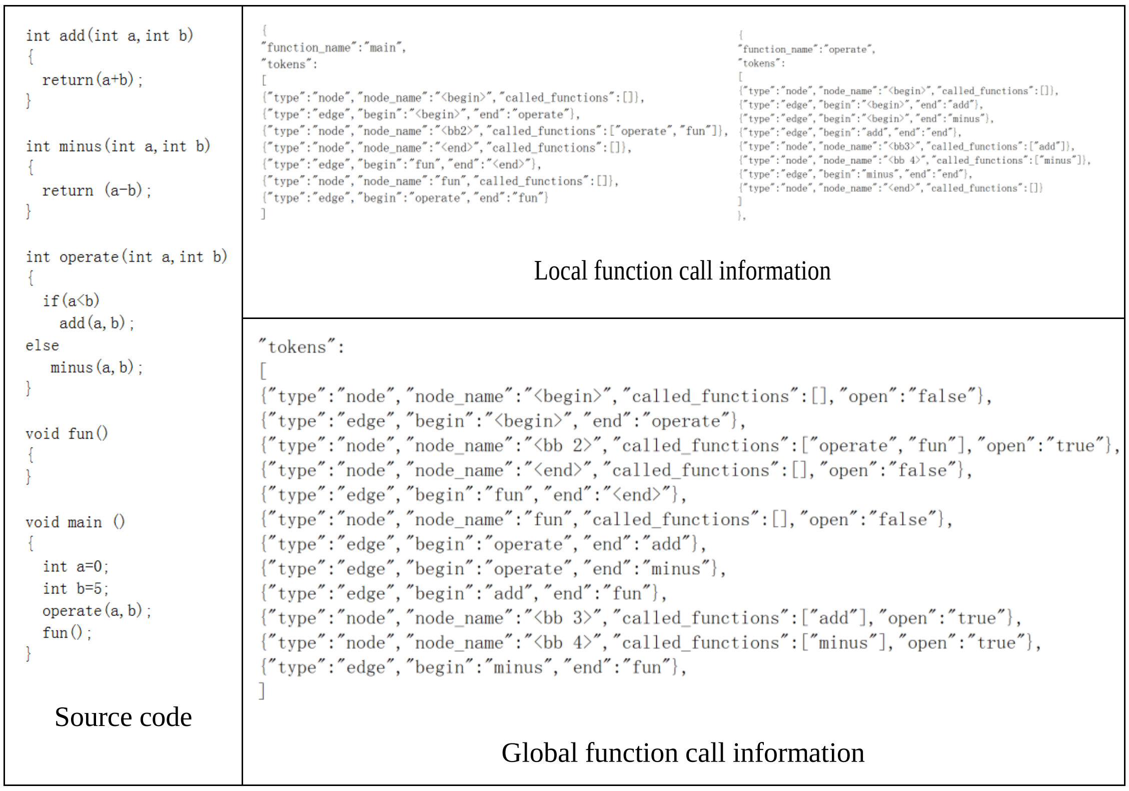


Figure 6. The File Structure Tree of the Global Function Call Graph

The core idea of extracting global function call relation based on the local function call relation is starting with *main* function, getting the local function call information according to the call depth step by step, then add them to the global function call information after processed by the algorithm, when all the functions within the call relationship have been launched, global function call relation can be obtained, analysis of the sample as shown in Figure 7.



```
int add(int a, int b)
{
   return(a+b);
}

int minus(int a, int b)
{
   return (a-b);
}

int operate(int a, int b)
{
   if(a<b)
      add(a,b);
else
     minus(a,b);
}

void fun()
{
}

void main ()
{
   int a=0;
   int b=5;
   operate(a,b);
   fun();
}
```

Source code

```
{
"function_name":"main",
"tokens":
[
{"type":"node","node_name":"<begin>","called_functions":[]},
{"type":"edge","begin":"<begin>","end":"operate"},
{"type":"node","node_name":"<bb2>","called_functions":["operate","fun"]},
{"type":"node","node_name":"<end>","called_functions":[]},
{"type":"edge","begin":"fun","end":"<end>"},
{"type":"node","node_name":"fun","called_functions":[]},
{"type":"edge","begin":"operate","end":"fun"}
]
```

```
{
"function_name":"operate",
"tokens":
[
{"type":"node","node_name":"<begin>","called_functions":[]},
{"type":"edge","begin":"<begin>","end":"add"},
{"type":"edge","begin":"<begin>","end":"minus"},
{"type":"edge","begin":"add","end":"end"},
{"type":"node","node_name":"<bb3>","called_functions":["add"]},
{"type":"node","node_name":"<bb 4>","called_functions":["minus"]},
{"type":"edge","begin":"minus","end":"end"},
{"type":"node","node_name":"<end>","called_functions":[]}
]
},
```

Local function call information

```
"tokens":
[
{"type":"node","node_name":"<begin>","called_functions":[],"open":"false"},
{"type":"edge","begin":"<begin>","end":"operate"},
{"type":"node","node_name":"<bb 2>","called_functions":["operate","fun"],"open":"true"},
{"type":"node","node_name":"<end>","called_functions":[],"open":"false"},
{"type":"edge","begin":"fun","end":"<end>"},
{"type":"node","node_name":"fun","called_functions":[],"open":"false"},
{"type":"edge","begin":"operate","end":"add"},
{"type":"edge","begin":"operate","end":"minus"},
{"type":"edge","begin":"add","end":"fun"},
{"type":"node","node_name":"<bb 3>","called_functions":["add"],"open":"true"},
{"type":"node","node_name":"<bb 4>","called_functions":["minus"],"open":"true"},
{"type":"edge","begin":"minus","end":"fun"},
]
```

Global function call information

Figure 7. Storage Format of Global Function Call Information

The conversion steps from the local function call graph to the global function call graph are as follows:

First. Add the *open* attribute to the node of the local function call information, and the initial value is set false.

Second. Starting from the main function call information, traversing the corresponding nodes tokens of function *main*, if the node's *called_functions* array is not empty and *open* attribute value is false, then traversal the function in *called_functions* array, for each function using a dictionary *dict* to store the out edge information, the assignment form is *dict[i]=j*. Dict is stored data by key-value pair. Key *i* in *dict* is function name, the node set which function *i* pointed to in the function call graph is stored in *j*. After save an out edge information of *i* , delete this edge's information from the global function call information, at the same time set the *open* attribute value of this node to true.

Third. For each function node *k* in *j*, make different processing according to the complexity of the *i* function's call relationship, then add the corresponding node and edge information to the global function call graph.

In view of the above analysis, the core algorithm of extracting global function call graph from the local function call graph is given as follows. The input of this algorithm is the data file that containing local function call information, and the output is the data file that contains the global function call information. In the algorithm, starting with the local function call relationship of *main* function, through the analysis of the node in the local function call graph, local function call information is extracted step by step to the Global function call information. Due to the expansion series can be controlled by one condition, the algorithm of only expand at one level is shown below.

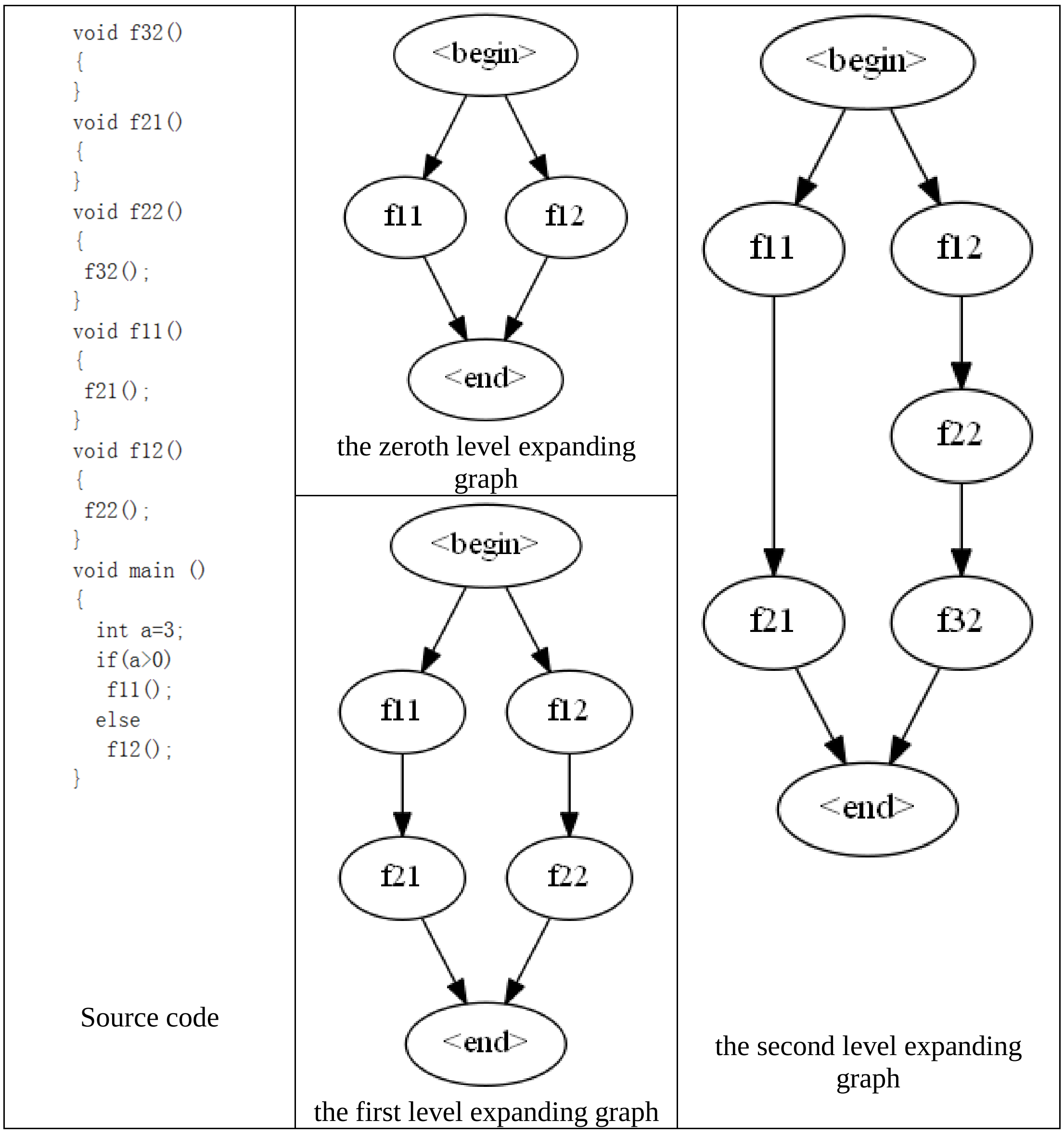
Algorithm 1 PCFG2GCFG

Input：PCFG (data)

Output：GCFG (main_tokens)

```
1:  tokens=get_tokens(data,'main')
2:      main_tokens=copy.deepcopy(tokens)
3:  for token in tokens:
4:      if token['type']=='node' and len(token['called_functions'])>0 and token['open']=='false':
5:          find this token in main_tokens and set token['open']= 'true'
6:          get called_funs=token['called_functions']
7:          for i in called_funs:
8:              copy_tokens=copy.deepcopy(main_tokens)
9:              for token in copy_tokens:
10:                 get function call relationship: dict[i]=j
11:         for i,j in dict.items():
12:             fun_tokens=get_tokens(data,i)
13:             for k in j:
14:                 if fun_tokens==None:
15:                     add edge in main_tokens: i→k
16:                 else:
17:                     for fun_token in for_tokens:
18:                         if fun_token['type']== 'node':
19:                             if len(fun_token['called_functions'])>0:
20:                                 append node: fun_token
21:                             else:
22:                                 pass
23:                         else:
24:                             if fun_token['begin']== '<begin>' and fun_token['end']== '<end>' :
25:                                 add edge in main_tokens: i→j
26:                             elif fun_token['begin']== '<begin>' :
27:                                 add edge in main_tokens: i→fun_token['end']
28:                             elif fun_token['end']== '<end>' :
29:                                 add edge in main_tokens: fun_token['begin']→j
30:                             else:
31:                                 add edge in main_tokens: fun_token['begin']→fun_token['end']
```

Every time the PCFG2GCFG algorithm is executed the function call relationship can be expanded one level, the local function call graph of function *main* is used as the zeroth level expansion graph, and then expanded step by step until all the local function calls are extracted. The specific example is shown in Figure 8.



```
void f32()
{
}
void f21()
{
}
void f22()
{
 f32();
}
void f11()
{
 f21();
}
void f12()
{
 f22();
}
void main ()
{
   int a=3;
   if(a>0)
     f11();
   else
     f12();
}
```

Source code
<begin>
f11
f12
<end>
the zeroth level expanding graph
<begin>
f11
f12
f21
f22
<end>
the first level expanding graph
<begin>
f11
f12
f22
f21
f32
<end>
the second level expanding graph

Figure 8. Function Call Relation Expansion Graph

In the source code shown in Figure 8, function *main* directly call the function *f11* and *f12*, so the zeroth level expanding graph of function call relation only shows *f11* and *f12*. Then obtain the local function call information of *f11* and *f12*, the source code show that *f11* directly call *f21* and *f12* directly call *f22*, so the first level expanding graph of function call information is shown above. In which the local function call relationship of *f11* and *f12* has been expanded, the local function call relationship of the local *f21* and *f22* is not expanded, so further processing can obtain the second level expanding graph of function call information. Due to the function *f21* does not contain any other function call information, it is only expanding the function call relationship of *f22* in the second level expanding graph of function call information. In term of source code, as a result of the function *f32* does not contain any other function call information, so the second level expanding graph of function call information is seem as the global function call graph of source code.

If a number of functions call the same function, the extraction of global function call information is not affected, but the visibility of global function call graph is relatively poor. Although the function call relationship is shown in the graph, but the function call path is not intuitive, on this basis, make a slight adjustment, described the function call relationship in the node, example as shown in Figure 9.

```c
void fun1()
{
}
int add(int a,int b)
{
  fun1();
  return(a+b);
}
int minus(int a,int b)
{
  fun1();
  return (a-b);
}
int operate(int a,int b)
{
    add(a,b);
   minus(a,b);
}
void main ()
{
  int a=0;
  int b=5;
  operate(a,b);
  fun1();
}
```

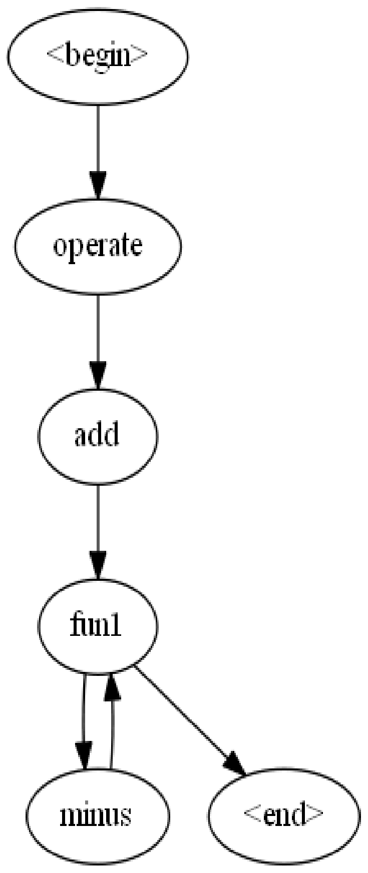
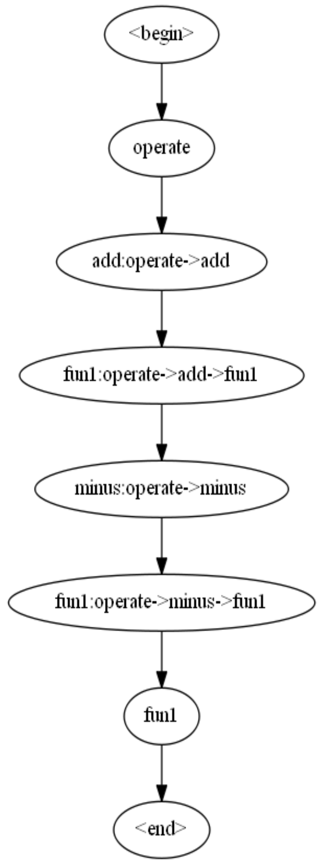
		
Source code	global function call graph 1	global function call graph 2

Figure 9. Comparison of Global Function Call Graph

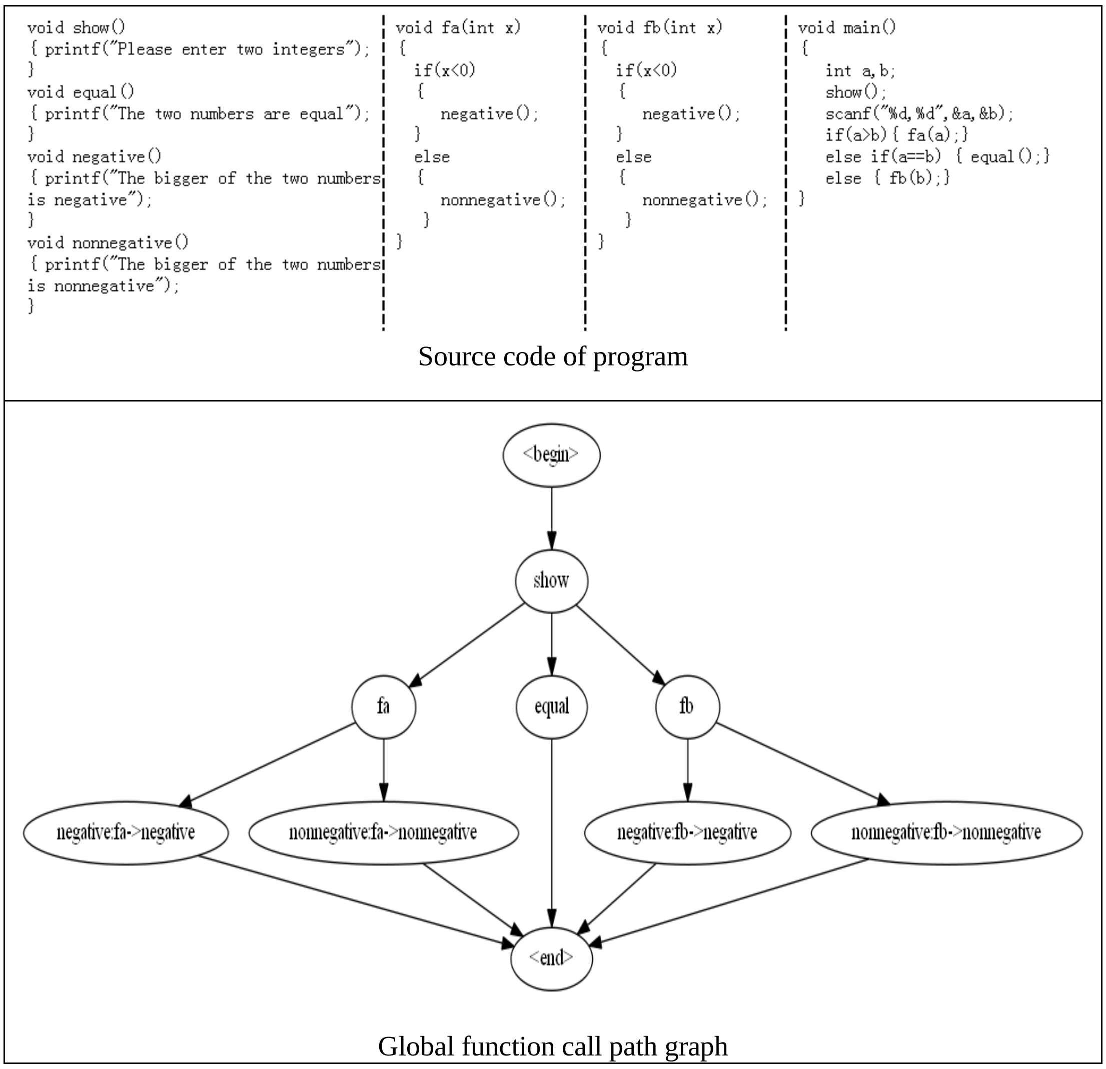
By the source code of example can be known, function *fun1* were called in the three functions *main, add and minus*, global function call path is P: *main→operate→add→fun1 →minus→fun1→fun1*, In the global function call graph 1 shows two paths, P$_1$:*<begin>→ operate →add →fun1 →minus →fun1 →<end>*, P$_2$:*<begin> →operate →add →fun1 → <end>*. The main reason for this deviation is that there is only one node representing *fun1* function, no matter which function calls *fun1* all point to this node. In order to increase the visibility of the function call graph, adjusted display form is shown as global function call graph 2. In the graph the nodes are not directly called by *main* function are added an illustration. Although *fun1* implement three times, but it is being called by different functions, so there are three nodes are used to express *fun1* in order to be distinguished. This form of display may seems complicate, but it can clearly expressed the function call path and the execution relationship between functions.

5. Experiment and Evaluation

Sequence, selection and circulation are the three basic sentence structures of C language, which can be nesting used without limitation, and each of them can contain function call statements. So in the experiment, the method of extracting the global function call path is verified by the way of three kinds of statement structure nested each other. The function call statement in the loop structure is likely to be repeated several times, during the extraction of the function call path, the function call statement in the loop is divided into two cases, namely execution and not execution, the number of repeated execution is not considered.

5.1 Sequential and Selection Nested Program

In the upper part of Figure 10 is the source code of sequential and selection nested program, *main* function used branch structure *if…else if…else* , the value of two input variables are used as branch condition, in the four functions that are called directly by *main* function, *show* function and the other three functions are sequential execution relation, and function *fa*, *equal*, *fb* will not be executed simultaneously, among them, function *fa* and *fb* used branch structure *if... else*. The following part of Figure 10 is the global function call relationship graph that extracted from the experimental program.



```
void show()
{ printf("Please enter two integers");
}
void equal()
{ printf("The two numbers are equal");
}
void negative()
{ printf("The bigger of the two numbers
is negative");
}
void nonnegative()
{ printf("The bigger of the two numbers
is nonnegative");
}
```

```
void fa(int x)
{
  if(x<0)
  {
     negative();
  }
  else
  {
     nonnegative();
   }
}
```

```
void fb(int x)
{
  if(x<0)
  {
     negative();
  }
  else
  {
     nonnegative();
   }
}
```

```
void main()
{
    int a,b;
    show();
    scanf("%d,%d",&a,&b);
    if(a>b){ fa(a);}
    else if(a==b) { equal();}
    else { fb(b);}
}
```

Source code of program

Global function call path graph

Figure 10. Global Function Call Path Extraction from Sequential and Selection Nested Program

According to the single variable's value and the value size relation between the two variables, combined with the source code manual analysis the function call path as shown in Table 1.

Table 1. The Global Function Call Path Analysis of Sequential and Selection Nested Program

Number	The value size relation of variable a and b	The value of a single variable	Function call path
1	a>b	a<0	main→show→fa→negative
2		a>=0	main→show→fa→nonnegative
3	a=b	a、b take any value	main→show→equal
4	a<b	b<0	main→show→fb→negative
5		b>=0	main→show→fb→nonnegative

Comparing the global function call graph shown in Fingure10 and the analysis result of function call path shown in table 1, it is obvious that the extracted function call path and manual analysis results is accordance. It is demonstrated that using global function call path extraction method can accurately obtain the function call path of source code in this case.

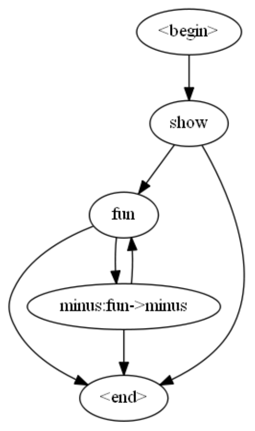
5.2 Sequential and Loop Nested Program

The left side of Figure 11 is an experimental source code of a sequential and loop nested program. Function *main* used *for* loop structure, if meeting the condition of *for* loop, function *show* and *fun* in the *main* is sequential execution, Function *fun* used the *while* loop structure, if meet the condition of the *while* loop, it will performed *minus* function. On the right side of Figure 11 is the global function call graph which extracted from the experimental program.

```
void show()
{ printf("Please enter
two integers");
}
int minus(int t)
{ t--;
}
void fun(int x)
{
  while(x>0)
   { x=minus(x);}
}
```

```
void main()
{
   int a,b;
   show();
   scanf("%d,%d",&a,&b);
   for(int k=a;k<a+b;k++)
   {
     fun(a);
   }
}
```

Source code



Global function call graph

Figure 11. Global Function Call Path Extraction from Sequential and Loop Nested Program

According to the value of two variables, combined with the source code manual analysis the function call path as shown in Table 2.

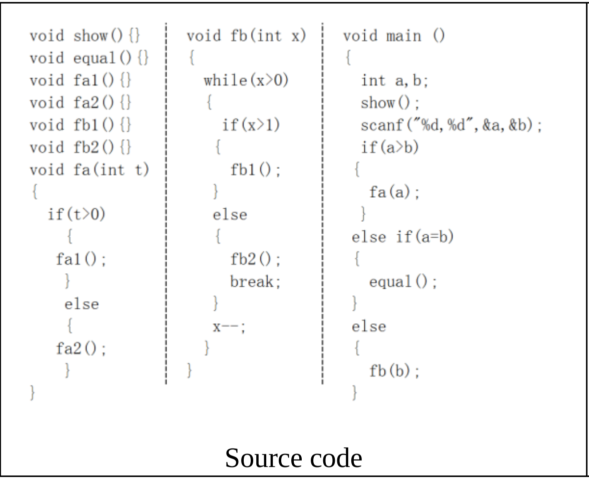
Table 2. The Gobal Function Call Path Analysis of Sequential and Loop Nested Program

Number	The value of b	The value of a	Function call path
1	b>1	a>1	main→show→fun→minus(2~n)→··· →fun→minus(2~n)
2		a=1	main→show→fun→minus→···→fun→minus
3		a<1	main→show→fun→···→fun
4	b=1	a>1	main→show→fun→minus(2~n)
5		a=1	main→show→fun→miuns
6		a<1	main→show→fun
7	b<1	a take any value	main→show

minus (2~n) in Table 2 indicates that the function *minus* performs two times or more than two times. Due to the function call in loop structure only considers the two cases, namely execution and not execution, so in Table 2, the function call path 1 and 2, 3 and 5, 4 and 6 can be merged. The four function call path after merged and the path of global function call graph shown in Figure 11 is accordance, indicating that using global function call path extraction method can accurately obtain the function call path of source code in this case.

5.3 Sequential, Selection, and Loop Nested Program

The left side of Figure 12 is the source code of sequential, selection, and loop nested program, *main* function used branch structure *if…else if…else* , the value of two input variables are used as branch condition, in the four functions that are called directly by main function, *show* function and the other three functions are sequential execution relation, and function *fa, equal, fb* will not be executed simultaneously, among them, function *fa* used branch structure *if... else*, function *fb* used *while* loop structure, and in the loop structure used branch structure *if... else* again. On the right side of Figure 11 is the global function call graph which extracted from the experimental program.

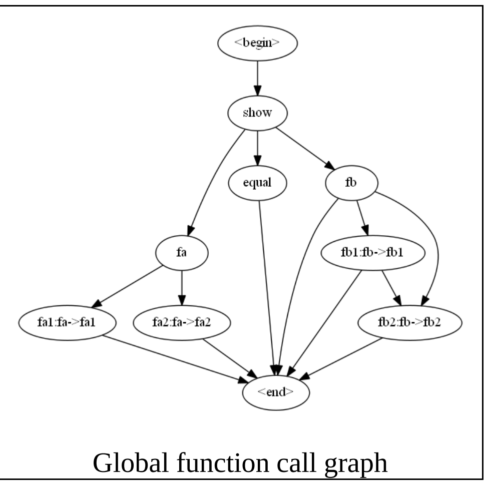


```
void show(){}
void equal(){}
void fa1(){}
void fa2(){}
void fb1(){}
void fb2(){}
void fa(int t)
{
  if(t>0)
    {
   fa1();
    }
    else
    {
   fa2();
    }
}
```

```
void fb(int x)
{
  while(x>0)
  {
    if(x>1)
   {
     fb1();
   }
   else
   {
     fb2();
     break;
   }
   x--;
  }
}
```

```
void main ()
{
  int a,b;
  show();
  scanf("%d,%d",&a,&b);
  if(a>b)
  {
   fa(a);
  }
 else if(a=b)
 {
   equal();
 }
 else
 {
   fb(b);
 }
```

Source code



Global function call graph

Figure 12. Global Function Call Path Extraction from Experimental Program

According to the single variable's value and the value size relation between the two variables, combined with the source code manual analysis the function call path as shown in Table 3.

Table 3. The Global Function Call Path Analysis of Experimental Program

Number	The value size relation of variable a and b	The value of a single variable	Function call path
1	a>b	a>0	main→show→fa→fa1
2		a<=0	main→show→fa→fa2
3	a=b	a、b take any value	main→show→equal
4	a<b	b>2	main→show→fb→fb1(2~n) →fb2
5		b=2	main→show→fb→fb1→fb2
6		0<b<2	main→show→fb→fb2
7		b<=0	main→show→fb

fb1(2~n) in Table 3 indicates that the *fb1* function performs two times or more than two times. As the same, the repeated execution number of functions in the loop structure is not considered, so in Table 3, the function call path 4 and 5 can be merged. Thus we can obtained six function call paths by manual analysis, Comparing with the path of global function call graph which shown in Figure 12, results and expectations are consistent, It is demonstrated that using global function call path extraction method can accurately obtain the function call path of source code in this case.

6. Conclusion

The path coverage method which based on function call information improves the test efficiency and at the same time ensures the adequacy of the test. The key is to obtain the set of testing path, which is the set of the function call path. In this paper, a global function call path integration method based on module path is proposed. First, obtained the local function call relation of the single function based on control flow graph. then, according to the correlation between local function call relation and global function call relation, starting from the local function call relation of local function of function *main*, lunching the local function call relation step by step through the algorithm, then the global function call relation of the program is obtained. Finally, analyzing reachable path from begin point to end point of the global function call relation to obtain the function call path. The experimental data show that this method can effectively extract the function call path. It provides a new way of thinking for the accurate acquisition of path test set based on function call path test coverage method.

References

[1] Z. H .Zhang and Y. M. Mu, “Research of path coverage generation techniques based function call graph”, Acta Electron Sin, vol. 138, **(2010)**, pp. 1808–1811.
[2] I. Jovanović, “Software Testing Methods and Techniques”, [J]. The IPSI BgD Transactions on Internet Research, vol. 30, **(2006)**.
[3] L. X. MEI and Z. L. ZHANG, “A Research on Basis Path Acquire Method for Software Testing”, [J]. Journal of LanZhou JiaoTong University, vol. 1, **(2011)**, pp. 6-9.
[4] A. H. Watson, T. J. McCabe and D. R. Wallace, “Structured testing: A testing methodology using the cyclomatic complexity metric”, [J]. NIST special Publication, vol. 500, no. 235, **(1996)**, pp. 1-114.
[5] Y. M. MU, Y. H. ZHENG and Z. H. ZHANG, “The algorithm of infeasible paths extraction oriented the function calling relationship”, [J]. CHINESE JOURNAL OF ELECTRONICS, vol. 21, no. 2, **(2012)**，pp. 236-240.

[6] D. Dewu, “Complex Network Analysis of Call Graph in Linux Kernel”, [J].Journal of Chizhou University, vol. 26, no. 6, **(2012)**, pp. 1-3.
[7] D. ZHAO, “The Research of call graph construction based on static type anlysis for Java program”, [D].HuNan University, **(2006)**.
[8] H. Shuangling, “Research on C/C++ programs’ function call relations based on static analysis”, [D].University of Science and technology of China, **(2015)**.
[9] X. CHEN, X. L. JU, W. Z. WEN and Q. GU, “Review of Dynamic Fault Localization Approaches Based on Program Spectrum.Journal of Software, vol. 26, no. 2, **(2015)**, pp. 390−412.
[10] Y. MU， Z. JIANG and Z. ZHANG, “Path extraction based on C program instrumentation”, Computer Engineering and Applications, vol.47, no. 1, **(2011)**, pp. 67-69.
[11] S. Baharom and Z. Shukur, “Module documentation based testing using grey-box approach”, In: Proceedings of Information Technology, Kuala Lumpur, **(2008)**, pp. 1–6.
[12] Y. H. Zheng, Y. M. Mu and Z. H. Zhang, “Research on the static function call path generating automatically.”, In: Proceedings of Information Management and Engineering, Chengdu, **(2010)**, pp. 405–409.
[13] Y. M. Mu, Y. H. Zheng and Z. H. Zhang, “The algorithm of infeasible paths extraction oriented the function calling relationship”, Chinese J Electron, vol. 21, **(2012)**, pp. 236–240.
[14] Y. M. MU and M. T. LIU, “Determination of overload uniqueness in C++ based on finite-state machine”, [J]. Application Research of Computers, vol. 31, no. 4, **(2014)**, pp. 1059-1062.
[15] D. F. Liu, Y. M. Mu and Y. J .He, “Generation of Static Function Calling Paths in C++ Based on Finite-State Machine”, [C]. Applied Mechanics and Materials, vol. 568, **(2014)**, pp. 1497-1504.
[16] X. ZHU, Y. MU and Z. ZHANG, “Analysis of function call path based on Soot control flow graph”, [J]. Data Communication, vol. 4, **(2012)**, pp. 26-29+35.
[17] M. M. Yan, Y. M. Mu and Y. J. He, “The Analysis of Function Calling Path in Java Based on Soot”, [C]. Applied Mechanics and Materials, vol. 568, **(2014)**, pp. 1479-1487.
[18] A. P. Xu, Y. M. Mu and Z. H. Zhang, “The Dynamic Function Calling Path Generation Based on Instrumentation”, [C]. Applied Mechanics and Materials, vol. 568, **(2014)**, pp. 1469-1478.
[19] J. C. Huang, “Program Instrumentation and Software Testing” [J]. Computer, vol. 11, **(1978)**, pp. 25-32.
[20] Z. Nethercote and J. Seward, “Valgrind: a framework for heavyweight dynamic binary instrumentation”, [C]//ACM Sigplan notices. ACM, vol. 42, no. 6, **(2007)**, pp. 89-100.
[21] S. L. Graham, P. B Kessler and M. K. Mckusick, “Gprof:A call graph execution profiler”, [C]//ACM Sigplan Notices.ACM, vol. 17, no. 6, **(1982)**, pp. 120-126.
[22] Y. M. MU and Z.YANG, “Verify consistency of software implementation and design based on function call path”, [J]. Science in China: Information Science, vol. 10, **(2014)**, pp. 1290-1304.
[23] L. Bang, A. Aydin and T. Bultan, “Automatically computing path complexity of programs”, [C]// Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. ACM, **(2015)**, pp. 61-72.