A Novel Approach to Construct Object-Oriented System Dependence Graph and Algorithm Design

Lin Du
School of Computer Science and Technology, University of Qilu Normal, Jinan, China
Email:dul1028@163.com

Guorong Xiao
School of Computer Science and Technology, Guangdong University of Finance, Guangzhou, China
Email: newducky@126.com

Daming Li
School of Computer Science and Technology, University of Qilu Normal, Jinan, China
Email: deff_lee@sina.com

***Abstract*—On the basis of analyzing the defects that traditional system dependence graph have, a novel method based on ripple effect is proposed to construct coarse-grained system dependence graph. The method perfects object-oriented program semantics and reduces the computation complexity through expanding the signification of coarse-grained and analyzing ripple effect. Object-oriented program semantics are described in detail. The algorithms for analyzing ripple effects and constructing system dependence graph are designed. Furthermore the computation complexity of algorithms is analyzed to validate effectiveness.**

***Index Terms*—object-oriented system dependence graph, ripple effect, coarse-grained, algorithms implement, computation complexity**

I. INTRODUCTION

Dependence analysis is the most important part of any program restructuring process. When a compiler tries to restructure a given program to satisfy a certain goal without changing the meaning of the program, it must honor certain dependence constraints. These constraints are determined by the order in which each program variable is defined and used during the course of execution of the program. Object-oriented system dependence graph (OOSDG) is the technology of dependence analysis. OOSDG can reflect the semantic characteristics of object-oriented programs, can deal with data flow and control flow between processes, can describe parameter transference and carry out inter-process analysis.

However, there are the following questions in the actual construction of system dependence graph.Firstly, the method to construct system dependence graph is complicated, what's more, lack of accuracy. Because of high complexity, on account of the different study, the actual construction ignores some of the semantic characteristics of object-oriented program [1]. This would result in inaccurate. For example, in the OOSDG, only one parameter node is constructed for each corresponding class member variable. The class member variable has a separate copy in each class instance respectively. That is to say that a class member variable defined in one place is used in different places. This would result in the wrong data dependence among different class instances. Secondly, traditional construction method results in the loss of program semantic. System dependence graph based on analysis of process, lack of semantic association, does not reflect the characteristics of object-oriented language completely. The interactions among objects constitute the main framework of the object-oriented program. However this interaction doesn't base on the order of execution. So the construction of the traditional system dependence graph would not take into account all of relationship among objects. Therefore, the semantic of object-oriented is lost a lot. Sometimes the use of traditional system dependence graph for program understanding is more difficult than a complete program.

Based on analyzing above defects that traditional system dependence graph have, the method based on ripple effect analysis is presented to construct coarse-grained system dependence graph. First of all, the meaning of coarse-grained is extended in order to make the size of grain come up to object-oriented program's semantic unit that is class, instance, member method and member variable. Ripple effects analysis plays the role of two aspects. First, the results of the ripple effect are mapped to the dependence graph in order to add semantic relationship among different objects. Second, the scope of analysis through ripple effect is narrowed in order to reduce the complexity of constructing graph.

The rest of the paper is organized as follows. In section II, the meaning of coarse-grained is extended to simplify system dependence graph. The method of ripple effect analysis is proposed. Object-oriented program semantics is described in detail. In section III, this paper designs the algorithms for analyzing ripple effects and constructing system dependence graph .What's more, the computation complexity of the algorithms is analyzed to validate effectiveness. In the section IV, we show an example to

doi:10.4304/jsw.7.1.133-140

demonstrate our proposed approach for constructing coarse-grained system dependence graph. Section V concludes the whole paper.

II. Constructing Coarse-grained System Dependence Graph Based on Ripple Effect Analysis

A. Expanding the Meaning of Coarse-grained and Simplifying System Dependence Graph

For the traditional system dependence graph, fine-grained reach the level of statement, coarse-grained reach the level of method. In the object-oriented program, the factor that affect program is not only variable but also the interaction among objects which constitute the main framework of the program. In order to analyze object-oriented programming, it is necessary to make class, member method, member variable which are separate semantic units as the research object independently [2]. For understanding of object-oriented programming, analyzing the interaction relationship among multiple units is better than analyzing single statement. The meaning of coarse-grained is extended in order to make the size of grain come up to object-oriented program's semantic unit that is class, instance, member method and member variable. Coarse-grained expanded is defined as follows.

Definition 1. *The graph G which meets the following characters is referred to as coarse-grained. (1) Graph G contains statement and predicate in the main(),class, instance, member method and member variable.(2) If a statement which is in the member method M belongs to G, then M also belongs to G. (3) If the instance, member method or member variable in the class A belongs to G, then class A also belongs to G.*

On the basis of defining the coarse-grained, system dependence graph is simplified. Describing the process dependence doesn't need to enter the process inside but indicate process prelude node only. The data dependence which belongs to parameter nodes of different methods is indicated by data dependence among multiple methods. It is achieved by data dependence edge which point to the call directly. Take the following figures for example, figure 1 is a traditional system dependence graph, figure 2 shows the meaning of edges in the graph, and figure 3 is the corresponding system dependence graph which is simplified by our method.

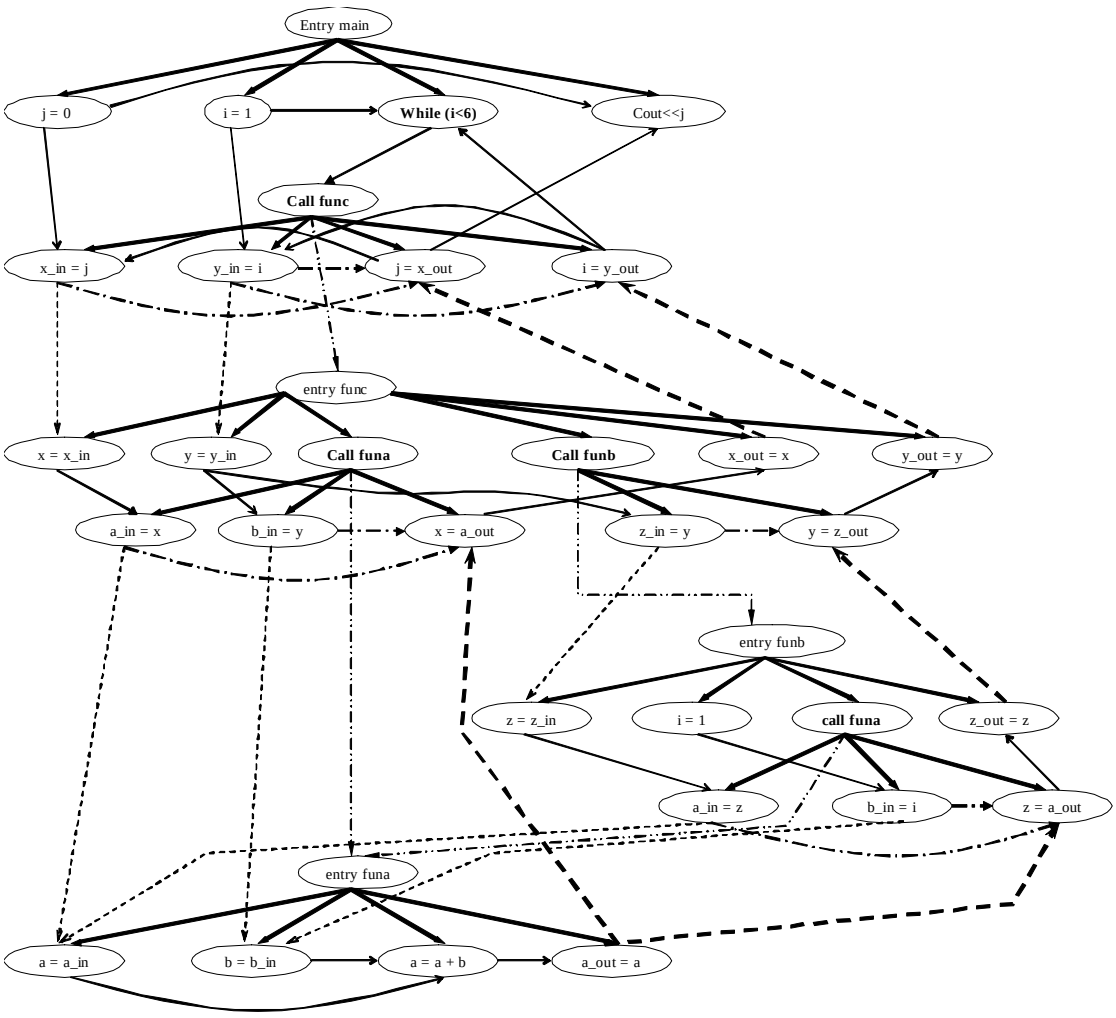

Figure 1. an example of traditional system dependence graph

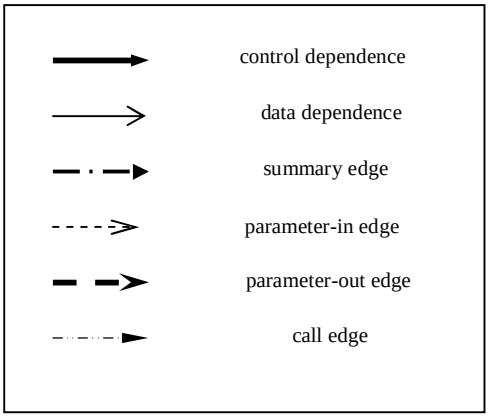

Figure 2. the meaning of edges

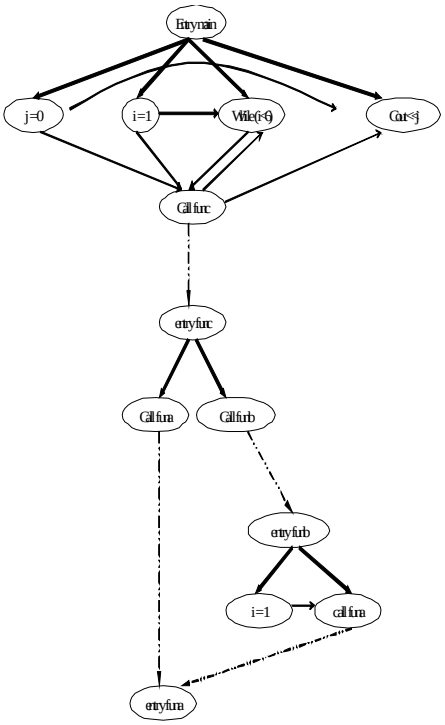

Figure 3. simplified system dependence graph

B. Ripple Effects Analysis

Ripple effects analysis plays the role of two aspects. First, the results of the ripple effect are mapped to the dependence graph in order to add semantic relationship among different objects. Second, the scope of analysis through ripple effect is narrowed in order to reduce the complexity of constructing graph.

Analyzing ripple effect is to record the units involved by the ripple of one unit which is called the source of ripple. The following method is used. First step, the complete ripple graph which reflects corresponding object-oriented programming is constructed. Starting from the source of ripple, the direct and indirect ripple unit can be found through traversal all the ripple edges. However above method has problem as follows. The method to construct the complete ripple graph is complicated whose computation complexity is same as constructing system dependence graph. The result does not match our original intention we want to reduce the analysis scope through ripple effects analysis. Object-oriented program has the following properties. The interaction among the various units is either direct or indirect. In particular, the indirect relationship can be expressed by the direct relationship among multiple units. This is called transitive. We can draw on the experience of the method to process transitive in the cluster. Ripple effect can be recorded through the use of matrix. Thereby the complete ripple effects can be calculated through matrix transitive operations.

Example 1. Let us consider the following example in which L1...L13 is the number of the corresponding statement.

```
L1:  class A
       { public:
L2:      virtual int F( ){…}
       }
L3:  class B: class A
       { public:
L4:      virtual int F( ) {…}
       }
L5:  class C
       { public:
L6:      A *ca;
L7:      int V( )
L8:       { A *a; }
L9:      a=new A( );
       }
L10:  a->F( );
L11:  ca=a;
L12:  delete a;
L13:  ca->F( );
```

The matrix is defined as follows. It is the first to make sure the units participated in ripple effect. If the number of all the units is n, then n * n matrix is constructed. In the matrix, the elements can be used 1 or 0 to represent ripple or not. Each row (or column) corresponds to the ripple unit. If the row number is the same as the column number, then its corresponding ripple unit is same. For each unit which is located in row i and column j, if ripple effect exits because corresponding ripple unit of row i acts on column corresponding ripple unit of column j, the unit's value is 1.On the contrary, the unit's value is 0. All the diagonal units' values are 1 because ripple effect influences units themselves. The matrix defined as above records ripple effect of the object-oriented program.

For above experimental codes, the unit B is the source of the ripple which launches the ripple. Both REO and REA are 10 * 10 matrix. The units corresponding to the row (or column) in accordance with row number (or column number) are arranged in order of size as follows. A, B, C, AF, BF, CV, C.ca, C.V.a, C.ca.F, C.V.a.F. The following matrixes are derived from above matrix definitions and algorithms.

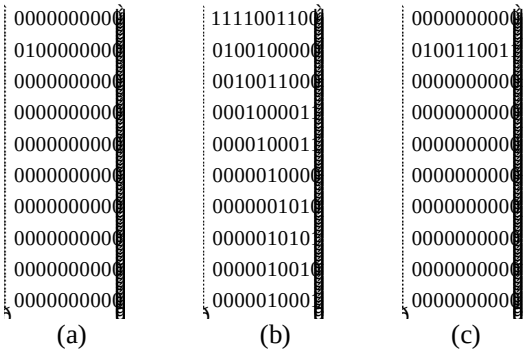

Figure 4. (a) the initial REO; (b) REA; (c) the final REO

For the final REO, the value of each unit doesn't change after the calculation is done over. If the unit value is 1, the unit is involved by ripple whose ripple source is class B. These units are the following(B, B.F, C.V, C.ca.F, C.V.a.F). It means that these units change with class B together.

C. Semantic Description

Object-oriented program semantics is described in detail in the system independence graph based on ripple effect as follows.

(1) Description of Class, Instance, Member Method, the Relationship among Member Variables

In order to express membership, instance node, member method prelude node and member variable node are connected to the accessory class prelude node. Method and process have the same status. For method and process we do not achieve internal processing but provide prelude node. The meaning of method prelude node is expanded through hiding data transference among multiple parameter nodes. The expression of data dependence among parameter nodes of different methods relies on data dependence among methods. Then three kinds of nodes should be increased as follows. Member variable node should be increased because member method refers to member variable. Instance node should be increased because member method refers to class instance node. When a method is called by the class instance, instance node expressing message receiver object is increased in the method node. The aim is to reflect the change of object's state.

(2) Description of Class Inheritance

In order to express inheritance, different class prelude nodes which have inheritance are connected. When one class interacts with another class, it is convenient to couple each other through class prelude node and class member edge. In order to reflect the inheritance hierarchy clearly and reduce backtracking, we take the following approach. If a virtual method in the child class which inherits from the parent class is modified, the method is described only in the child class. Meantime, associated edge should be increased between class prelude node of the parent class and method prelude node of the child class [3]. This makes the expression of inheritance mechanism and virtual method doesn't require increasing associated edge between method of the parent class and method of the child class. Only the associated edge is increased between class prelude node and method prelude node.

(3) Description of Polymorphism and Dynamic Binding

Polymorphism can be expressed completely by the virtual method prelude node and polymorphism call edge. Through multiple call edge, call node can be connected to each method node which is called by object possibly. The dynamic selection can be expressed by multiple polymorphism nodes which have the same protocol. This method can express all the possibilities.

III. ALGORITHM DESIGN AND ANALYSIS OF THE COMPUTATION COMPLEXITY OF ALGORITHMS

A. Algorithm Design

(1) Computing Ripple Effect

Algorithm 1. Ripple effect analysis.
Input: the source of the ripple
Output: the units influenced by the ripple source
Step 1: The range of analysis which consists of the source of the ripple and all the units which participate in the ripple effect analysis is determined.
Step 2: The matrix REA and the matrix REO are defined. REA recorded the direct ripple effect of all the units. REO is defined as follows. The value of the unit which is the source of the ripple is 1.Another units' value are 0. It's easy to find that the unit whose value is 1 is must located in the diagonal of the matrix REO.
Step 3: The transmission of ripple effect is calculated. REO=REO*REA. For the matrix REO, all the units' value is modified 1 if the value is not 0.
Step 4: If the matrix REO is different from the initial REO, then the algorithm return to step 3.Otherwise, the algorithm would carry on the next step.
Step 5: If the matrix REO varies no longer, the ripple effect analysis is over. For the final REO, the units whose value is 1 are the required units.

(2) Constructing system dependence graph

The scope of the description of the system dependence graph is reduced to the statement, the predicate in the main ( ) and class, instance, member method and member variable achieved by the ripple effect analysis.The object-oriented program is represented as a two-tuples(M, C) in which M is the main ( ) and C is a collection of classes. The algorithm calls the function connect ( ) which connect call node of process dependence graph and method prelude node . Meantime class graph and the main program are connected [4-5].

Algorithm 2. *Constructing system dependence graph..*
Input: the abstract syntax tree of P = (M, C)
Output: the OSDG of P

```
void ConstructOSDG ( )
  { for (class Ci of C)
     { for (method m defined in Ci)
        {if (m is "marked")
           make Ci and the"marked"
           method of base    class
           connected as membership;
         else{
           calculate the PrDG of m;
           make m as"marked";
             }
        }
      }
    connect( );
  }//end ConstructOSDG
```

B. Analysis of the Computation Complexity of Algorithms

In order to validate effectiveness of this paper's algorithm, let us compare the algorithms among another methods and this paper's method. The multiple variables which influence the computation complexity are defined in the table as follows [6].

TABLE I.
variables which influence the computation complexity

Name of the variable	Meaning of the variable
Vertices	the maximum number of predicates and assignment statements in the method or process
Edges	the maximum number of edges in the method or process
Globals	the number of global variables
InstanceVars	the maximum number of instance variables in the class
CallSites	the maximum number of call nodes in the method or process
TreeDepth	the number of call path which is direct or indirect
Methods	The number of methods or processes

(1) The computation complexity of constructing process-based program system dependence graph.

The parameters of a method include formal parameter input node, formal parameter output node, actual parameter input node and actual parameter output node. Upper bound for the parameters of the method m shows as follows.

$$Parameter(m)=Params+Globals+InstanceVars \quad (1)$$

The meaning of Params is the number of parameters. Then the computation complexity of the method m shows as follows.

$$Size(m)=O(Vertices+CallSites*(1+TreeDepth*(2*Parameter(m)))+2*Parameter(m)) \quad (2)$$

(2) The computation complexity of constructing traditional object-oriented program system dependence graph.

The parameters of a method include formal parameter input node and formal parameter output node. This is different from the process-based program system dependence graph as above. Then the computation complexity of the method m shows as follows [7-8].

$$Size(m)=O(Vertices+CallSites*(1+TreeDepth*ParamVertices(m))+ParamVertices(m)) \quad (3)$$

(3) The computation complexity of this paper.

The matrix records only direct ripple. Indirect ripple can be calculated through matrix transitive operations. Expressing dependence needs only one edge in ripple graph. However, multiple dependence edges are need in traditional system dependence graph. The meaning of method prelude node is expanded through hiding data transference among multiple parameter nodes. The expression of data dependence among parameter nodes of different methods relies on data dependence among methods. The parameters of a method include statement and predicate in the main( ), class, instance, member method and member variable.Upper bound for the number of the nodes depends on Size(m) as follows.

$$Size(SDG) = O(Size(m) * Methods) \quad (4)$$

In summary, the computation complexity of this paper is lower than traditional method.

IV. CASE STUDY

In this section, we show an example to demonstrate our proposed approach for constructing coarse-grained system dependence graph. Based on our method, ripple effects analysis plays the role of two aspects. First, the results of the ripple effect are mapped to the dependence graph in order to add semantic relationship among different objects. Second, the scope of analysis through ripple effect is narrowed in order to reduce the complexity of constructing graph. On the basis of defining the coarse-grained, system dependence graph is simplified. Describing the process dependence doesn't need to enter the process inside but indicate process prelude node only. The data dependence which belongs to parameter nodes of different methods is indicated by data dependence among multiple methods. It is achieved by data dependence edge which point to the call directly. Next, we apply this paper's method to construct coarse-grained system dependence graph for case codes in the figure 5. And figure 6 shows the meaning of edges and nodes in the system dependence graph. Corresponding system dependence graph for case codes is showed in figure 7.

```
L1: class D2Vector{
        protected:
L2:     int x;
L3:     int y;
        public:
L4:  D2vector(){
L5:         x = 0;
L6:         Y = 0;}
L7:  D2vector(int a, intb){
L8:         x = a;
L9:         y = b;}
L10:D2vector(const D2vector & p){
L11:        x = p.x + 1;
L12:        y = p.y +2;}
L13:virtual void scale(int n){
L14:        x = x * n;
L15:        y = y * n;}
L16:void reset(){
L17:        x = 0;
L18:        y = 0;}
}
L19:classs D3Vector{
        protected:
L20:    int z;
        public:
L21:D3Vector(){
L22:        D2Vector();
L23:        Z = 0;}
L24:D3Vector(int a, int b, int c){
L25:        D2Vector(a, b)
L26:        z = c:}
L27:D3Vector(const D3Vector &p){
L28:        x = p.x;
S29:        y = p.y;
L30:        z = p.z;}
L31:void scale(int n){
L32:        D2Vector::Scale(n);
L33:        z = z * n;}
L34:friend int extend_z(D3Vector &, int);
L35:friend int sum(D3Vector); }
```

```
    int extend_z(D3Vector &v, int n){
L36:    v.z = v.z * n;
L37:    return v.z;
}
    int sum(D3Vector v){
L38:    int v_sum;
L39:    v_sum = v.x + v.y + v.z;
L40:    return v_sum;
}
    main(int argc){
L41:    D2Vector * vp;
L42:    D3Vector v3(10, 10, 10);
L43:    int v3sum;
L44:    int dim_z;
L45:    if (argc > 1)
L46:        vp = new D3Vector(1, 1, 1);
        else
L47:        vp = new D2Vector(1, 1);
L48:    vp ->scale(10);
L49:    v3sum = sum(v3);
L50:    dim_z = extend_z(v3, 10);
L51:    cout<<v3sum<<endl;
L52:    cout<<dim_z<<endl;
}
```

Figure 5. case codes

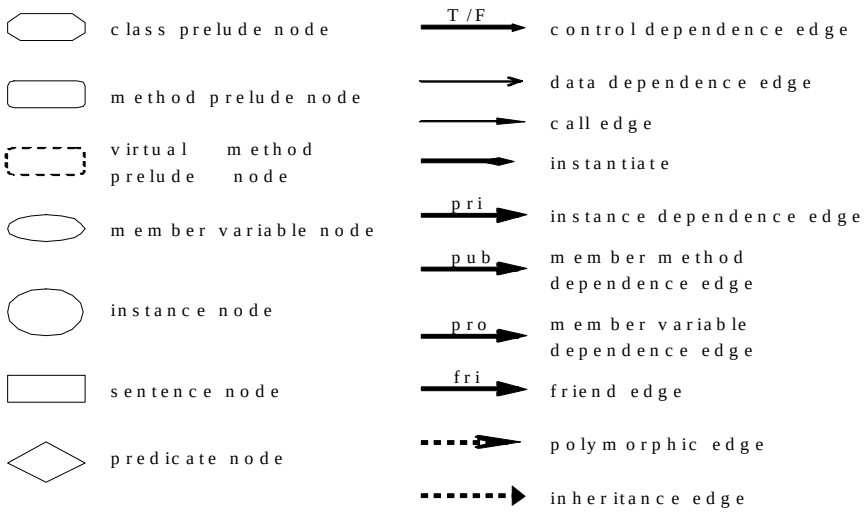

Figure 6. the meaning of edges and nodes

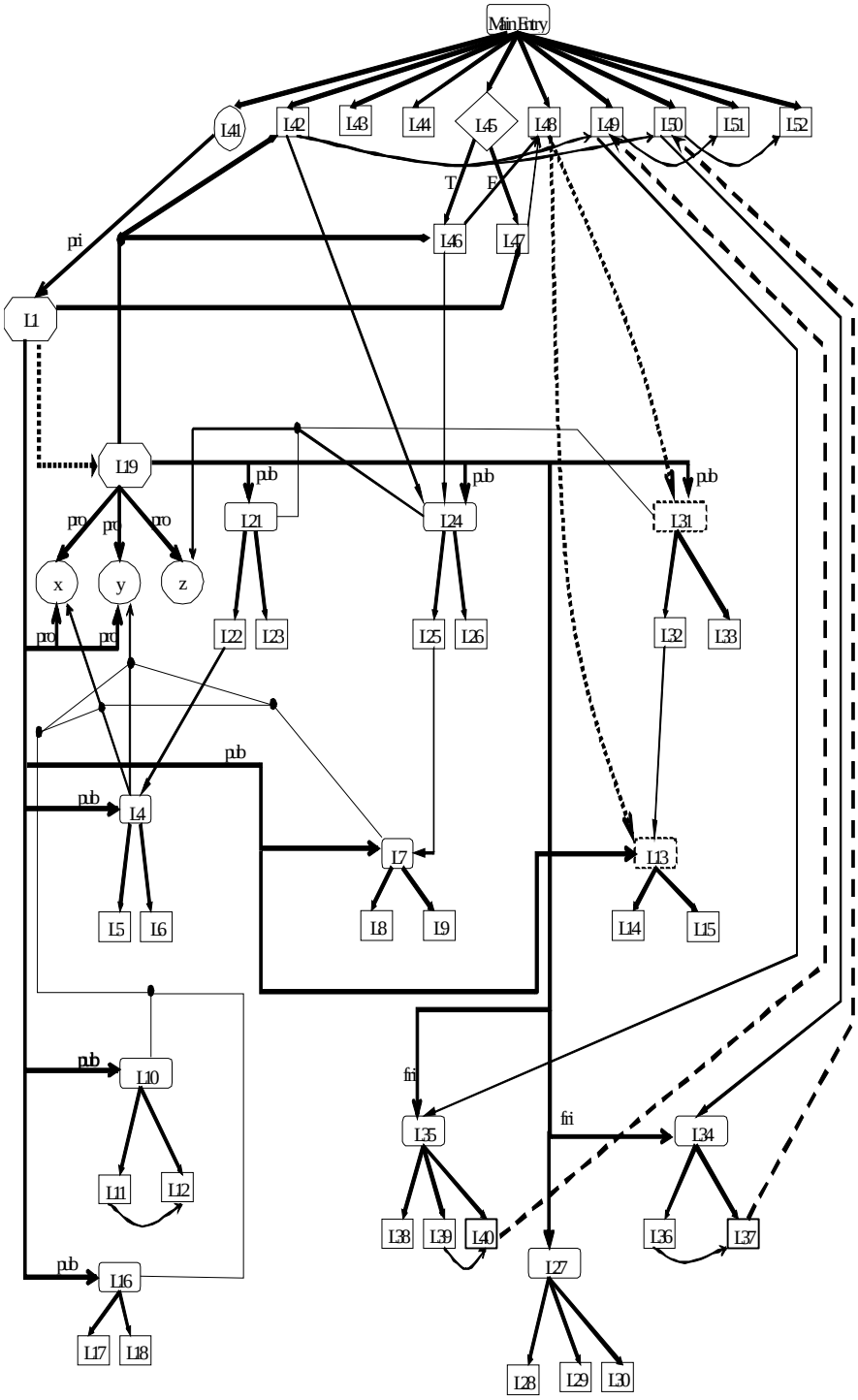

Figure 7. system dependence graph for case codes

V. CONCLUSION

The defects that traditional system dependence graph have include high computation complexity, deficiency of accuracy and loss of program semantic. In this paper the method based on ripple effect is proposed to construct coarse-grained system dependence graph. Coarse-grained is extended and defined in order to make the size of grain come up to object-oriented program's semantic unit that is class, instance, member method and member variable. Ripple effects analysis plays the role of two aspects. First, the results of the ripple effect are mapped to the dependence graph in order to add semantic relationship among different objects. Second, the scope of analysis through ripple effect is narrowed in order to reduce the complexity of constructing graph. Finally, the algorithms for analyzing ripple effects and constructing system dependence graph are designed. Furthermore the computation complexity of algorithms is analyzed to

validate effectiveness. This paper's constructing methods correctly reflect the semantic of object-oriented programs, also provide a solid foundation for the further analysis and reengineering of legacy software. Recent works about dependence analysis are concerned on improving the precision of constructing algorithms. We strongly believe that, in the near future, this research field will be paid more and more attentions by the researchers and will promote the fundamental theories research in the related fields, for example program debugging [9-10], program testing [11-13], software measurement [14-15] and software maintenance [16].

ACKNOWLEDGMENT

This work is partially supported by the Shandong Province High Technology Research and Development Program of China (Grant No. 2011GGB01017); Research Foundation of Qilu Normal University; Soft Science Research Program of Shandong Province (Grant No. 2011RKB01062).

REFERENCES

[1] Chae HS, Kwon YR:A cohesion measure for classes in object-oriented systems, Proceedings of the 5th International Software Metrics Symposium. IEEE Computer Society Press, 1998, pp.158-166.
[2] Briand LC, Morasca S, Basili VR: Defining and validating measures for object-based high-level design. IEEE Transactions on Software Engineering, vol.25, no. 5,1999, pp.722-743.
[3] B.Korel, L.Taha, A.Bader: Slicing of state based models，Proceedings of the IEEE International Conference on Software Maintenance, 2003, pp.34-43.
[4] Mary Jean Harrold: Regression Test Selection for Java Software, OOPSLA 2001, pp.313-326 .
[5] Ap Xu BW, Zhou YM: Comments on a cohesion measure for object-oriented classes, Software--Practice and Experience, vol.31, no.14, 2001, pp.1381-1388.
[6] Chen ZQ, Zhou YM, Xu BW, Zhao JJ, Yang HJ:A novel approach to measuring class cohesion based on dependence analysis, IEEE International Conference on Software Maintenance,2002, pp.377-383.
[7] Chen ZQ: Slicing object-oriented Java programs, ACM SIGPLAN Notices, vol.36, no. 4, 2001, pp.33-40.
[8] Xu BW, Chen ZQ, Zhou XY: Slicing object-oriented Ada95 programs based on dependence analysis. Journal of Software, vol.12, no. 12, 2001, pp.208-213.
[9] Jiang,S, Zhang,C: A Debugging Approach for Java Runtime Exceptions Based on Program Slicing and Stack Traces. In: Proceedings of the 10th Quality Software International Conference, 2010.
[10] Horwitz, S.,Liblit, B.,Polishchuk, M: Better Debugging via Output Tracing and Callstack-Sensitive Slicing. Software Engineering, 36(1), 2010, 7-19.
[11] Seoul, Korea: Test Sequence Generation from Combining Property Modeling and Program Slicing. In: Proceedings of the 34th Annual Computer Software and Applications Conference, 2010.
[12] Rupak Majumdar, Ru-Gang Xu: Reducing Test Inputs Using Information Partitions. Lecture Notes in Computer Science, Vol 5643,2009, 555-569.
[13] Seoul: An Approach to Regression Test Selection Based on Hierarchical Slicing Technique. In: Proceedings of 2010 IEEE 34th Annual Computer Software and Applications Conference.
[14] San Diego, California: Program Execution-Based Module Cohesion Measurement. In: Proceedings of the 16th IEEE International Conference on Automated Software Engineering, 2001.
[15] Meyers, T.M, Binkley, D: Slice-based cohesion metrics and software intervention. In: Proceedings of the 11th working conference on Reverse Engineering, 2004.
[16] Emily .Hill,Lori.pollock: Exploring the neighborhood with dora to expedite software maintenance. In: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, 2007.

Lin Du, born in 1978, earned a B.S. and M.S. degree in Computer Science& Technology from Shandong University, in 2002 and 2006 respectively. After graduate school, he joined school of Computer Science & Technology, Qilu Normal University in 2006. His current research interests include software reengineering, system comprehension, software measurement and image retrieval.

Guorong Xiao, received the B.E. and M.E. degrees in computer science and technology from South China University of Technology, Guangzhou City, China. He is currently a lecturer at the department of computer science and technology, GuangDong University of Finance, China. His research interests are data mining, data warehouse and financial analysis etc. He has finished several banking, securities and mobile data warehouse.

Daming Li, born in 1964, Ph.D. degree, he is an associate professor of computer department in Qilu Normal University, engages in the teaching and scientific research. The main area of his research is algorithm analyzing and design.