종속성 그래프 기반 클래스 테스팅

The class testing based on a dependence graph

임 동 주* 배 상 현**
Dong Ju Im Sang Hyun Hae

요 약

절차적 프로그램의 표현방법은 클래스, 객체, 계승, 동적 바인딩등으로 이루어진 객체지향 프로그램 표현에 그대로 적용될 수 없다. 더군다나 기존의 프로그램 종속성은 변수간이 아니라 문장간의 종속성을 나타내고 있다. 즉, 주어진 변수에 어떠한 변수들이 영향을 미치고 있는가 하는 문제를 해결할 수 없다. 따라서 본 연구는 객체지향 프로그램에서 변수간의 종속성을 포함한 구현 수준의 정보를 나타내는 메소드 종속성 모델을 제시하고자 한다. 또한 객체지향 프로그램의 테스트 적합성 기준에 근거한 구현기반 클래스 테스팅 방법을 제안한다. 데이터 멤버간 종속성과 테스트 데이터 적합성에 대한 공리들을 고려하여 흐름 그래프 기반 테스팅 기준을 만족시키는 테스트 케이스인 메소드의 시퀀스를 생성시킨다. 파생 클래스 테스팅을 위해서 유산관계와 실험을 통해 테스트 비용 절감을 검증한 부모 클래스에 대한 테스팅 정보의 재사용성을 고려한다.

Abstract

The representation of a procedural program cannot be applied directly to object oriented program representation consisting of class, object, inheritance, and dynamic binding. Furthermore, preexisting program dependence represented the dependence among statments, but not among variables. That is, it could not solve the problem of which variables make an effect on given variables. Consequently, this study presents the method dependence model representing implementation level information including the dependence among variables in an object oriented program. I also propose implementation-based class testing technique based on the test adequacy criterion of an object-oriented program. Considering inter-data member dependences and a set of axioms for test data adequacy, it generates sequences of methods as test cases which satisfy a flow graph-based testing criterion. For a derived class testing, it considers inheritance relationship and the resuability of the testing information for its parent classes which verified the reduction of test cost through the experiment.

1. Introduction

A program dependence graph is the most widely used application field in the various representation methods of a procedural program[5]. OMT(Object Modeling Technique) which is the representation of the object-oriented design cannot be used for a code analysis, because it contains the design-level information, but not the detailed implementation information.

The existing representation has some restrictions when a program dependence graph is used in the new field of software engineering.

First, the representation cannot be applied directly to an object-oriented program. A program dependence graph should represent the difference from a procedural program caused by class, object, inheritance, and dynamic bindings for the proper representation of an object-oriented program dependence. For example, it should represent the method which is not a procedure, the dynamic bindings of method call due to the polymorphism, and the flow dependence relationship between methods.

Second, the prior program dependence could not

* 정 회 원 : 조선대학교 전산통계학과
imdongju@hanmail.net
** 종신회원 : 조선대학교 전산통계학과 교수
shbae@chosun.ac.kr

express the variable dependence precisely, although it represented the statement dependence because the primary concern of a compiler was whether or not the order of statements could be changed without affecting a program execution. That is, the problem of which variables can make an effect on given variables is not able to be solved.

Third, the program dependence graph is too large and complex to understand, becauseit contains the dependnece graphs of all the called procedures in a single graph. The procedure-unit query can be solved by the modularized dependence graph only. Therefore, it is desirable to apply the divide&conquer and modularization principles of software engineering to a program dependence representation.

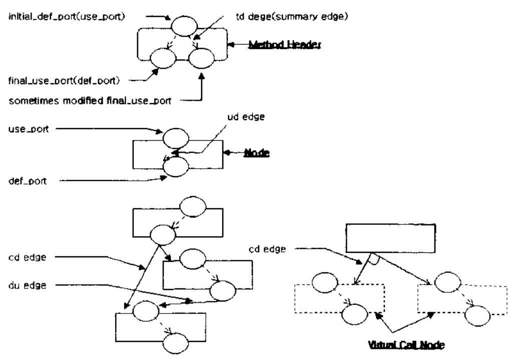
Most of the research on software testing is for a procedure-oriented software. Object-oriented software structure is different from procedure-oriented one. Even though a conventional testing is efficient, it cannot be applied directly to object-oriented software [8][9][10].

A basic composition unit in object-oriented software is a class. An individual testing of the methods which are the operations defined in a class is simple, but a class testing cannot be reduced to the independent testing[11][15]. Each method operates mutually with other methods by being executed based on a given object state and by changing the state. The execution environment of the methods is given not only by parameters but also by an object state. The methods influencing an object state can be regarded as the public methods accessible out of the exterior of the class, and the methods can be called in an arbitrary order. Therefore, it is important to determine in what order the methods are executed for a class testing.

2.Method dependence model

The method dependence graph proposed in this section is based on the procedural program dependence model Jackson and Rollins[4] studied, and it is extended so that it can be applied to the object-oriented program. A few kinds of procedural program dependence graphs exist without a standardization, and they are a little different from one another in the application field. The important thing is the level to which a graph represents a dependence. Horwitz et al. asserted that a dependence graph should be as follows for the programs to terminate with the same execution state when the program dependence graphs are of the same type[2][3]. It should represent the dependence relationship at least described below with an entry node, an initial definition node of each variable, and a final use node of each variable added in addition to the node representing each statement. The dependence is divided into a control and a data dependnece, a data dependence is into a flow and def-order dependence, and a flow dependence is into a loop independent and a loop carried dependence.

A node to call a method has data members as an implicit parameter in addition to parameters and global variables, because the data members are like global variables in the method defined in a class. Therefore, the used or defined data members occur



(Fig. 1) Method dependence graph representation symbol in the port pertaining to the calling node.

For simplicity, it is assumed that each statement is defined as a node and a parameter has a different name from a data member.

A method dependence graph using Fig. 1 is 6-tuple as follows.

Method dependence graph
MDG(M)=<Node, Port, $\rightarrow^{cd}$, $\rightarrow^{du}$, $\rightarrow^{ud}$, $\rightarrow^{td}$>
Node = program_Statement $\cup$ {method_header}
Vars = variable $\cup$ object $\cup$ class $\cup$ {γ, τ, null_port}
Port $\subseteq$ Vars $\times$ Node

Summary edge($\rightarrow^{td}$), internal edge($\rightarrow^{ud}$), and external edge($\rightarrow^{cd}$ or $\rightarrow^{du}$) are defined with a different dependence, and external edge is divided into a data flow and a control dependence.

$\rightarrow^{du}$, $\rightarrow^{ud}$, $\rightarrow^{td}$ $\subseteq$ Port $\times$ Port
$\rightarrow^{cd}$ $\subseteq$ Node $\times$ Node

Internal edge $\rightarrow^{ud}$ represents a data dependence from the use of a variable or a constant through a variable definition, and external edge $\rightarrow^{du}$ from the variable definition of a node through the same variable use of the

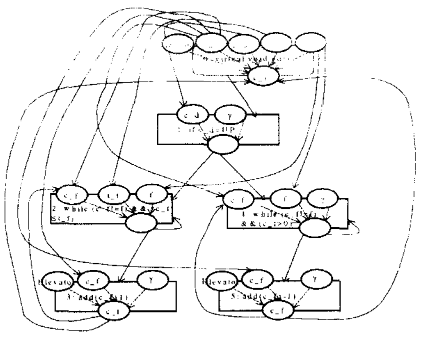
other node. $\rightarrow^{cd}$ designates a control dependence from the execution result of a node(or a header node) through the other node.

$\rightarrow^{du}$ $\subseteq$ {((x,I),(x,J)) | DEF(x,I) $\wedge$ USE(x,J)}
$\rightarrow^{ud}$ $\subseteq$ {((x,I),(y,I)) | USE(x,I) $\wedge$ DEF(y,I)}
$\rightarrow^{cd}$ $\subseteq$ {(I,J) | J is control dependent on I}

The definitions of USE(v,n) and DEF(v,n) are as follows.

■ USE(v,n)

A node n∈Node is the use node of a variable v ∈Vars. ⇔ a variable v is used at the statement to a node n.


(Fig. 2) Method dependence graph

■ DEF(v,n)

A node n∈Node is the definition node of a variable v∈Vars. ⇔ a variable v is defined at the statement to a node n.

Figure 2 is the representation for which this research is applied to the example program , Elevator::go() that is extracted a from literature[6].

■ Example program

```
class Elevator{
public:
  Elevator(int_top_floor)
   {current_floor=1;
     current_direction=UP;
     top_floor=1_top_floor;}
  virtual~Elevator() {}
  void up()
    {current_direction=UP;}
  void down()
    {current_direction=DOWN;}
  int which_floor()
    {return current_floor;)}
  Direction direction()
    {return current_direction;)
  virtual void go(int floor)
    {if(current_direction=UP)
      {while((current_floor!=floor)&&
                          (current_floor>0))
```

```
         add(current_floor,-1);}
  }
  Private:
    void add(int &a, const int &b)
     {a=a+b}
  protected:
    int current_floor;
    Direction current_direction;
    int top_floor;
  };

  class AlarmElevator:public Elevator{
    public:
      AlarmElevator(int top_floor):
                           Elevator(top_floor)
        {alarm_on=0;}
      void set_alarm()
        {alarm_on=1;}
      void reset_alarm()
        {alarm_on=0;}
      void go(int floor)
        {if(!alarm_on)
         Elevator::go(floor);
        }
  protected:
    int alarm_on;
  };
```

3. Class testing using slicing

The testing of a class which is a basic composition element of an object oriented program is proposed in this section. As a class testing cannot be reduced to the independent testing of the methods, the mutual operation between the methods through the data members should be tested. The dependence between the data members should be also tested. To begin with, a base class testing is proposed in section 3.1, with the points considered.

A derived class testing should consider the mutual operation between the classes through an inheritance and the reusability of the test cases developed already in the process of a base class testing.

3.1 Base class testing

The individual method is a condensed function and a minimum unit of the testing. The unit testing of a conventional software testing can be applied for a method unit testing. The local variables or data members defined and used in the body of the method are tested with the previous data flow testing.

In case the call of the other method occurs in the body of a method, the integration testing of a conventional software testing can be applied for the method integration testing, as the call relationship can be determined statically.

The set of the methods using or defining a data member d as level-0 can be identified from a class data flow graph. The definition set of level-i can be grasped by the backward traversal of a class hierarchy with a data member d selected as a slicing criterion. That is, when a backward edge traversal is made from a data member d, the methods marked first become the elements of $D_0(d)$, and the methods marked second in the continual traversal become the elements of $D_1(d)$.

[Definition 1] $D_i(d)$, definition of level-i m_1 defines d_1 as level-0, if a method m_1 defines a data member d_1 directly.
m_2 defines d_1 as level-1, if a method m_1 defines a data member d_1 as level-0, and a method m_2 defines a data member d_2 used to define d_1 as level-0. The definition of level-i is also defined in the same way. If level-k or level-(k+1) is possible to be defined, it is defined as level-k. The set of the methods defining d as level-i is designated as $D_i(d)$.

For example, even though the method defining d_l as level-1 does not define d_l within it directly, it makes an indirect effect on d_l by defining d_2 influencing the definition of d_l.

The use of level-0 is defined like the definition of level-0.

[Definition 2] $U_0(d)$, the use of level-0 m_l uses d_l as level-0, if a method m_l uses a data member d_l directly. The set of the methods using d as level-i is designated as Ui(d).

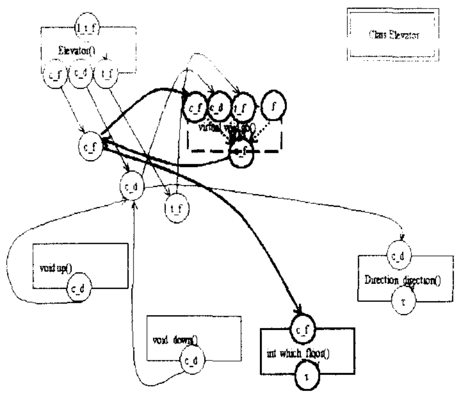
A data member definition matrix represents the direct or indirect dependence between the methods through the data members caused by the use-definition relationship of data members.

[Definition 3] Data member definition matrix
When the number of a class data member is n, a data member definition matrix is a n×n matrix designated as Def_M(i,j), which is defined as follows.
If $D_0(d_j)$ defines d_i as level-k, Def_M(i,j) = d_k

For example, as $D_0(d_i)$ is the set defining d_i as level-0 from Def. 1, the value of Def_M(i,i) is d_0. If the value of Def_M(i,j) is d_k, the method set $D_0(d_j)$ defining d_j as level-0 makes an indirect effect on a data member d_i.

$$\begin{array}{c|c|c|c|c|c|c|}
 & D_0(d_1) & D_0(d_2) & \cdots & D_0(d_j) & \cdots & D_0(d_n) \\
\hline
d_1 & d_0 & & \cdots & & & \\
\hline
d_2 & & d_0 & \cdots & & & \\
\hline
\vdots & & & \cdots & & & \\
\hline
d_i & & & & d_k & & \\
\hline
\vdots & & & & & & \\
\hline
d_n & & & \cdots & & & d_0 \\
\hline
\end{array}$$

The base class testing process described so far has been studied along with the examples. An example of a class Elevator is explained in the program of Fig. 2. The data flow graph of a class Elevator is as Fig. 3.



(Figure 3) data flow graph of class edges omitted

A data member definition matrix, and the definition and use set of which a class slice of each data member is composed, are as below.

$$\begin{array}{c|c|c|c|}
 & D_0(c\_f) & D_0(c\_d) & D_0(t\_f) \\
\hline
c\_f & d_0 & d_1 & d_1 \\
\hline
c\_d & & d_0 & \\
\hline
t\_f & & & d_0 \\
\hline
\end{array}$$

Constructor = { Elevator },

$D_0(c\_f)$ = {go}, $D_0(c\_d)$ = {up, down}, $D_0(t\_f)$ = ∅,

$U_0(c\_f)$ = {go, which_floor}, $U_0(c\_d)$ = {go, direction}, $U_0(t\_f)$ = {go}

$D_1(c\_f) = D_0(c\_d) \cup D_0(t\_f)$ = {up, down}, $D_1(c\_d) = D_1(t\_f)$ = ∅

(Table 1) Test example of base class

d	$D_0(d)$	$D_1(d)$	unit testing of data member	integration testing of data member
c_f	go()	up(), down()	Elevator, go, which_floor, go	Elevator, up, go, which_floor, down, go, which_floor, go, which_floor, go
c_d	up(), down()	∅	Elevator, up, go, direction, down, go, direction	Executed in unit testing
t_f	∅	∅	Elevator Executed in unit and integration testing of method	Executed in integration testing

The test case generated in the unit testing and the integration testing of data members is represented in Table 1.

3.2 Derived class testing

Inheritance is the mechanism sharing the specification and code of preexisting classes in defining a new class. Therefore, a derived class can be generated by defining the new attributes only different from the ones of the upper class using inheritance.

The class order in a class hierarchy is given by an inheritance relationship. The order is an inheritance relationship between class pairs. Consequently, through an inheritance, the reuse of the upper class testing information is possible in the testing process of a derived class, and the testing information of an immediate upper class only is considered.

A class AlarmElevator is studied in an example program of Fig. 2. The data flow graph is as Fig. 4.

A data member definition matrix, and the definition and use sets composing a class slice of each data member are as below.

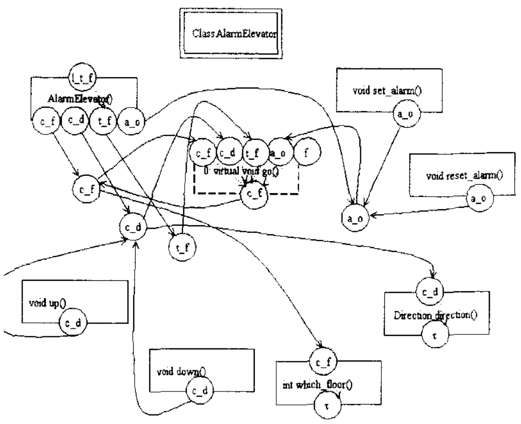
	D_0(c_f)	D_0(c_d)	D_0(t_f)	D_0(a_o)
c_f	d_0	d_1	d_1	d_1
c_d		d_0		
t_f			d_0	
a_o				d_0

Constructor = { AlarmElevator }

D_0(c_f) = {go}, D_0(c_d) = {up, down}, D_0(t_f) = $\emptyset$, D_0(a_o) = {set_alarm, reset_alarm }

U_0(c_f) = {go, which_floor}, U_0(c_d) = {go, direction}, U_0(t_f) = {go}, U_0(a_o) = {go}

D_1(c_f) = D_0(c_d) $\cup$ D_0(t_f) $\cup$ D_0(a_o) = {up, down, set_alarm, reset_alarm}, D_1(c_d) = D_1(t_f) = D_1(a_o) = $\emptyset$

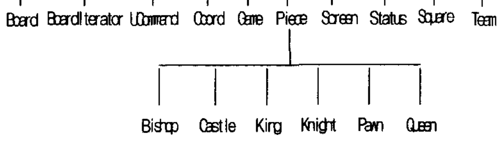


(Figure 4) Data flow graph of class Alarm Elevator(The member edges omited)

4. The analysis and the comparison of the class testing

I examined the reusability and the number of the test cases through the experiment. I analyzed which methods and data should be tested for the class testing in the class hierarchy as in the example of a chess game.

The result of the experiment showed that the number of the test case was 509, in the worst case not considering the reusability. In the case considering the reusability, however, only the 51% of the test cases could be performed, the 16% of which tested again, but it had the reusability. That is, the consideration of the reusability of the test cases applied in this paper could bring about the reduction of the 57% test cases of the whole.



(Figure 5) Class hierarchy used in the experiment

(Table 2) the classes used in the experiment and the number of the test cases of the research

Class	lines	class property in types					X	reusability	
		A	B	C	D	E		F+G	Y
Board	52	15	0	0	3	0	45	28	18
BoardI	14	3	1	0	5	0	15	9	9
UComm	20	14	1	0	4	0	25	19	19
Coord	36	9	0	0	2	0	31	20	11
Game	20	5	1	0	6	0	17	12	12
Piece	32	12	1	0	4	0	31	20	17
Screen	19	8	1	0	12	0	19	10	10
Status	24	12	1	0	7	0	26	15	15
Square	45	14	1	0	14	0	49	30	22
Team	64	19	1	0	1	0	61	44	34
Bishop	18	6	1	12	2	4	31	8	8
Castle	18	6	1	12	2	4	32	9	9
King	18	6	1	12	2	4	31	8	8
Knight	18	6	1	12	2	4	31	8	8
Pawn	30	9	1	12	2	4	34	11	11
Queen	18	6	1	12	2	4	31	8	8
Total	446	150	14	72	59	24	509	259 (51%)	219 (43%)
Average	28	9	1	5	4	2	32	16	14

A	:	newly defined methods
B	:	redefined methods
C	:	inherited methods
D	:	newly defined data
E	:	inherited data
X	:	test cases not considering the reusability
F+G	:	the test cases to be performed
Y	:	the test cases to be generated

(Table 3) The number of the test cases in types

class	A+B+C+D	F	G=H+I	H	I
Board	45	15	13	10	3
BoardI	15	3	6	2	4
UComm	25	14	5	2	3
Coord	31	9	11	9	3
Game	17	5	7	2	4
Piece	31	12	8	5	32
Screen	19	8	2	2	5
Status	26	12	3	2	3
Square	49	14	16	10	0
Team	61	19	25	12	1
Bishop	12	6	2	1	6
Castle	13	6	3	2	13
King	13	6	2	1	1
Knight	13	6	2	1	1
Pawn	16	6	5	4	1
Queen	13	6	2	1	1
Total	399	147	112	66	46
Average	25	9	7	5	3

A+B+C+D	:	inheritance and reusability not considered
F	:	intra-method & inter-method level
G	:	intra-class & inter-class level
H	:	data interdependency considered
I	:	data interdependency not considered

5. Conclusion

A proram dependence graph is used for a code analysis and for the analysis of an information for a test case generation in an implementation-based method and class testing, because it contains the detailed implementation information including a control and data flow dependence relationship between the variables.

In this paper, an implementation-based testing and the generation of the test cases are proposed, and the adequacy criterion of object-oriented program testing studied so far is considered as follows. First, the test cases generated in this research satisfies all use coverage criterion of the test case selection criterion based on a flow graph. Second, the antidecomposition, anticomposition, and antiextensionality axioms of Perry and Keiser are applied to the testing process. Object-unit testing considering the inter-dependence between data members for the validation of a class correctness is proposed, and in the case of a derived class, the inter-class testing considering an inheritance is proposed. Third, the reusability of the test cases is considered. That is, the testing information analyzed previously in the testing process of a derived class is reused.

It is proved through the experiment the fact that the consideration of the reusability can reduce the number of the test cases.

Acknowledgement

The study was supported by Factory Automation Research Center for Parts of Vehicles(FACPOV) in Chosun Univ. Kwangju. Korea. FACPOV is designated as a Regional Research Center of Korea Science and Engineering Foundation(KOSEF) and Ministry of Science & Techonology(MOST) operated by Chosun Univ.

References

[1] D. W. Binkley and K. B. Gallagher, "Program *Slicing," Technical Report,* IBM, 1996.

[2] S. Horwitz, J. Prins, and Thomas Reps, "On the adequacy of program dependence graphs for representing programs," *Conference Record of the 15'th ACM Symposium on Principles of Programming Languages,* Jan. 1988.

[3] S. Horwitz, Thomas Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transaction on Programming Languages and Systems,"* Jan. 1990.

[4] D. Jackson and E. J. Rollins, "A new model of program dependence for reverse engineering," *Proceedings of the 2nd ACM SIGSOFT Conference on Foundations of Software Engineering,* Dec. 1994.

[5] A. Krishnaswamy, "Program Slicing: An Application of Object-oriented Program Dependency Graphs," *Technical Report TR94-108,* Clemson University, 1994.

[6] L. Lasen and M. J. Harrold, "Slicing Object-Oriented Software," *Proceedings of the 18th International Conference on Software Engineering,* May 1996.

[7] M. Lorenz and J. Kidd, *Object-Oriented Software Metrics : A Practical Guide,* PTR Prentice Hall, 1994.

[8] I. Bashir and A. L. Goel, "Testing C++ Classes," *Proceedings of the 1st International Conference on Software Testing, Reliability,* and Quality Assurance, 1994

[9] M. J. Harrold and G. Rothermel, "Performing dataflow testing on classes," *Proceedings of the 2nd ACM SIGSOFT Symposium on the Foundation of Software Engineering,* Dec. 1994.

[10] H. Kim and C. Wu, "A Class Testing Technique Based on Data Bindings," *Proceedings of '96 Asia-Pacific Software Engineering Conference,* Dec. 1996.

[11] M. Smith and D. Robson, "Object-Oriented Programming - the Problems of Validation," *Proceedings of Conference on Software Maintenance,* 1990.

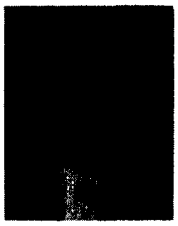
[12] A. S. Parrish, R. B. Borie, and D. W. Cordes, "Automated Flow Graph-Based Testing of Object-Oriented Software Modules," *Journal of Systems Software,* Nov. 1993.

[13] D. E. Perry and G. E. Kaiser, "Adequate Testing and Object-Oriented Programming," Journal of Object-Oriented Programming, Jan. 1990.

[14] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the Effectiveness of Dataflow- and Controlflow-Based Test Adquacy Criteria," *Proceedings of the 16th International Conference on Software Engineering,* May 1994.

[15] M. Ross, C. A. Brebbia, G. Staples, and J. Stapleton, "The Problematics of Testing Object-Oriented Software," *SQM'94,* vol. 2, July 1994.

◐ 저 자 소 개 ◑



임 동 주
1985년 전남대학교 영문학과(문학사)
1993년 뉴욕주립대학교 전산학과 이학석사
2000년 조선대학교 전산통계학 이학박사
※관심분야: 객체지향모델링, 멀티미디어시스템, 컴퓨터 통신

배 상 현
1982년 조선대학교 전기공학과졸업
1984년 조선대학교 전기 · 전자공학과 졸업(공학석사)
1988년 일본동경도립대학 정보공학과 졸업(공학박사)
1988년~현재 조선대학교 자연과학대학 전산통계학과 교수
※관심분야: 대규모지식베이스, 인공신경망, 퍼지 시스템, GIS, 전문가시스템