韓國 OA學會 論文誌
第5卷 第2號, 2000. 6.

2000-5-2-1-2

객체지향 프로그램의 클래스 테스팅

임 동 주*, 최 호 진**

Class testing of Object-oriented program

Dong-Ju Im*, Ho-Jin Choi**

요 약

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

I 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.

* 조선대학교 전산통계학과 박사과정수료
** 조선대학교 전산통계학과 박사과정

Ⅰ. Introduction

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[1][2][3].

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[4][9]. 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. The test cases for a class testing are a series of the method combination, and a class testing is the validation of what effect is made on an object state[1].[5]

Ⅱ. Test adequacy criterion

All the cases of the call of the methods in an arbitrary order cannot be tested. Therefore, the significant sequence of the methods satisfying the adequacy criterion of the test cases should be tested[8]. The adequacy criterion of the test cases is a selection criterion, and the criterion means that the testing with the generation of the test cases satisfying the criterion brings the access to the precise program. The test criterion and level of an object-oriented program researched so far is studied in this section.

The adequacy criterion of an object-oriented program so far is concerned with a few of the axioms Perry and Keiser studied[7].

The criterion satisfying the control and data flow Parrish et al proposed on the basis of a class flow graph is as follows[6].

- all-node coverage

All the methods represented in a flow graph are tested once or more.

- all-edge coverage

All the edges represented in a flow graph are tested once or more.

- all-definition coverage

The paths between all the definitions represented in a flow graph and the uses reachable from them are tested once or more.

- all-use coverage

The paths between all the definitions represented in a flow graph and all the reachable uses are tested once or more.

- all-definition/use coverage

All the paths between all the definitions represented in a flow graph and all the reachable uses are tested once or more.

The level of a class testing is identified as follows, and also the consideration is divided into the exactitude of a single data member only and the inter-operation between data.

· Intra-Method Testing

Individual method is a condensed function, the level of which corresponds to the unit testing of a conventional software testing. Therefore, the conventional functional or structural testing technique can be applied, and the definition-use path within a single method is tested.

· Inter-Method Testing

The use-definition path is tested including the other method called explicitly from a method. The testing corresponds to the integration testing of a conventional software testing.

· Intra-Class Testing

The class structure of an object-oriented program has not an imperative, but a declarative property which is executed in an arbitrary order. That is, the data members defined within a class influence the methods within the class as the global variables do, and so the data dependence by data members can occur without a direct call between the methods. Therefore, the definition-use path caused by a significant method call which is not a method call in a predetermined order should be tested.

· Inter-Class Testing

The methods defined in classes can operate mutually between the methods not only in a single class but also in some classes. For example, if the data members are inherited from the upper class, the mutual operation of the methods between the classes through the data members occurs. Consequently, the definition-use path caused by a significant method call between the classes is tested.

Ⅲ. 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

Procedure 1. The unit and integration testing of the method

In procedure 1, a method unit and method integration testing are performed with a conventional functional testing or structural testing applied.

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 testing of procedure 1 is not different from the previous testing. However, consider the following.

When the method defined in a class is called and executed, the behavior of the method operates mutually with other methods by redefining the data members on the basis of the data members of the object receiving the message. When the exactitude of a class consists in the object state, that is, the combination of the data members, the unit testing of individual methods and method integration testing performed in procedure 1 are not enough to detect the errors concerned with the definition and use of the data members. Therefore, a class level testing process is needed.

What is important in procedure 1 is the testing of the constructor. The constructor initializes all the data members, which makes an object level testing possible.

The capsularization of an object oriented concept can be thought to reduce the possibility of the erroneous mutual operation between the modules of an object-oriented program. The anticomposition axiom of Perry and Keiser's test case selection criterion[7] asserted the that the proper independent testing of the individual program elements did not guarantee the proper testing of the integrated program. Therefore, a class level testing should be performed along with a method level testing. A class level testing is to validate if the object state treated by the methods called in a significant order is correct. The object state is the combination of the values the data members have, and its validation is to check if the values of the data members are exact. The unit testing of the data members is performed in procedure 2. The slice concerned with a class hierarchy slicing of each data member is identified and tested in this research. The process means testing if the methods operating mutually through the data members deal with the data members exactly. To begin with, the definitions needed for the explanation are described.

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 D0(d), and the methods marked second in the continual traversal become the elements of D1(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 d1 as level-1 does not define d1 within it directly, it makes an indirect effect on d1 by defining d2 influencing the definition of d1.

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).

[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

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. For example, as D0(di) is the set defining di as level-0 from Def. 5.1, the value of Def_M(i,i) is d0. If the value of Def_M(i,j) is dk, the method set D0(dj) defining dj as level-0 makes an indirect effect on a data member di.

A class slice of a data member d is composed of a constructor and the method set defining and using d. In procedure 2 performing the unit testing of a data member, i=0 is defined, and in procedure 3 performing the integration testing of a data member, i≥0 is defined. Procedure 2 and 3 are described below.

	$D_0(d_1)$	$D_0(d_2)$	···	$D_0(d_i)$	···	$D_0(d_n)$
d_1	d_0		···			
d_2		d_0	···			
⋮			···			
d_i				d_k		
⋮						
d_n			···			d_0

[Definition 4] Class slice of data member d

Slice(C, d) = (Constructor, $\cup D_i(d)$, $U_0(d)$)

Procedure 2. Unit testing of data member

A class is divided into the slices concerned with each data member in order to generate the test case validating if each data member is treated exactly. Therefore, a single class is divided into the slices as many as the number of the data members to be used in the class. The definition set of a slice is composed of a direct definition set(Di(d), i=0) only.

In general, the analysis of the control and data flow in a program code is used for the information analysis for the generation of the test case in an implementation-based program testing. A data flow analysis is to trace the use of the variables after their definition through the execution flow path, and the trace degree of a definition-use path depends on the selection criterion of the test case based on a data flow. The test cases mean the scenario tracing the definition-use relationship of the data members. The approach for a class testing is to perform the method sequences in different many orders. Therefore, the testing order of the methods can be defined based on the definition-use relationship of the data members.

In order to apply the data-flow coverage criterion using a class flow graph, each slice of the classes should define a flow graph. The pairs of all the methods have a control edge. That is, all the methods are assumed to be called by an arbitrary method.

[Definition 5] All-use coverage criterion(data member unit)

If a test case T includes a method sequence $<i, n_1, ..., n_m, j>$ ($m \geq 0$, $n_i \notin D_0(d)$) for all $i \in D_0(d)$ and all $j \in U_0(d)$, T satisfies all-use coverage criterion for d.

Consequently, each data member is assumed to be independent, the test case generated in procedure 2 is the suitable test case satisfying all-use coverage criterion for each data member. In general, however, because the dependence between the data members exists, procedure 3 is executed with an anticomposition axiom considered.

Procedure 3. Integration testing of data members

In terms of an anticomposition axiom, the exactitude after its integration should be revalidated, even though each element is tested properly. The dependence between the data members of which the objects consist exists, but the unit testing of the data members performed in procedure 2 does not consider the dependence between the data members. Consequently, in order to test a class properly in an object level, an object unit testing, that is, the integration testing of the data members considering the inter-data dependence between them caused by the use-definition relationship is needed.

As in procedure 2, in procedure 3 a class is divided into the slices concerned with the data members, and each slice is tested one by one. The definition set of a slice includes an indirect definition set($D_i(d)$, $i \geq 0$), for the inter-data dependence between the data members should be considered. The test cases are generated with the application of the data-flow coverage criterion to the definition-use path of an object unit. The definition or use of a data member means the definition or use of an object. Therefore, unlike Def. 5 considering only the definition-use path of the same data members, the definition-use of an object unit should consider the definition-use path of the other data members, and it is defined as follows.

Ⅳ. Conclusion

The testing of object-oriented program has begun to develop, most of the previous research is a specification-based testing based on the abstract data type, and the research on an implementation-based testing is very weak. In this thesis, 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.

References

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

[2] 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.

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

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

[5] S. Zweben, W. Heym, and J. Kimich, "Systematic Testing of Data Abstractions Based on Software Specifications," Journal of Software Testing, Verification, and Reliability, 1992.

[6] 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.

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

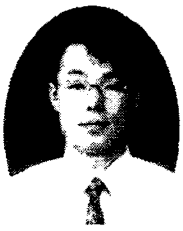
[8] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the Effectiveness of Dataflow-and Controlflow -Based Test Adquacy Criteria," Proceedings of the 16th International Conference on Software Engineering, May 1994.

[9] 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)
1992년 Concordia Univ 전산학 Graduate Diploma 수료
1993년 SUNY at New Paltz 전산학과(이학석사)
현재 조선대학교 전산통계학과 박사과정 수료
관심분야 : 객체지향모델링, 시스템 분석, 역공학



최 호 진

1996년 광주대학교 전자계산학과(공학사)
1998년 조선대학교 전산통계학과(이학석사)
1999년 ~현재 조선대학교 전산통계학과 박사과정
관심분야 : 객체지향 , 정보통신, 데이터베이스