Test Order for Inter-Class Integration Testing of Object-Oriented Software

Kuo-Chung Tai
Dept. of Computer Science
North Carolina State University
Raleigh, NC 27695-8206
kct@csc.ncsu.edu

Fonda J. Daniels
Dept. of Electrical and Computer Engineering
North Carolina State University
Raleigh, NC 27695-7911
fjdaniel@eos.ncsu.edu

Abstract

One major problem in inter-class integration testing of object-oriented software is to determine the order in which classes are tested. This test order, referred to as inter-class test order, is important since it affects the order in which classes are developed, the use of test stubs and drivers for classes, and the preparation of test cases. This paper first proposes a number of desirable properties for inter-class test order and then presents a new inter-class test order strategy. In this new strategy, classes are integrated according to their major and minor level numbers. Major level numbers of classes are determined according to inheritance and aggregation relations between classes, where an aggregation relation refers to a class' inclusion of objects of another class. For classes with the same major level number, their minor level numbers are determined according to association relations between these classes, where an association relation refers to a class' dependency (other than inheritance and aggregation relations) on another class.

1. Introduction

In recent years the use of object-oriented (OO) analysis, design and programming has increased significantly. One important issue in OO programming is how to test OO software. Four levels of testing OO software were suggested [11], [2]: unit (or method level) testing, intra-class (or class level) testing, inter-class (or cluster level) testing, and system testing. Unit testing focuses on the testing of individual methods in a class. Intra-class testing refers to the testing of the interactions among methods encapsulated within a single class [12], [6], [9], [3]. Inter-class testing refers to the testing of a group of classes that interface with each other [2], [5], [7], [10]. System testing ensures that all classes in a system mesh correctly and the overall system performance is achieved.

Intra-class and inter-class testing are different forms of integration testing of OO programs. There are major differences between integration testing of OO programs and that of traditional (or non-OO) programs. For a traditional program, the "call" relation between modules of the program is used as the basis for integration testing of these modules. Various integration strategies, including top-down and bottom-up integration, were defined [8]. For an OO program, however, the call-based integration testing approach cannot be applied directly. First, methods in a class often interact with each other through shared variables. Second, the call relation is not the only type of relation between classes.

One major problem in inter-class integration testing is the order in which classes are tested. This test order, referred to as **inter-class test order**, is important for several reasons. First, this test order affects the order in which classes are developed. Second, inter-class test order impacts the use of test stubs and drivers for classes and the preparation of test cases. Third, inter-class test order determines the order in which inter-class faults are detected.

In this paper, a new strategy for inter-class test order is proposed. The structure of this paper is as follows. The remainder of this section provides basic definitions to be used in this paper. Section 2 shows a number of desirable properties for inter-class test order. Our new inter-class test order strategy consists of two parts, which are presented in Sections 3 and 4 respectively. Section 5 discusses related work. Section 6 concludes this paper. Intra-class integration testing and test generation for inter-class integration testing are not discussed in this paper.

In [7], three types of relations between classes were identified: inheritance, aggregation, and association. Also, the notion of **object relation diagram** (ORD) was defined to represent such relations between classes in an OO

This work was supported in part by NSF grant CCR-9320992 and a NASA fellowship.

program. An ORD is a directed graph (or digraph) with each node denoting a class and each edge a relation. An edge from node N to node N′ is denoted as N-to-N′, and this edge has N and N′ as its head and tail nodes, respectively. Inheritance permits a subclass to inherit attributes from its parent class and either extend, restrict or redefine these attributes. In an ORD, an inheritance edge from class C1 to class C2 indicates that C1 is a subclass of C2. Aggregation refers to a class′ inclusion of objects of another class. In an ORD, an aggregation edge from class C1 to class C2 indicates that C1 contains objects of C2. An association relation refers to a class′ other dependency on another class. In an ORD, an association edge from class C1 to class C2 indicates that C1 accesses some data of C2, C1 invokes some methods of C2, or some objects of C2 are parameters of methods in C1. In an ORD, inheritance and aggregation relations do not form cycles. However, an ORD may contain cycles due to association edges.

Fig. 1 shows an ORD, called G1, which contains a set of 15 classes in the InterViews library [7]. In this ORD, inheritance, aggregation, and association edges are labeled with "I", "Ag", and "As", respectively. The following abbreviations are used for the names of these 15 classes: B (for Button), BL (for ButtonList), BS (for ButtonState), C (for Control), CS (for ControlState), CV (for Canvas), CR (for CanvasRep), E (for Event), II (for InteractorItr), MS (for MonoScene), S (for Subject), SC (for Scene), SS (for Sensor), TB (for TextButton), and W (for World).

In this paper, we assume that testing a class C for inter-class integration involves the following two steps:

(a) Test each outgoing edge of C at least once. (How to test an edge is not discussed in this paper.) If C has an outgoing edge to class C′ that has not been integrated yet, then a stub for C′ is used for testing this edge.
(b) For each incoming edge of C from a class that has been integrated, retest this edge at least once. Such an edge was tested earlier using a stub for C. Resting such an edge is needed since a stub for C is a simplified version of C.

2. Desirable properties for inter-class test order

If class C has an outgoing inheritance or aggregation edge to class C′, then C′ should be tested before C for inter-class integration. Since inheritance and aggregation relations between classes do not form cycles, the following property for inter-class test order is desirable:

Property 1: For classes C and C′ in an ORD, if there exists a directed path from C to C′ such that the path contains inheritance and aggregation edges only, then C′ is tested before C for inter-class integration.

The above property can be extended by allowing association edges:

Property 2: For classes C and C′ in an ORD, if there exists a directed path from C to C′ and no directed paths from C′ to C, then C′ is tested before C for inter-class integration.

An ORD contains cycles only if association edges exist. For an ORD with cycles, we can remove some association edges to break all cycles and then produce a test order. However, two new issues arise. The first issue is about the creation of stubs for classes that are needed for testing, but have not been integrated yet. The number of stubs needed depends upon the choice of deleted association edges. Thus, the following property is desirable:

Property 3: For an ORD with cycles, minimize the number of stubs needed for inter-class integration testing.

How to find a test order to satisfy property 3 is unknown yet. One idea is to delete a minimum number of association edges in order to break all cycles in an ORD. However, finding a minimum set of edges in a cyclic digraph for deletion in order to produce an acyclic digraph is an NP-complete problem [4]. Furthermore, deleting a minimum number of association edges for breaking all cycles does not necessarily imply finding a test order that requires a minimum number of stubs. Thus, a practical version of property 3 is the following:

Property 4: For an ORD with cycles, reduce the number of stubs needed for inter-class integration testing at reasonable cost.

The second issue occurs when a class C whose stub was used earlier is being tested for inter-class integration. As mentioned earlier, the incoming association edges of C from classes that have been integrated need to be retested. In what order should these association edges be tested? Assume that both classes C′ and C″ have outgoing association edges to C and have been integrated already such that C″ was integrated before C′. The association edge from C″ to C should be retested before that from C′ to C, since there may exist a directed path from C′ and C″ that contains inheritance and aggregation edges. Thus, we have the following desirable property:

Property 5: Assume that classes C′ and C″ have outgoing association edges to C, C″ is tested for inter-class integration before C′, and C′ is tested for inter-class integration before C. When C is being tested for inter-class integration, the association edge from C″ to C should be retested before that from C′ to C.

3. Assigning level numbers to classes

Now we describe a new inter-class test order strategy that satisfies properties 1, 2, 4 and 5. Our strategy consists of two parts. In the first part, which is shown in this section,

each class in an ORD is assigned a level number. In the second part, which is shown in Section 4, how to perform inter-class integration testing according to level numbers of classes is described.

The assignment of level numbers to classes in an ORD involves the following two steps:

(1) Assign each class a major level number according to inheritance and aggregation relations between classes.
(2) For classes with the same major level number, assign each class a minor level number according to association relations between these classes.

A class is said to have level number i.j, if its major and minor level numbers are i and j respectively. Details of steps (1) and (2) are given in Sections 3.1 and 3.2 respectively. Both steps use algorithm Level_ADG to assign level numbers to nodes in an acyclic digraph. Below we first give a definition of the level number of a node in an acyclic digraph and then show the algorithm itself.

For an acyclic digraph, let Lev(i), i>0, be the set of nodes in the digraph with level number being i.

Lev(1) = { nodes without outgoing edges }.

For i>1, Lev(i) = { nodes with each outgoing edge to a node in Lev(j), j < i, and with at least one outgoing edge to a node in Lev(i-1) }.

Assignment of level numbers to nodes in an acylic digraph can be performed during a depth-first search of the digraph. When the search returns from a node n, if node n has no outgoing edges, then its level number is 1. Otherwise, the level number of node n is (1 + the maximum of level numbers of nodes reached by outgoing edges of node n). Below is an algorithm for assigning level numbers.

Algorithm Level_ADG.

```
Let N_ADG be the set of nodes in an acyclic digraph.
for each node n in N_ADG, let
  mark(n) indicate whether node n has been visited or not,
  suc(n) be the set of nodes reached by outgoing edges of
  node n,
  level(n) be the level number of node n, and
  suc_level(n) be the set of level numbers of nodes in
  suc(n).
for each node n in N_ADG   { mark(n) = unvisited };
for each node n in N_ADG
    { if mark(n) = unvisited then Level_Sort(n) };
procedure Level_Sort(n)
{ mark(n) = visited;
  if |suc(n)| = 0
  then   level(n) = 1;
  else   { for each node n' in suc(n)
             if mark(n') = unvisited then Level_Sort(n');
           level(n) = max(suc_level(n)) + 1;
         }
}
```

3.1 Assigning major level numbers to classes

For an ORD, let Major_Lev(i), i>0, be defined as follows.

Major_ Lev(1) = { classes without outgoing inheritance or aggregation edges }

For i>1, Major_Lev(i) = { classes with each outgoing inheritance or aggregation edge to a class in Major_Lev(j), j<i, and with at least one outgoing inheritance or aggregation edge to a class in Major_Lev(i-1) }

A class in Major_Lev(i), i> 0, has i as its major level number and is said to be in major level i. Assignment of major level numbers to classes in an ORD G can be done by applying algorithm Level_ADG to an ORD G', where G' is G modified by removing all association edges.

Fig. 2 shows the major level diagram for ORD G1 without association edges. In this diagram, major level i, i > 0, contains classes with their major level number being i.

3.2 Assigning minor level numbers to classes

Now we consider how to assign minor level numbers to classes in the same major level. For an ORD G, let ML(G,i) be the subgraph of G that contains only the classes in major level i and the association edges between these classes. (There are no inheritance or aggregation edges between classes in the same major level.) For each class in ML(G,i) that has j as its minor level number, its level number is i.j.

ML(G,i) has the following three cases :

(1) ML(G,i) contains only one class. The minor level number of this class in 1.
(2) ML(G,i) contains two or more classes and does not contain cycles. In this case, apply algorithm Level_ADG to ML(G,i) to assign each class in ML(G,i) a minor level number.
(3) ML(G,i) contains two or more classes and also contains cycles. In this case, perform the following steps:
    (3.1) Identify strongly connected components in ML(G,i).
    (3.2) For each strongly connected component in ML(G,i), remove some edges to break cycles.
    (3.3) Apply algorithm Level_ADG to the modified ML(G,i) to assign each class a minor level number.

In step (3.2), how to delete edges in a strongly connected component for breaking cycles affects the number of stubs needed for inter-class integration testing. As mentioned in Section 2, deleting a minimum number of edges for breaking cycles is not practical. To reduce the number of stubs needed, we use the following two steps to delete edges in a strongly connected component SC in ML(G,i).

(a) Let Stubs(SC) be the set of classes in SC that have incoming association edges from classes with their major level numbers less than i. When classes in major level i are

tested, stubs for classes in Stubs(SC) already exist since they were used for testing classes in major levels 1, 2, ..., and (i-1). If an edge to a class C in Stubs(SC) is deleted for breaking cycles in SC, there is no need to create a stub for C, since this stub already exists. Thus, incoming edges of classes in Stubs(SC) from other classes in SC should be considered first for deletion. Let Edges_to_Stubs(SC) be the set of incoming edges of classes in Stubs(SC) from other classes in SC. For edges in Edges_to_Stubs(SC), delete one at a time from SC until all cycles in SC disappear. After all edges in Edges_to_Stubs(SC) have been deleted, if SC still contains cycles, go to step (b).

(b) To delete more edges in SC to break cycles, assign each edge e in SC a value, called weight(e), which is defined as the sum of the number of incoming edges of the head node of e and the number of outgoing edges of the tail node of e. If an edge in SC has a higher weight value, then it has a higher chance of breaking more cycles in SC. Edges in SC are deleted one at a time in decreasing weight value, until all cycles in SC disappear. (After the deletion of an edge in SC, the weight values of other edges in SC may change. Thus, an improvement of step (b) is to recompute the weight values of the remaining edges in SC after each deletion of an edge. But this improvement is time-consuming.)

In step (3.3), two or more classes in ML(G,i) that are connected by association edges deleted in step (3.2) may be assigned the same minor level number. What is the test order for classes with the same level number? Since the number of classes with the same level number is usually small, a practical solution is to perform inter-class integration of these classes at the same time. By doing so, no stub for any of these classes is needed.

Fig. 3 shows the major level diagram for G1 with each class associated with its level number. In Fig. 3, major level 1 contains a strongly connected component consisting of classes II and S. Both edges II-to-S and S-to-II have the same weight value. The edge II-to-S is deleted to break the cycle and thus is shown as a dashed line in Fig. 3. Other major levels do not contain strongly connected components. Note that in our strategy, there is no need to break cycles involving classes in different major levels, since the test order among these classes is determined by their major level numbers.

4. Inter-class integration based on level numbers

Now we consider how to use the level numbers of classes to determine the order in which inter-class integration testing is performed. For two level numbers i.j and u.v, i.j < u.v if either i<u or i = u and j<v. Basically, classes are integrated in increasing order of their level numbers. However, two issues need to be addressed. One is the determination of stubs needed and the other is the satisfaction of property 5.

For an ORD, let Major_Max be the maximum major level number, and for $1 \le i \le$ Major_Max, let Minor_Max(i) be the maximum minor level number in major level i.

```
for i = 1 to Major_Max
  for j = 1 to Minor_Max(i)
  { If level i.j contains two or more classes connected by
    association edges, integrate these classes at the same
    time;
    for each class C in level i.j
    { for each edge from C to another class C' with level
        number larger than i.j, if no stub for class C'
        exists, create a stub for C'.
      Test C for each of its outgoing edges;
      Let Use_Stub(C) be the set of classes that have their
        level numbers less than i.j and have outgoing
        edges to C;
      Let m be the size of Use_Stub(C) and let L be a list
        of the classes in Use_Stub(C) in increasing order
        of level numbers;
      for k = 1 to m, retest the kth class in L for its
        outgoing edge to C.
    }
  }
```

Note that the set of stubs needed and the Use_Stub set for each class can be determined before the start of inter-class integration testing.

By applying the above inter-class integration algorithm to the ORD in Fig. 3, test stubs are needed for S (used by II), B (used by BL), W (used by E), and C (used by CS). When S is being tested, the edge from II to S is retested. Also, when B is being tested, the edge from BL to B is retested.

5. Related work

In [7], Kung et al. studied several issues on regression testing of OO programs. They defined various types of code changes in classes, showed a method for identifying these changes and the affected classes in an OO program, described an algorithm for assigning a test order for the affected classes, and presented an algorithm for test order based regression testing. Although the test order assignment algorithm and the test order based regression testing algorithm are defined for the affected classes during regression testing, they can also be applied (with minor modifications) to all classes in an OO program for inter-class integration testing.

The test order assignment algorithm in [7] is the following, assuming that all classes in an ORD need to be tested for inter-class integration.

(1) If the ORD has cycles, transform the ORD into an

acyclic ORD by identifying strongly connected components and replacing each strongly connected component with a single node. The classes in each strongly connected component are referred to as a cluster of classes.

(2) Produce a test order, called the **major test order**, for nodes in the ORD by using an topological sorting algorithm. (A topological sort of an acyclic digraph is a linear ordering of all nodes in the digraph such that if there exists an edge from node U to node V, then U appears before V in the ordering [1]. How to use topological sorting to produce a test order is not explained in [7]. According to the examples in [7], the test order numbers assigned to nodes in an acyclic digraph are produced in the same way as algorithm Level_ADG in Section 3.)

(3) For each cluster, if it contains just one class, the minor order number of this class is not assigned. Otherwise, perform the following steps:

(3.1) Remove one or more association edges to break all cycles in the cluster.

(3.2) Apply topological sorting to the cluster to produce a test order, called the minor test order, for classes in the cluster.

For any ORD, the test order produced by the above algorithm satisfies properties 1 and 2. Note that if an ORD is modified by deleting or adding association edges, major test order numbers of classes may change. Property 3 (minimizing the number of stubs needed for integration testing) was mentioned in [7], but no effort was made to deal with property 3 or 4 (reducing the number of stubs needed for integration testing at reasonable cost).

The test order based integration testing algorithm in [7] integrates all classes with the same major order number at the same time and tests their outgoing edges to classes with smaller major order number. It does not use minor order numbers to determine the test order for classes with same major order number. Also, it does not discuss the use of stubs in integration testing and the retesting of association edges from already integrated classes to newly integrated classes. Thus, this integration testing algorithm does not satisfy property 5.

Assume that an association edge from BS to B is added to ORD G1. Then BS, B and BL form a strongly connected component. According to Kung et al.'s strategy, the major order number of B is changed and an additional test stub is needed. According to our strategy, changes of association edges do not affect major level numbers of classes. Now major level 2 contains a strongly connected component consisting of BS and B. Since a stub for B already exists, we delete the edge from BS to B to break the cycle. Thus, the minor level numbers of BS and B are unchanged and no additional stubs are needed.

In [10] Paradkar proposed a different inter-class test order strategy for an ORD. This test order strategy does not satisfy properties 1, 2 and 5 and does not deal with the reduction of stubs needed for inter-class integration testing.

6. Conclusions

In this paper, we have described a number of desirable properties for inter-class test order and presented a new inter-class test order strategy. Our inter-class test order strategy satisfies properties 1, 2, 4 and 5. The use of major level structure of classes in an ORD has several advantages. First, the major level structure provides a guidance for the order for design, implementation, and testing of classes Second, the major level structure is not affected by changes of association relations between classes. Third, if changes of inheritance and aggregation relations between classes are needed, the major level structure can be used to evaluate different alternatives for such changes.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1983.

[2] R. J. D'Souza and R. J. LeBlanc, "Class Testing by Examining Pointers", Journal of Object-Oriented Programming, pp. 33-39, July-August 1994.

[3] J. Gao, D. Kung, P. Hsia, Y. Toyoshima, and C. Chen, "Object State Testing for Object Oriented Programs", Proc. IEEE COMPSAC '95, pp. 232-238, 1995.

[4] F. Gavril, "Some NP-Complete Problems On Graphs", Proc. 1977 Conf. on Information Sciences and Systems, pp. 91-95, April 1977.

[5] P.C. Jorgensen and C. Erickson, "Object Oriented Integration Testing", Comm. of the ACM, Vol, 37, Number 9, pp. 31 - 38, Sept. 1994.

[6] S. Kirani and W. T. Tsai, "Method Sequence Specification and Verification of Classes", Journal of Object Oriented Programming, pp. 28-38, Oct. 1994.

[7] D. Kung, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen, "On Regression Testing of Object Oriented Programs", Journal of Systems Software, Vol 32, pp. 21-40, Jan. 1996.

[8] G. J. Myer, *The Art of Software Testing*, John J. Wiley, New York 1979.

[9] A. S. Parrish and D. W. Cordes. "Applying Conventional Unit Testing Techniques to Abstract Data Type Operations", Journal of Software Engineering and Knowledge Engineering, Vol. 4, No 1, pp. 103-122, Jan. 1994.

[10] A. Paradkar, "Inter-class Testing of O-O Software in the Presence of Polymorphism," Proc. CASCON'96, pp. 137-146, 1996.

[11] M. D. Smith and D. J. Robson, "A Framework for Testing Object-Oriented Programs", Journal of Object-Oriented Programming, pp. 45-53, June 1992.

[12] C. D. Turner and D. J. Robson, "The State-Based Testing of Object-Oriented Programs," Proc. IEEE Conf. Software Maintenance, pp. 302-310, 1993.

Figure 1. Object relation diagram G1

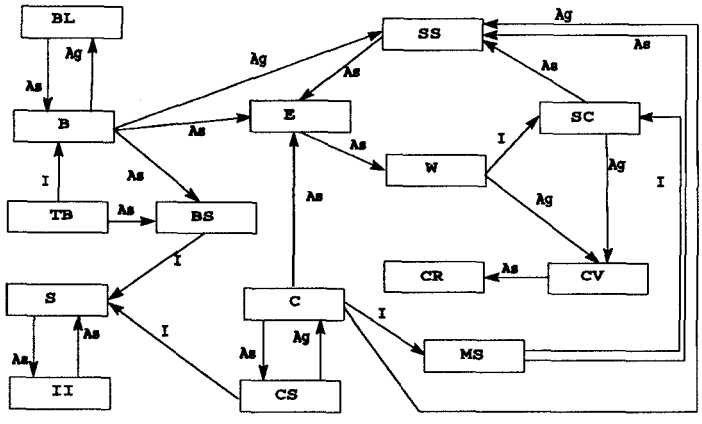

Figure 2. Major level diagram for G1 without association edges

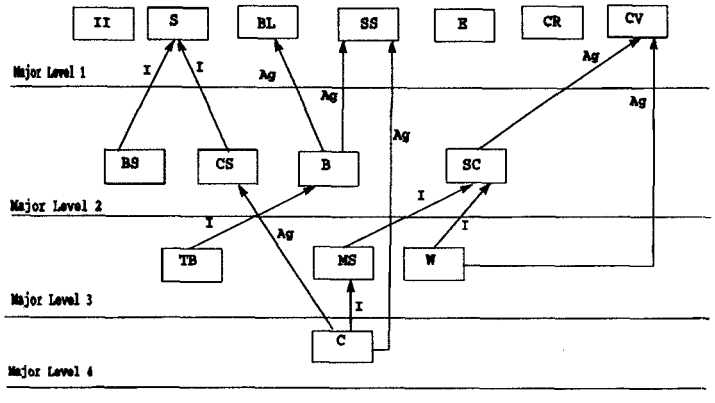

Figure 3. Major level diagram for G1 with level numbers