3

A Framework for Distributed Object-Oriented Testing

*Alan C.Y. Wong, Samuel T. Chanson, S.C. Cheung and Holger Fuchs**

Department of Computer Science, Hong Kong University of Science and Technology Clear Water Bay, Hong Kong

Abstract

Distributed programming and object-oriented programming are two popular programming paradigms. The former is driven by advances in networking technology whereas the latter provides vigorous software principles needed in developing complex software systems. While more and more distributed object-oriented software has appeared, not much work exists on the testing of these systems in an integrated manner. Instead, the distributed and object features have been tested separately. In this paper, we propose an integrated framework known as DOOT for incremental testing of distributed object-oriented software systems. It combines various testing techniques to provide comprehensive test coverage at four levels - class testing, intra-cluster testing, inter-cluster testing and system testing. Each level uses a specific fault model, test strategy and test case generation that build on the previous test level to reduce the overall test effort. They are designed to handle the distinct requirements of the two paradigms at each level. Moreover, a reduction algorithm for testing the inherited class is also included in the framework. The approach is illustrated using a real life example of a conferencing system.

1. INTRODUCTION

The popularity of the Internet coupled with advances in local area network and high speed network technologies have introduced many new distributed applications. Object-oriented techniques are often used to cope with the complexity of developing these software systems which have come to be known as distributed object-oriented software systems (DOOSS). Like other software these systems must be thoroughly tested before use. Software testing deals with checking the correctness of the implementation against its formal specification [10, 20]. Much work has been done on verifying [13] or testing [6, 7, 9, 15, 16, 19, 23, 24] object-oriented software. However, the validation [8] or testing [11,

---

* Holger Fuchs is currently with the Brandenburg University of Technology at Cottbus, Germany.

Formal Description Techniques and Protocol Specification, Testing and Verification
T. Mizuno, N. Shiratori, T. Higashino & A. Togashi (Eds.) © 1997 IFIP. Published by Chapman & Hall

12, 25, 26, 27] of distributed systems have received less attention. Moreover, few existing work has specifically addressed the issue of combining the two types of testing methodologies in testing DOOSS.

This paper reports a framework for distributed object-oriented testing known as DOOT. The objective is to integrate the testing techniques for distributed systems and object-oriented software into an integrated framework while minimizing the testing effort. DOOT unifies the results at different test levels and reduces the global test space to a manageable size. The test levels are class testing, intra-cluster testing, inter-cluster testing and system testing. Each level has its own fault model, test strategy and test case generation scheme that addresses the specific requirements of the distributed and object paradigms at their level of abstraction. Each level utilises the test results from the previous levels so that the overall test effort is minimized.

The rest of this paper is organised as follows. Section 2 describes the general structure of DOOSS and their advantages as well as the problems in testing them. The DOOT framework is outlined in Section 3. Sections 4, 5, 6 and 7 present the details of each level of testing, namely class testing, intra-cluster testing, inter-cluster testing and system testing respectively. Section 8 concludes the paper, and the application of our approach is illustrated using a conferencing system in the Appendix.

2. DISTRIBUTED OBJECT-ORIENTED SOFTWARE SYSTEMS

Due to the many useful software construction features such as encapsulation, abstraction, inheritance, genericity and dynamic binding, the object model is widely adopted in both academic research and in the industry in recent years. These features also help to promote software reusability, maintainability, reliability and performance [5]. Distributed systems have also become increasingly common as more and more organisations use networks to share resources, enhance communication and increase performance. Examples of these systems range from the Internet, to workstations in a local area network within a building, to processors within a single multiprocessor [14]. They are characterised by the presence of independent activities, loosely coupled parallelism, heterogeneous software and hardware.

Increasingly, the object model and distributed technologies are being amalgamated [1, 4]. The advantage is obvious: the complexity and dependencies of the entities can make use of the object model in a distributed system to break down the intensive design process into efficient constructs. Moreover, the techniques contributed by CORBA, OLE and WWW have greatly promoted the approach of distributed object technology [22].

However, DOOSS presents a major challenge for testing and maintenance. Many problems known to object-oriented systems are compounded in the distributed system environment, especially those related to concurrency. Some solutions have been proposed in each area individually. In general, these approaches adopt traditional testing and analytic techniques that work well in either sequential programs [10, 23] or communication protocols [8, 12]. The major problem of state space explosion in DOOSS remains unsolved. Moreover, the solutions for object-oriented testing are not handled satisfactorily. These include the efficient use of inheritance to reduce test effort for derived classes.

Our approach combines existing analysis and test techniques with new solutions specifically oriented towards a new set of conditions and requirements in DOOSS. The selected test cases result in an acceptable level of confidence on the correctness of our case study implementation (see Appendix).

3. THE FRAMEWORK OVERVIEW

The DOOT framework distinctly separates the testing of object-oriented and distributed properties. To handle the various types of interactions found in distributed object-oriented software, DOOT is driven by four test levels, each associated with a different fault model and test strategy. Table 1 relates the four test levels to those commonly adopted in traditional testing approaches.

Traditional System	**Distributed Object-Oriented Software System**		
approaches	object-oriented properties		distributed properties
unit testing	*class testing*	*method testing* *object testing*	---
Integration testing	*intra-cluster testing*		*inter-cluster testing*
System testing	*System testing*		

Table 1. Classification of test Levels

Class testing comprises two procedures: method testing and object testing. The goal of class testing is to validate the class definition of an object. Method testing is essentially unit testing on each method in the class. During the test, a value table (VT) and a control flow diagram (CFG) are derived for each method. While method testing focuses on the details of each method, object testing examines the interactions among the member functions (methods) of a class. Integration testing is realised in two levels of testing: intra- and inter-cluster testing. A cluster is formed by a collection of objects executed within a single control thread. Intra-cluster testing constructs from the CFGs an execution tree (ET). While intra-cluster testing aims at checking the interaction among objects within a cluster, the inter-cluster testing focuses on the interactions between clusters. The last level of DOOT follows the traditional approach of system testing. Algorithm 1 presents an overview of the entire framework. The input and output elements will be described in subsequent sections.

```
1   input:   CLIB - class library for each cluster
2            CPC, FSM - pseudo-code and finite state machine for each class
3   output:  VT, CFG - value table and control flow graph for each method
4            LIG, ET - list of interaction graphs and execution tree per cluster
5            TT - Transaction tree of the system
6   ------------------------------------------------------------------------
7   LIG = empty;                          // initialise LIG
8   for every cluster do {                // for each single thread
9     for every concrete class in CLIB do {   // no testing for abstract classes
10      for every new method do {         // no redundant methodTesting
11        apply methodTesting(CPC);       // also produce VT and CFG (algorithm 2)
12        derive IG from CFG;             // form IG for each method (algorithm 3)
13        add IG to LIG;                  // to be used in intraClusterTesting
14      }
15      apply objectTesting(FSM,CFG);     // testing causal order of methods
16    }
17    apply intraClusterTesting(LIG);     // also generate ET (algorithm 4)
18  }
19  combine ET to TT;                     // produce the composite TT (algorithm 5)
20  apply interClusterTesting(TT);        // testing the cluster communications
21  apply systemTesting;                  // random walk and/or requirement testing
```

Algorithm 1. Distributed object-oriented testing

DOOT supports an integrated and incremental approach for software testing. It is an integrated approach because each level utilises the test results from the previous levels. For example, the CFGs derived from the pseudo-code are reused by object testing (lines 11 and 16 in Algorithm 1). Hence the overall test effort is reduced. DOOT is also an incremental framework since the set of elements tested in one level is considered as a basic unit of interaction in the next level. For example, object testing examines method interactions whereas intra-cluster testing uses objects as test units. This is an important concept of DOOT. First, the complexity of state space is reduced by this folding technique. Furthermore, given the well-tested paths in the lower level, the next level needs only transverse one of these paths for a comprehensive test coverage. Finally, the same or similar test modelling techniques can be employed at different levels of abstraction in the framework to improve understandability and reduce the complexity of the approach. Each level of testing will be explained in more detail in the following sections.

4. CLASS TESTING

In DOOT, a class is assumed not to contain internal concurrency or non-determinism[1]. Thus no special effort to deal with concurrency or non-determinism is made in class level testing which consists of two procedures: method testing and object testing. Method testing is a structural testing of individual class methods whose behaviours are specified in pseudo-code. While method testing focuses on the internal details of each method, object testing

[1] Current components within a class should be discouraged and replaced by multiple interacting objects.

validates the interaction among these methods against the allowable object behaviour. Like other object-oriented testing techniques [7, 23], the allowable object behaviour is assumed to be given in the form of finite state machine (FSM). Class testing is only the first step of our testing framework. The testing of class relationships is done in other steps in the framework. Class testing can also be further optimised to take advantage of the inheritance hierarchy of an object-oriented system which has been discuss in by others [16, 17, 24, 25]. Class testing aims at identifying the following types of faults:

- Data-anomalies such as misspelling, name confusion, deletion of statements and missing initialisation. These faults can be detected by static data flow analysis, compilation [2, 28].
- State errors associated with object behaviour. These include the selection of a wrong transition, invalid execution of a transition and missing transitions.
- Errors that are associated with object instantiation, such as memory allocation, invalid object name and missing initialisation of attributes.

4.1 Method Testing

Method testing comprises of analysing and collecting test information. The analytical phase scans the pseudo-code of each method, and generates a predicate-used [10] variable table VT with its values and its corresponding CFG. This information is collected for controlling method execution. Method testing is summarised in the following algorithm:

```
1   input:  CPC - pseudo-code for each class
2   output: VT, CFG - value table and control flow graph for each method
3   ------------------------------------------------------------------
4   for each new method in CPC do {
5     create a VT with all attributes and method variables;
6     fill in the VT with the values from the transition predicates;
7     create a CFG from the VT and label the predicates;
8     generate test cases based on the values in the VT;
9     execute every path of the CFG at least once;
10  }
```

Algorithm 2. Method testing

Let us consider a simple class *Account* (bank account) that consists of three attributes (*accountNo*, *customerName* and *balance*) and several methods (four of which are relevant to this discussion: *open*, *deposit*, *withdraw* and *close*). Figure 1 illustrates the mechanism of method testing using a segment of the *Account* class specification. Variables other than the three class attributes are formal parameters of the methods. As indicated in Algorithm 2, a VT and a CFG are generated for each method. The node column of each VT denotes the statement number and the variable column lists all variables accessed by that method. There are three types of values in VT. A *don't care* means the initial value of that variable is not important, e.g., *accountNo* in node 3 of the *open* method. A

random entry means a random value is provided for the variable in testing that method, e.g., a random integer for *accNo* in the same node. Lastly, each predicate generates a set of specific values. For example, two values of *acctNo* are provided in the *deposit* method: one is equal to *acccountNo* and the other does not. They are also labelled (*a* and *b* in this case) for identification. In this testing, a method is considered to be an imperative program and its interactions with other methods are treated as interaction to black boxes, e.g., *error(accNo)* in node 5 of the *deposit* method. Each VT is then transformed to a CFG where the predicates are shown as parameters associated with the transitions (arrows). The 'n_0' and 'n_e' symbols in the CFG denote the start and the end nodes respectively, whereas the labels reference the conditions given in the VT for the associated transitions to occur.

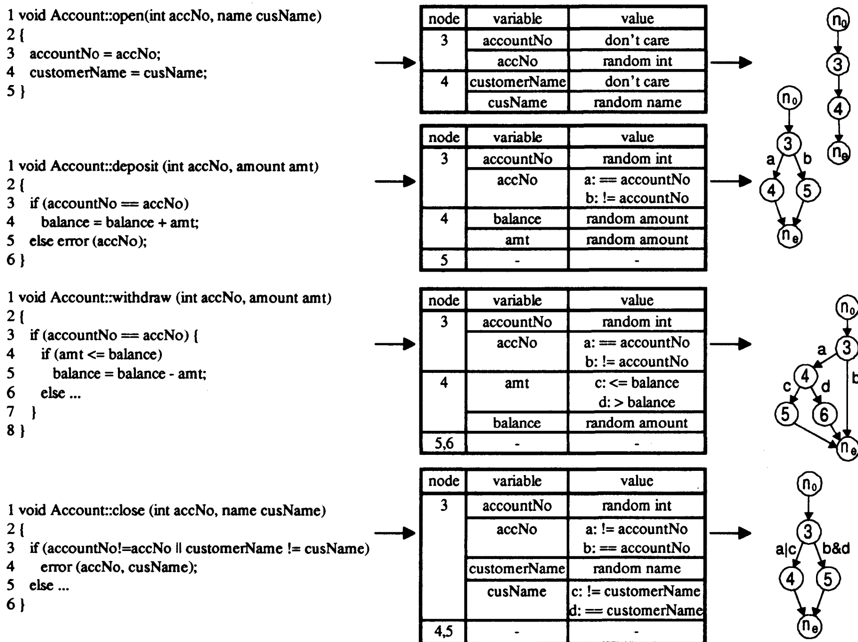


Figure 1. Construction of VT and CFG from class definition

The test coverage of method testing is governed by the CFGs of a class. Each predicate-use combination generates a separate test case. For instance, the three test cases of the *withdraw* method are given by the parameter set {*ac*, *ad*, *b*}. This test coverage guarantees that every specification statement of a method (or every control flow in a CFG) is executed at least once. The *close* method demonstrates the combined parameters, where the '|' and '&' stand for the 'or' and the 'and' boolean operator respectively. This method also generates three test cases which are defined by the set {*a*, *c*, *b&d*}.

4.2 Object Testing

The second part of class testing is object testing. The objective is to identify faults in the implementation that violate the predefined permissible object behaviour. The permissible object behaviour is specified in an FSM. Each path in the FSM describes a legitimate method execution sequence. A shortest test sequence is generated using transition tours (such as the Chinese postman algorithm) that covers all transitions in the automaton. Figure 2 illustrates object testing by showing an FSM of the *Account* class. The correct causal order of method invocations is examined. For example, the *open* method must be invoked before the first *deposit* method; and a *withdrawAll* method must be followed by a *close* method.

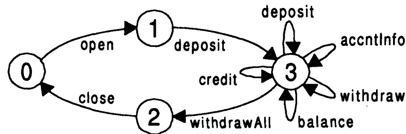


Figure 2. Finite state machine for the *Account* class

After a test sequence is generated, the values of test data are set to ensure the execution path is followed. This is achieved by using a simple static data flow analysis [18, 28]. For each invoked method in the sequence, the predicate-use values of each path in the CFG are examined. The largest matching parameter set is chosen for generating the test cases. For instance, in order to execute the sequence [*open deposit withdraw*] of the *Account* class in Figure 1, the three methods must use the same value of *accNo*. Therefore, while a random value is allowed in the *open* method, the parameter *a* of the *deposit* method must match that of the *withdraw* method. On the other hand, if no matching parameter is needed, the longest sequence is used and the unreachable transitions are marked for use in the generation of the next test sequence. This procedure is applied until all transitions in the FSM are covered. Give the test data to enforce the execution of the following sequence [*open deposit accntinfo deposit withdraw credit balance withdrawAll close*] in the Account class example.

5. INTRA-CLUSTER TESTING

A cluster is a collection of objects to be executed in a single thread. Intra-cluster testing focuses on finding faults related to object interaction within a cluster. The difficulties of this testing lie in the complexity and dynamic structure of a cluster [16, 19]. As such, traditional techniques like data flow and control flow testing are not adequate [23]. However, by combining several techniques, it is possible to reduce the complex structure of object interactions into something more manageable. DOOT tackles this by localising all dependencies of interactions in the cluster. Intra-cluster testing consists of two steps: static and dynamic

analyses. In static analysis, an interaction graph (IG) is constructed from each CFG derived in the method testing phase. The interaction graph of a method *m* shows all the methods that can be invoked within *m* and the conditions for their invocations. The IGs in the same cluster are then linked up to form an execution tree (ET). In dynamic analysis, test cases are generated by traversing the ET.

Static analysis searches for faults that are related to unreachable codes and attribute anomaly. The unreachable codes may reveal errors due to the dynamic binding feature of object-oriented systems. Data anomaly may occur within a single method or in attributes that are used by more than one method. For example, errors may exist when an attribute is used before being defined or is redefined before use. On the other hand, dynamic analysis concentrates on wrong method invocation and missing instantiation of objects. The former may be caused by confusion on method names, and the latter may occur if the type of the invoked method is incorrect or the object is not yet instantiated when called.

5.1 Static Analysis

From the predicate-used variables in the VT, the information on all method invocation is collected and specified in the CFG. Additional control flows (V-headed arrows) are inserted to indicate method dependence. The simplified model is a tree of depth one and is known as an IG. The root node of an IG contains the method name whereas the leaf nodes contain the names of invoked methods. An edge from the root node to a leaf node represents a method invocation. These leaf nodes are depicted from left to right in the order of sequential execution. Each edge may contain a guarded condition that has to be satisfied to invoke the corresponding method. The condition is obtained by a trace of parameters on the path defined in the CFG. Algorithm 3 shows the generation of a list of IGs within a cluster.

Figure 3 illustrates the transformation of three IGs (for methods *A*, *B* and *C*) from the models in class testing. The VTs only show the variables (*x*, *y*, *z*, *v*) that are used in the predicates. The labels of predicate (*a*, *b*, *c*, *d*) specify the conditions for which the corresponding transitions in the CFG will take place. An extra column is added to show possible method invocation. For instance, *m2(b)* in node 4 of method *B* indicates that *m2* is invoked if condition *b* holds. A grey node in the CFG depicts the point of method invocation. DOOT classifies the methods within a cluster into four mutually disjoint categories:

main method: every cluster has only one method of this type

entry method: an interaction point where calls between clusters are made, i.e., can be called from other clusters and may call other methods in the local cluster (e.g. method *C*)

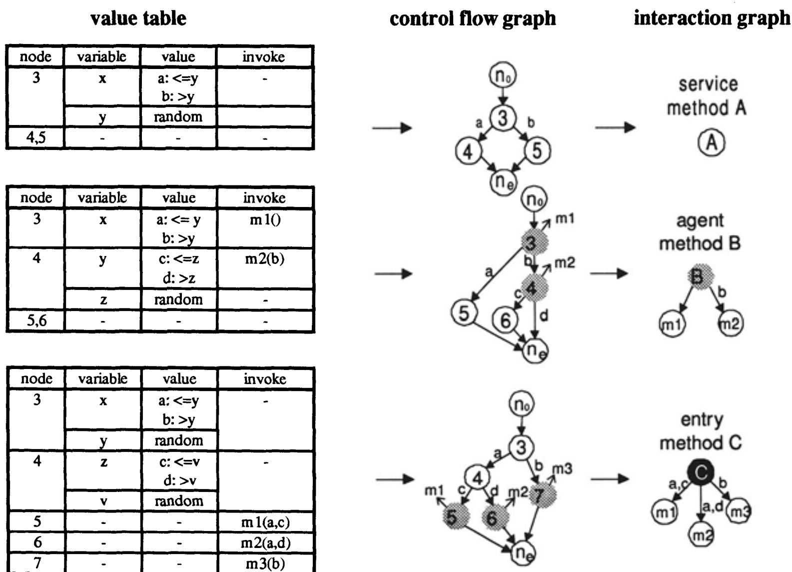
agent method: a non-entry method that may invoke other methods (e.g. method *B*)

service method: a non-entry method that does not invoke any methods (e.g. method *A*)

```
1   input:   CLIB - class library of the cluster
2            CPC, CFG - pseudo-code for each class and control flow graph per method
3   output:  LIG - list of interaction graphs
4   ---------------------------------------------------------------------------
5   for each class in CLIB do                       // each class in the cluster
6     for each method in CPC do {                   // each method in a class
7       create new root node to LIG;                // create a new IG
8       for each path in CFG do                     // obtained from method testing
9         while (not end of path) do {
10          follow the path;                        // ignore non-interaction statements
11          collect parameter on predicate-use;     // form trace of parameters
12          if (invocation of other method) {       // find new method invocation
13            create new node with the method;      // create leaf node
14            record parameter on edge;             // create condition on edge
15            if (!first node at this path)
16              connect to last node found;         // create links between leafs
17          }
18        }
19    }
```

Algorithm 3. Interaction graph generation

An entry method is depicted as a black node in IG. There are two more points to note in Figure 3. First, an interaction without parameter is an *automatic invocation* (e.g. node 3 in method B). Second, there may be causal relationships among the variables used in the predicates (e.g. in method B, x depends on y and y depends on z). These method types can be easily determined by scanning the pseudo-code.



node	variable	value	invoke
3	x	a: <=y b: >y	-
	y	random	
4,5	-	-	-

node	variable	value	invoke
3	x	a: <= y b: >y	m1()
4	y	c: <=z d: >z	m2(b)
	z	random	-
5,6	-	-	-

node	variable	value	invoke
3	x	a: <=y b: >y	-
	y	random	
4	z	c: <=v d: >v	-
	v	random	
5	-	-	m1(a,c)
6	-	-	m2(a,d)
7	-	-	m3(b)

Figure 3. Transformation of models from class testing into interaction graph

After the IGs of each method are constructed, an execution tree (ET) is formed by linking and merging the IGs in the cluster. An ET is a finite tree whose root node corresponds to an initial method and the leaf nodes are the terminal services in the cluster. The conditions on the edges from each IG are also preserved in the ET. The original control flow model of the cluster can be very complex, but this reduced model only captures the interaction information for intra-cluster testing. The construction of the execution tree is shown in Algorithm 4.

```
1  input:  CLIB, LIG - class library of the cluster and a list of interaction graphs
2  output: ET - execution tree for the cluster
3  -------------------------------------------------------------------------
4  LLIG = empty;                               // set LLIG for construction phase
5  for every concrete class in CLIB do         // no abstract classes
6    for each method in LIG do {               // get method of the class
7      if the method not in LLIG               // new method found
8        create new node and add to LLIG;      // create node with method, connect to list
9      if method type is not service           // main or agent or entry method
10       for each method invocation do {       // found new link
11         create edge to invoked method;      // connect invoked methods
12         copy parameters to the edge;        // store conditions for the call
13       }
14   }
15 for every method in LLIG do                 // expansion phase
16   for each edge to other method do          // deal with a link each time
17     if invoked method != the method {       // not recursion in ET
18       add node, edge and parameters;        // clone the branch
19       mark the invoked method is called;    // record the subtree is tested
20       handle the node at the same way;      // depth first expansion
21     }
22 for each marked method in LLIG do           // delection phase
23   delete the subtree start from the method; // redundant subtree
24 ET = LLIG;                                  // final execution tree
```

Algorithm 4. Constructing an execution tree from interaction graphs

The example used class testing is expanded to demonstrate the construction of ET in static analyse and test case generation in dynamic analysis. The automatic teller machine (ATM) network model[2] from Rumbaugh's book [21] is used. Figure 4 shows the components in a banking system. The four types of cluster are shown by the rectangular boxes. Each cluster may have multiple instances with the exception of *Consortium*[3]. A solid arrow depicts a communication link between two clusters whereas a V-headed arrow denotes method invocation among the objects in the cluster. Also, each method is labelled for identification. In order to simplify the discussion, only the *BankComputer* cluster contains multiple objects in this example.

---

[2] The system supports a computerised banking network including both human cashiers and ATMs to be shared by a consortium of banks. Each bank provides its own computer to maintain its own accounts and process transactions against them.

[3] ATMs communicate with the Consortium which clears transactions with the appropriate banks. The bankInfo method serves to select the right instance of BankComputer to interact.

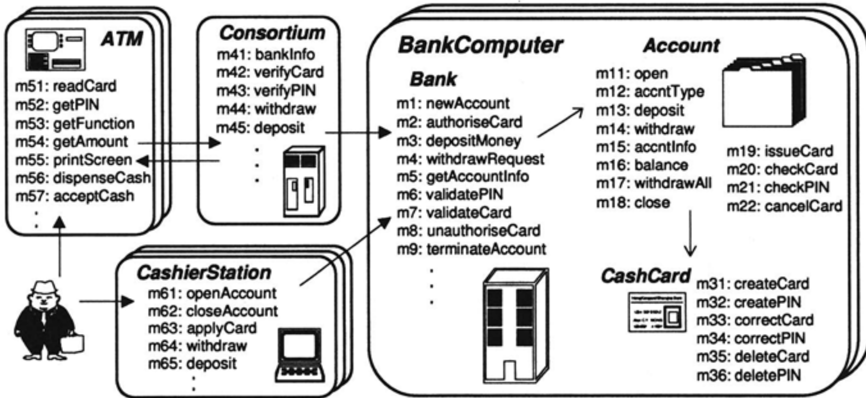
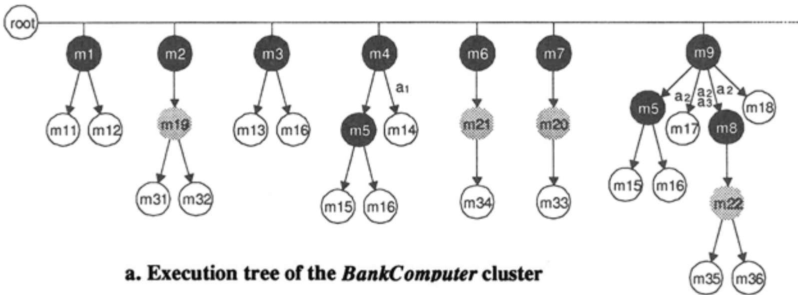
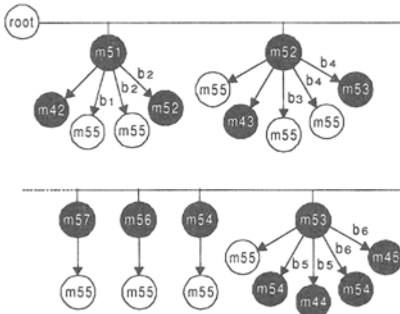


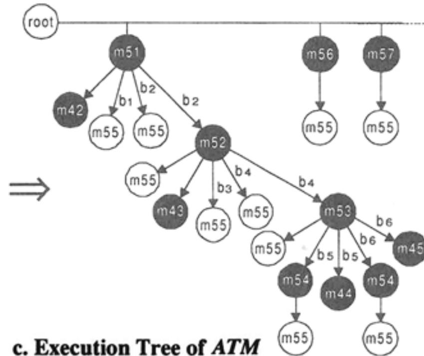
Figure 4. Clusters, objects and methods in a banking system



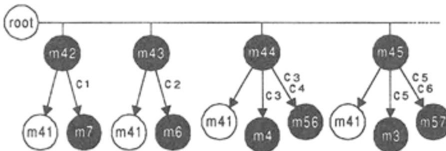
a. Execution tree of the *BankComputer* cluster



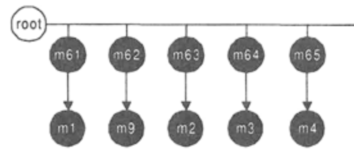
b. List of interaction graphs in *ATM*



c. Execution Tree of *ATM*



d. Execution Tree of *Consortium*



e. Execution Tree of *CashierStation*

Figure 5. Execution tree of the banking system example

A list of IGs is derived from the VTs and CFGs of each method. By applying Algorithm 4, an execution tree of each cluster is constructed as shown in Figure 5. All entry methods of a cluster are connected to the root node. They serve as an interface of that cluster. For example, all methods of the *bank* object (m1 to m9 in Figure 5.a) are considered as entry methods in the BankComputer cluster. An agent method or an entry method is refined until all leaf nodes are replaced by service methods or entry methods from other cluster. For instance, in the ET of *CashierStation* (Figure 5.e), the *applyCard* method (m63) triggers an entry method *authoriseCard* (m2) in the *BankComputer* cluster. On the other hand, this method in the *Bank* object (Figure 5.a) invokes an agent method (m19) *issueCard* in the *Account* object, which instantiates a new *CashCard* by invoking the *createCard* and *createPIN* (m31, m32) service methods. The parameters in each edge specify the conditions of method invocation. This is illustrated in Figures 5.b and 5.c which show the generation of the *ATM* execution tree from the interaction graphs. In static analysis the execution tree is examined for erroneous structures. A path is unreachable if an agent method or a service method connects directly to the root node. Since only the main method and entry methods can be initiated by the root node, a fault is detected.

5.2 Dynamic Analysis

Test cases in intra-cluster testing are presented by sequences of method invocations. Since we have assumed no concurrent execution within a cluster, every method execution sequence has pre-determined start and end points. Starting from a root node of an ET, a backtracking algorithm derives the path to every leaf node. Each path specifies a sequence of method invocations from top to bottom and from left to right of a subtree. The conditions in each edge are noted and they must be satisfied by the previous invoked methods. Static data flow analysis is employed to derive the values of test data required for the sequence. These values define a particular path of execution in a method such that the desired conditions are fulfilled. In intra-cluster testing, an invocation of an entry method in another cluster is considered as an interaction to a black-box (denoted as 'X' in the test cases shown below). The faults caused by this type of interactions are handled by inter-cluster testing.

Every execution sequence generates a test case. The conditions associated to each invocation is placed in the brackets next to the method (similar to that of VT shown in Figure 3). By eliminating the sequences that are completely contained in some other test sequences, the number of test cases is significantly reduced. The complete set of test cases of the *ATM* cluster (shown in Figure 6a) can be reduced to that shown in Figure 6b. In this example, the reduction ratio is approximately 5 to 2.

The faults such as type confusion of objects, name confusion of methods, missed instantiations may result in wrong or shorten execution paths, or even stopped

execution. These incorrect interactions can be identified by examining the ET. The static analysis detects every unreachable code of the cluster whereas the dynamic analysis guarantees that every interaction in a cluster is executed at least once.

a. complete set of test cases
1: m51 X
2: m51 X m55(b1)
3: m51 X m55(b2)
4: m51 X m55(b2) m52(b2)
5: m51 X m55(b2) m52(b2) m55
6: m51 X m55(b2) m52(b2) m55 X
7: m51 X m55(b2) m52(b2) m55 X m55(b3)
8: m51 X m55(b2) m52(b2) m55 X m55(b4)
9: m51 X m55(b2) m52(b2) m55 X m55(b4) m53(b4) m55
10: m51 X m55(b2) m52(b2) m55 X m55(b4) m53(b4) m55 m54(b5) m55
11: m51 X m55(b2) m52(b2) m55 X m55(b4) m53(b4) m55 m54(b5) m55 X(b5)
12: m51 X m55(b2) m52(b2) m55 X m55(b4) m53(b4) m55 m54(b6) m55
13: m51 X m55(b2) m52(b2) m55 X m55(b4) m53(b4) m55 m54(b6) m55 X(b5)
14: m56 m55
15: m57 m55

b. reduced set of test cases
2: m51 X m55(b1)
7: m51 X m55(b2) m52(b2) m55 X m55(b3)
11: m51 X m55(b2) m52(b2) m55 X m55(b4) m53(b4) m55 m54(b5) m55 X(b5)
13: m51 X m55(b2) m52(b2) m55 X m55(b4) m53(b4) m55 m54(b6) m55 X(b6)
14: m56 m55
15: m57 m55

Figure 6. Intra-cluster test cases for the *ATM* cluster

6. INTER-CLUSTER TESTING

DOOT verifies the functionality and reachability of a cluster by exhaustive class testing and intra-cluster testing. After the first two levels of testing, the correctness and consistency of individual cluster is assumed. Therefore inter-cluster testing only focuses on faults that are related to the interactive calls between entry methods among clusters. The success of this test ensures the reachability of all possible communication links within a distributed system. This is achieved by constructing a transaction tree of the entire system. Since there is no concurrency within a cluster, each execution path in a cluster can be considered as a transaction. The condition for the successful execution of a transaction is the aggregate of all the conditions denoted in the edges of the path.

Algorithm 5 describes the construction of a transaction tree from the ETs of each cluster. The algorithm comprises of three phases. The first phase removes all non-entry methods but keeps the necessary conditions. The second phase expands each leaf entry method with its subtree structure. The last phase deletes all the redundant subtrees.

```
1  input:  ET - execution tree per cluster
2  output: TT - transaction tree of the whole system
3  ---------------------------------------------------------------------------
4  LRET = empty;                                   // form an empty list of reduced ET
5  for each ET in the system do                    // reduction phase
6    for each subtree do {                         // handle a subtree at a time
7      create new node in LRET;                    // this can be entry method or main
8      for each path do                            // search all paths
9        while (not end of path) do {              // ignore non-entry methods
10         follow path and collect conditions on edge; // form trace of parameters
11         if (invoked an entry method)            // find new entry method invocation
12           create new node and record conditions; // produce interaction point with edge
13       }
14   }
15 for each subtree in LRET do                     // combination phase
16   while (a leaf is defined as subtree) do       // the transaction can be extended?
17     substitute the leaf with the subtree and mark it; // copy related transaction and record tested
18 for all marked subtree do                       // deletion phase
19   delete from LRET;                             // remove redundant subtree
20 TT = LRET;                                      // final transaction tree
```

Algorithm 5. Formation of transaction tree

Figure 7.a shows the transaction tree that starts from the entry method *m51* of the *ATM* cluster (refer to the ETs in Figure 5). Figure 7.b shows some transaction sequences in the system[4]. Since the reachability of all possible paths within each cluster has been checked in the previous test levels, these paths are not the focus in this test. For instance, the sequence $\alpha_1, \alpha_2, \alpha_3$ in the *ATM* cluster[5] is represented by the transaction flow β. It is only necessary for each transaction to connect two entry methods. For example, the transaction flow β connects the entry methods *m51* and *m52* in the *ATM* cluster, and the transaction flow γ links the entry method *m52* to the entry method *m43* in the *Consortium* cluster. Moreover, a transaction sequence terminates if it invokes an entry method that does not have any outgoing transaction flow. For example, the transaction flow δ terminates the sequence on invoking the entry method *m7*.

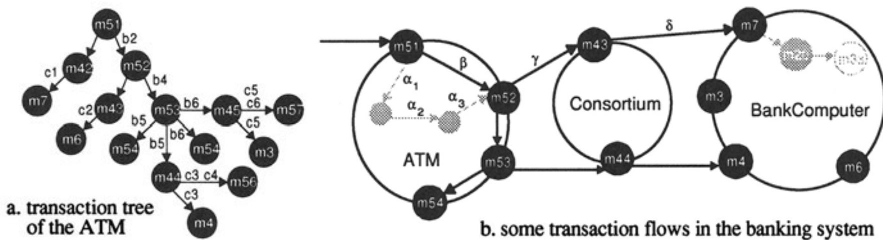


Figure 7. Inter-cluster testing

[4] All transaction flows have been recorded in the transaction tree. This incomplete diagram is shown only for description purpose.

[5] The shaded parts are not supposed to be in the diagram, they are shown for clarity reason.

In the transaction tree, every path from the root node to a leaf node forms a possible test case. The execution sequences between two entry methods are taken from the test cases generated by the intra-cluster testing. Similar to other test levels, all test cases that are completely embedded in other test cases are removed. The transaction tree in Figure 7.a generates two test sequences given below. In this case, they are the expanded version of test cases 11 and 13 from Figure 6.

new 11: m51 m42 m41 m7(c1) m20 m33 m55(b2) m52(b2) m55 m43 m41 m6(c2) m21 m34 m55(b4) m53(b4) m55 m54(b5) m55 m44(b5) m41 m4(c3) m5 m15 m16 m14(a) m56(c3,c4) m55

new 13: m51 m42 m41 m7(c1) m20 m33 m55(b2) m52(b2) m55 m43 m41 m6(c2) m21 m34 m55(b4) m53(b4) m55 m54(b5) m55 m45(b5) m41 m3(c5) m3 m13 m16 m57(c5,c6) m55

Again, all the conditions in the transaction flow must be satisfied for the execution. The results of data flow analysis are reused for deriving test data. This test level guarantees the coverage of all coupled entry methods between clusters. In other words, every communication link of the entire system is executed at least once.

7. SYSTEM TESTING

Due to the interleaving and non-deterministic behaviour of DOOSS, traversing all possible paths is not practical (and often impossible). Various techniques have been proposed to leave out a particular interleaving in the state space search or to reduce the global states by partial ordering reduction [8, 25]. DOOT employs two common approaches for system testing. The first one checks user specified requirements and the second one employs the random walk technique.

The former detects context constraints that are specific to the application domain. This enhances the exhaustive testing in the previous levels with compositional reachability analysis [3]. The important system invariants are declared as test requirements in the domain context. Test cases are specifically generated with the proper parameter values to enforce the execution of pre-determined paths. With this test in DOOT, we can guarantee the essential constraints of the system are satisfied. The latter performs a random simulation that walks through various classes and clusters of the system. Due to the scale of DOOSS, the global states can neither be represented nor found exhaustively. In this controlled search based on random selections of successor states, no effort is made to predict where errors are likely to occur in the state space. Some researchers argue that the approach is not only the simplest technique to implement, but is also likely to produce the highest quality search in testing [17].

8. CONCLUSION

We have proposed a framework for testing distributed object-oriented software systems known as DOOT. DOOT employs an integrated incremental testing

approach at four levels - class level, intra-cluster level, inter-cluster level, and system level. An exhaustive reachability test is adopted in the fine grain class level testing so that the test effort is significantly reduced in the coarse grain cluster and system level testing. Each level addresses the requirements at its level and builds on the results of testing at the previous levels. We also provide guidelines for system testing to cover the user-specified requirements and a final random-walk error detection. The approach was tested on a conferencing system given in the Appendix. More substantial work on inter-cluster testing is under development.

9. REFERENCES

[1] Andleigh P.K. and Gretzinger M.R., *Distributed object oriented data-systems design*, Prentice Hall, 1992.

[2] Chanson S.T. and Zhu. J., Automatic protocol suite derivation, *Proceedings of INFOCOM '94 Conference on Computer Communications*, Vol. 2, pp. 792-799, 1994.

[3] Cheung S.C. and Kramer J., Contextual local analysis in the design of distributed systems, *International Journal of Automated Software Engineering*, Vol. 2, No. 1, pp. 5-32, 1995.

[4] Chin R.S. and Chanson S.T., Distributed object-based programming systems, *ACM Computing Surveys*, Vol. 23, No. 1, pp. 91-124, 1991.

[5] Graham I., *Object-Oriented Methods*, Addison-Wesley, 1994.

[6] Hayes J.H., Testing of object-oriented programming (OOPS): A fault-based approach, *Proceedings of 14th ICSE*, IEEE Press, pp. 205-220, 1992.

[7] Hoffman D., A case study in class testing, *PROC/CASON'93*, Vol.1, pp.472-82, 1993.

[8] Holzmann G.J., *Design and validation of computer protocols*, Prentice-Hall, 1991.

[9] Jorgensen P.C. and Erickson C., Object-oriented integration testing, *Communications of the ACM*, Vol. 37, No. 9, pp. 30-33, 1994.

[10] Jorgensen P.C., *Software testing - a craftsman's approach*, CRC Press, 1995.

[11] Kim M., Chanson S.T. and Kang S., An approach or testing asynchronous communicating systems, *Proceedings of IWTCS'96*, pp. 141-155, 1996.

[12] Kim M.C., Chanson S.T. and Kim G.H., Concurrency model and its application to formal specifications of asynchronous protocols, *Proceedings of IEEE GLOBECOM*, Vol. 3, pp. 1580-4, 1995.

[13] Kirani S. and Tsai W.T., *Specification and verification of object-oriented programs*, Technical report, University of Minnesota, 1994.

[14] Lamport L. and Lynch N., Distributed computing: models and methods, *Handbook of theoretical computer science*. pp. 1157-1199, Elseiver Science, 1990.

[15] Marick B., *The craft of software testing - subsystem testing including object-based and object-oriented testing*, Prentice Hall, 1995.

[16] McGregor J.D. and Korson T.D., Integrating object-oriented testing and development processes, *Communications of the ACM*, Vol. 37, No. 9, pp. 59-77, 1994.

[17] Mihail M., Papadimitriou C.H. and Dill D.L., On the random walk method for protocol testing, *Proceedings of Computer Aided Verification*, pp. 132-41, 1994.

[18] Mueller F., Whalley D.B. and B. Le Charlier, Efficient on-the-fly analysis of program behavior and static cache simulation, *Proceedings of First International Static Analysis Symposium*, SAS '94, Springer-Verlag, pp. 101-15, 1994.
[19] Murphy G.C., Townsend P. and Pok S.W., Experiences with cluster and class testing, *Communications of the ACM*, Vol. 37, No. 9, pp. 48-58, 1994.
[20] Poston R.M., *Automating specification-based software testing*, IEEE Press, 1996.
[21] Rumbaugh J., Blaha M., Premerlani W., Eddy F. and Lorensen W.: *Object-Oriented Modelling and Design*, Prentice Hall, 1991.
[22] Ryan T.W., *Distributed object technology: concepts & applications*, Prentice Hall, 1997.
[23] Shel S., *Object oriented software testing*, John Wiley & Sons, 1996.
[24] Smith M.D. and Robson D.J., A framework for testing object-oriented programs, *Journal of object-oriented programming*, Vol. 5, No. 3, pp. 45-53, 1992.
[25] Tai K.C. and Carver R.H., Testing of distributed programs, *Handbook of parallel and distributed computing*, pp 955-978, McGraw Hill, 1995.
[26] Ulrich A. and Chanson S.T., An approach to testing distributed software systems, In *Proceedings of PSTV'95*, Chapman & Hall, pp. 121-136, 1995.
[27] Ulrich A., A Description model to support test suite derivation for concurrent systems, *Kommunikation in verteilten systemen (KiVS'97)*, Springer Verlag, pp. 151-166, 1997.
[28] Ural H., Testing sequence selection based on static data flow analysis, *Computer communication*, 10(5), 1987.
[29] Wong C.Y., Chanson S.T., Cheung S.C. and Fuchs H., *Testing distributed object-oriented system*, Technical report, Hong Kong University of Science and Technology, April 1997.

Appendix - Case Study

A simple conferencing system was implemented and tested using the framework. The system was built using the configuration language DARWIN and the REGIS-runtime environment and follows the client-server architecture. A conference client is required at each terminal while a conference server runs as a perpetual process. The conference server can handle more than one conference simultaneously, and a user can join more than one conference by starting different instances of the client program. In this exercise, class testing and intra-cluster testing concentrate on individual clusters which are the client and server programs. Inter-cluster testing deals with the concurrency within the conference client and the interaction between the the client and server programs. Different test levels are illustrated as given in the framework. To save space, the following description only focuses on the testing of the conference client (please refer to our technical report [29] for full details).

Conference Client

The conference client consists of a number of interacting objects organised in different class hierarchies. Some of these are static objects that exist throughout the lifetime of the client program. Others are created and destroyed dynamically. The two threads are described by the *console* and *c_client* objects. *Console* is a predefined REGIS keyboard class which interacts with the user. During user input, this object is blocked instead of the whole process of the conference client. In this way, the *c_client* object can also wait for incoming messages from the conference server. *C_client* contains the following basic objects:

server: is responsible for all interactions to and from the server through ports and entries;
interface: is responsible for all interactions of the user with the client-program;
user: is responsible for getting and maintaining data from user for identification;
screen: is responsible for all formatted outputs of the conference client;
overview: is an abstract class that presents an overview for selecting functions, records, users, conferences etc. It manages the display by using the *maskhandler* or *pullup* object.
conference: manages an active conference by controlled actions, such as input messages from message window, whiteboard or chairperson menu;
confpart: is responsible for the display and control of the conference participants;
pullup, *maskhandler*: are responsible for some special functionalities in the client program.

Figure A depicts a simplified architecture of the conference-client. A small circle denotes a port or an entry point whereas an arrow represents a 'used-by' relationship between two objects.

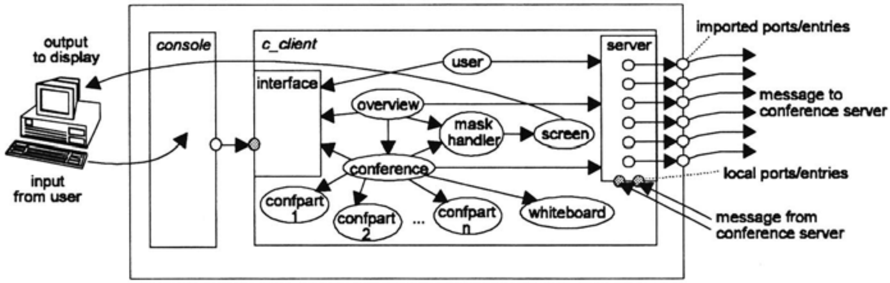


Figure A. The conference client cluster with use-relationships

Class Testing - The following table shows the number of test cases generated in method testing and object testing in each object class. It also includes the size of each class in terms of number of lines of code, attributes, methods and number of nodes in the CFG and IG. (# stands for 'number of' whereas % stands for 'number of test cases')

class	c_client	interface	server	user	screen	overview	conference	confpart	maskhandler	total
# lines in header	118	44	89	47	42	70	44	35	36	525
# lines in body	73	276	327	141	191	92	244	189	90	1623
# attributes	-	4	8	9	6	13	11	9	7	67
# methods	1	13	25	8	18	7	16	5	9	102
# nodes in CFG	14	187	188	104	96	68	139	108	53	957
# nodes in IG	12	40	97	67	40	46	91	55	39	487
% method testing	1	77	42	25	27	27	44	46	12	301
% object testing	0	8	10	6	10	16	22	6	5	83

Intra-Cluster Testing - By applying the intra-cluster testing, the global state transitions are transformed to interactions in the execution tree. The test case generation employs a reduction algorithm to eliminate repeated execution sequences. In the conference client example, the overall number of test cases generated for this test level is found to be 359.

Inter-Cluster Testing - This exercise only considers the interactions between the *console* and *c_client* objects. Since the *c_client* object is tested exhaustively in the previous levels, the internal execution sequences are not of interest. Therefore the overall number of test cases generated for inter-cluster testing is around 100 compared to the traditional approach (no abstraction in test levels) where the total number of test cases is well over 100,000.