Extended abstract of paper presented at the 13th International Conference on Testing Computer Software, Washington, DC (1996)

A Formal Framework for Improving Object-Oriented Software Testing*

T.H. Tse   Zhinong Xu
Department of Computer Science
The University of Hong Kong
Pokfulam Road, Hong Kong
Email: tse@cs.hku.hk, zxu@cs.hku.hk

1. Introduction

Object-oriented software development raises important and challenging testing issues that cannot be solved directly by existing testing techniques for conventional programming languages. Adaptation of existing testing techniques is necessary, and new testing methodologies specific to object-oriented programs need to be established [1-9].

Object-oriented software testing is generally done bottom-up at four levels [5-7]:

(a) Method-level (or unit) testing refers to the internal testing of an individual method in a class. Existing testing techniques for conventional programming languages [10-11] are fully applicable to method-level testing.
(b) Methods and attributes make up a class. Class-level (or intra-class) testing refers to the testing of interactions among the components of an individual class.
(c) Cooperating classes of objects make up a cluster. Cluster-level (or inter-class) testing refers to the testing of interactions among objects.
(d) Clusters make up a system. System-level testing is concerned with the external inputs and outputs visible to the users of a system.

* This project is supported in part by a grant of the Research Grants Council and a CRCG grant of the University of Hong Kong.

As an important aspect of object-oriented software testing, object-oriented state testing [1-4] focuses on the state-dependent behaviors of individual objects and the state-dependent interactions among objects. A number of papers on object-oriented state testing have been published. Kung et al. [1-2] introduced an object state test model based on Objectcharts. Turner and Robson [3-4] presented a state-based test model based on finite state machines. Our research has benefited from the concepts in both approaches. The main difference is that our approach provides a set of formal step-by-step guidelines on class-level object-oriented state testing. We help to produce better test cases and a more effective testing process. We also help to alleviate the test oracle problem by means of formal specifications and generate test cases through a formal *object test model*. The former is based on FOOPS [12] executable specifications and the latter is based on predicate/transition nets [13-16].

Discussions on the test oracle problem can be found in [8-9, 17-18]. This paper focuses on test case generation at the class level.

2. Formal Process for Object-Oriented State Testing

2.1 Test Oracles based on FOOPS

FOOPS, based on algebraic semantics and category theory, is a very high-level object-oriented specification language with an executable subset. A class in FOOPS is specified in an object module (*omod*), consisting of three sections: attributes, methods, and axioms.

(a) Observers are methods that produce outputs according to the current state of an object. They are declared under the attributes (*at*) section.
(b) Constructors, modifiers, and destructors are methods that change the state of an object. They are declared under the methods (*me*) section.
(c) The behavioral properties of the observers, constructors, modifiers, and destructors are defined under the axioms section by equational axioms (*ax*) and conditional axioms (*cax*).

An example of a FOOPS specification of a simple class *Account* is shown in Figure 1.

In our approach, we test an implementation by determining whether it conforms to the FOOPS executable specification. The FOOPS specification can therefore serve as a reliable reference to alleviate the test oracle problem.

2.2 Object Test Models based on Predicate/Transition Nets

Our *Object Test Models* (*OTMs*) are based on predicate/transition nets, which have successfully been used to model object-oriented concepts [13-14]. Extracted from the FOOPS specification as outlined in the last section, they are used to simulate exhaustively the state-dependent scenarios on the behaviors of objects. Using the concept of reachability trees [15-16], we can then study the set of exhaustive test cases and capture from it a crucial subset for effective testing.

The overall process for object-oriented state testing is outlined in Figure 2.

```
omod ACCOUNT is
protecting INT .
class Account .

let MAX : Int = 5000 .

at findBalance : Account -> Int [default: (0)] .

me credit : Account Int -> Account .
me debit : Account Int -> Account .

var A : Account .
var M : Int .

cax findBalance(credit(A, M)) = findBalance(A) + M
    if M + findBalance(A) <= MAX and M >= 0 .
cax findBalance(debit(A, M)) = findBalance(A) - M
    if M <= findBalance(A) and M >= 0 .

endo
```

Figure 1. FOOPS Specification of Class Account

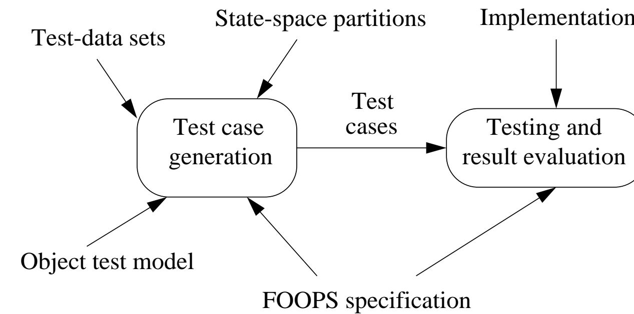

Figure 2. Object-Oriented State Testing Process

3. Class-Level Testing

3.1 Class-Level OTMs

In the class-level OTM, a class is described by a predicate/transition net, where each place represents a set of attributes, and each transition represents a method. Thus, an OTM fully describes the interactions among the components of an individual class. Consider, for example, the simple class *Account*. It has one attribute: *balance*, and five methods: *create*(), *delete*(), *credit*(*m*), *debit*(*m*), and *findBalance*(). Its FOOPS specification, C++ implementation, and OTM are shown in Figures 1, 3, and 4 respectively.

For the purpose of test case generation, we would like to concentrate on the most important parts of an OTM.

Definition 1. An *essential OTM* for a class is a predicate/transition net (P, T, F, M_0), such that:

(a) P is a set of places. Each place represents a set of attributes of the class.
(b) T is a set of transitions. Each transition may be associated with a Boolean expression called a guard, and a list of variables. A transition represents a modifier of the class, a guard represents the condition for execution of the modifier, and the variables represent the parameters of the modifier.

(c) $F \subseteq P \times T \cup T \times P$ is a set of arcs. Each arc is associated with an arc expression representing the relationship between the attributes and the modifier linked by the arc.
(d) $P \cap T = \emptyset$. $P \cup T \neq \emptyset$.
(e) M_0 is the initial marking, representing an initial state of the class. ∎

If a class has more than one constructor, it may have more than one essential OTM, with the same net structures but different initial markings.

Figure 5 shows the essential OTM of the class *Account*. We have $P = \{BALANCE\}$, $T = \{credit(m_1), debit(m_2)\}$, and $M_0 = [s_0]$, where s_0 is the initial state of the class *Account*.

3.2 State-Space Partitions

States are one of the most fundamental concepts in object-oriented programming. A state of an object is a collection of the values of its attributes at a certain moment. The set of possible states of a class is called its state space. Although the state space of a class may be very large, we can subdivide it into a finite number of subspaces according to the specification and implementation, such that all the states within a subspace have an equivalent type of behaviors. We shall call each subspace an *abstract state* of the class and each individual state a *concrete state*.

Definition 2. A *state-space partition* of a class is a collection of abstract states S_1, S_2, ..., S_n of the state space S of the class, where:

(a) $S_1, S_2, ..., S_n$ are non-empty subsets of S,
(b) $S_1 \cup S_2 \cup \ldots \cup S_n = S$,
(c) $S_i \cap S_j = \emptyset$ for every i and $j = 1, 2, ..., n$ such that $i \neq j$.

Let π be a state-space partition of a class. Let a and b be concrete states of the class. We define the equivalence relation ~ induced on the class by π as follows:

$$(a \sim b) \Leftrightarrow (\exists S_p [(S_p \in \pi) \wedge (a \in S_p) \wedge (b \in S_p)]) .$$ ∎

The following heuristics have been adapted from existing partitioning techniques in conventional program testing [10-11]:

(a) If a state-space partition is based on the specification of a class, it must be sufficient to cover the major functions defined in the specification. Testers should consider the following specific values of each attribute in the class and partition the domain of the attribute according to these values:
- Typical values of the attribute.
- Values used in the specification to check the occurrence of some conditions.
- Boundary values, which lie on the boundaries of the domain of the attribute.

(b) If a state-space partition is based on the implementation of a class, testers should partition the domain of each attribute according to the structures of the methods in the class. Such attribute domain partitioning can be based on existing recommendations on test coverage, such as path coverage, branch coverage, or statement coverage.

Based on the above guidelines, a state-space partition of the class *Account* is shown in Figure 6. The five abstract states are S_0, S_1, S_2, S_3, and S_4.

3.3 Test-Data Sets

While partitioning the state space of a class, testers should also choose a set of test data for class-level testing. This set should be the union of *test-data sets* at the method level. A test-data set for a method can be chosen from its input domain by functional or structural testing techniques for conventional programming languages [10-11].

Consider, for example, the test data in Table 1. $\{I_1, I_2, ..., I_7\}$ is the test-data set for the method `credit(m)`, devised by domain testing. $\{J_1, J_2, ..., J_7\}$ is that for `debit(m)`. The set of test data for *Account* at the class level is $\{I_1, I_2, \ldots, I_7\} \cup \{J_1, J_2, \ldots, J_7\}$.

3.4 Test-Trees and Test Cases

Definition 3. Let $\{v_1, v_2, \ldots, v_n\}$ be the set of variables of a transition t, $\{d_1, d_2, \ldots, d_n\}$ be a set of test data, and $\theta = \{v_1/d_1, v_2/d_2, ..., v_n/d_n\}$ be a substitution. The transition t with the substitution θ is enabled at a marking (representing a concrete state) s if and only if

(a) s and θ satisfy the arc expression of each arc from some place to t, and
(b) they cause the guard of t to be TRUE. ∎

```
#include "iostream.h"
#define MAX 5000

class account
{
    int balance;
public:
            account();
            ~account();
    void   credit(int m);
    void   debit(int m);
    int    findBalance();
}

account::account()
{
    balance = 0;
}

account::~account()
{
}

void account::credit(int m)
{
    if (balance + m <= MAX && m >= 0)
           balance = balance + m;
    else cout<<" Input error \n";
}

void account::debit(int m)
{
    if (balance - m >= 0 && m >= 0)
           balance = balance - m;
    else cout<<" Input error \n";
}

int account::findBalance()
{
    return balance;
}
```

Figure 3. C++ Implementation of Class Account

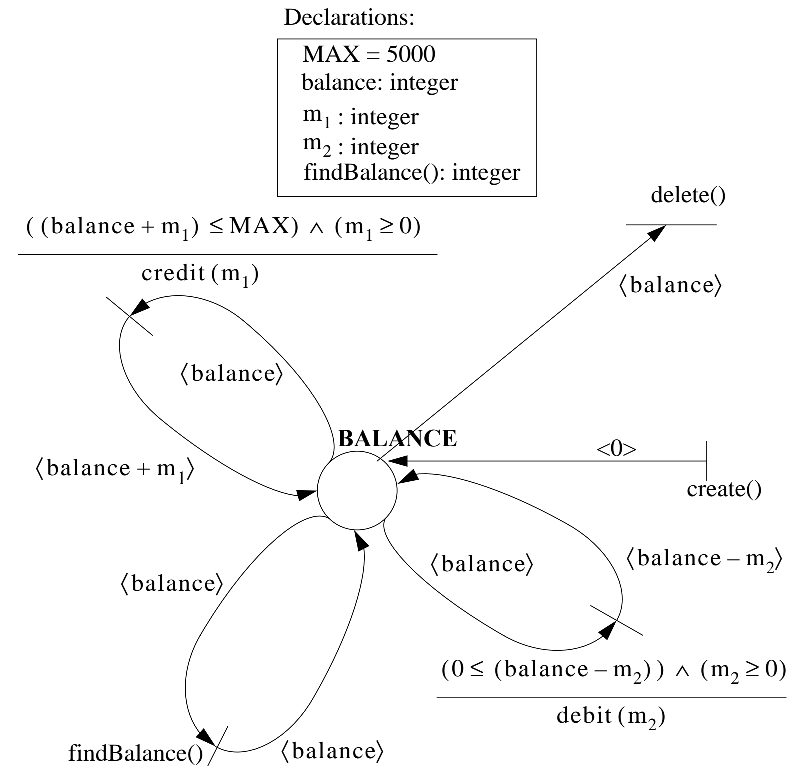

Figure 4. OTM of Class Account

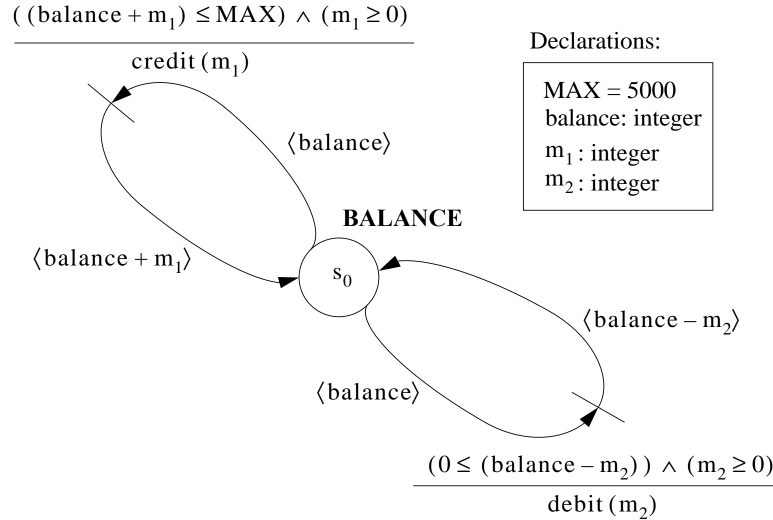

Figure 5. Essential OTM of Class Account

$$S_0 = \{(\texttt{balance} = 0)\},\ S_1 = \{(0 < \texttt{balance} < 1000)\},$$
$$S_2 = \{(\texttt{balance} = 1000)\},\ S_3 = \{(1000 < \texttt{balance} < 5000)\},$$
$$S_4 = \{(\texttt{balance} = 5000)\}.$$

Figure 6. State-Space Partition of Class Account

Table 1. Test-Data Sets for `credit(m)` and `debit(m)`

`credit(int m)`		`debit(int m)`	
partition	test data	partition	test data
m < 0	I_1 = -1	m < 0	J_1 = -1
m = 0	I_2 = 0	m = 0	J_2 = 0
0 < m < 1000	I_3 = 500	0 < m < 1000	J_3 = 500
m = 1000	I_4 = 1000	m = 1000	J_4 = 1000
1000 < m < 5000	I_5 = 3000	1000 < m < 5000	J_5 = 3000
m = 5000	I_6 = 5000	m = 5000	J_6 = 5000
5000 < m	I_7 = 6000	5000 < m	J_7 = 6000

Definition 4. Given
(i) an equivalence relation ~ induced on a class by a state-space partition, and
(ii) a test-data set for each modifier of the class,
the *test-tree* induced from an essential OTM for the class is a reachability tree such that:

(a) Each node of the test-tree is associated either with an abstract state in the state-space partition or with an *Abnormal label*.
(b) Each edge from node n_1 to n_2 is associated with an edge label, which is a list. Each element $t(\theta)$ of the edge label represents a transition t with a substitution θ. If $t(\theta)$ is enabled at the concrete state of n_1, the firing of $t(\theta)$ results in the concrete state of n_2. Otherwise n_2 is given an *Abnormal* label.
(c) For each abstract state, only one of its related nodes is picked for further processing. Every transition (representing a modifier) at this node is considered. Each element of the test-data set for the modifier is used in the substitution process to generate a node.
(d) The nodes with *Abnormal* labels are not processed further. ∎

The algorithm to produce test-trees is shown in Figure 7. s_0 denotes the initial state. s denotes either a concrete state or an *Abnormal* label. If it is a concrete state, the operation *NewNode*(s) creates a new node associated with the corresponding abstract state. Otherwise *NewNode*(s) creates a new node associated with the *Abnormal* label. x denotes a node associated with either an abstract state or an *Abnormal* label. In the former case, s_x denotes the concrete state associated with

the node. In the latter case, s_x denotes the *Abnormal* label. A new concrete state is created by the operation *NewState*(s_x, *t*, θ) that fires an enabled transition *t* with a substitution θ at s_x. Given a source node *x* and a destination node *y*, a new edge is created by the operation *NewEdge*(*x*, *y*). The edge label of this edge will be denoted by l_{xy}. A new element is appended to the edge label l_{xy} by the operation *Append*(l_{xy}, *element*).

A test-tree for the class *Account* is shown in Figure 8, where [S_0], [S_1], ..., [S_4], and [*Abnormal*] denote the nodes, S_0, S_1, ..., S_4 are the abstract states shown in Figure 6, and I_1, I_2, ..., I_7, J_1, J_2, ..., J_7 are the test data shown in Table 1.

In our approach, test cases contain various sequences of method invocations with respect to various test data according to Rule 1 below. Such sequences will be known as *method sequences*.

Rule 1. Class-level test cases can be generated through the following guidelines:

(a) Every constructor of a class should be tested against its corresponding initial state. For each initial state, a test-tree should be generated.
(b) Every method sequence in each test-tree, starting from the root and ending at a leaf, should be tested.
(c) Every observer and destructor at every distinct node of each test-tree should be tested. ∎

Definition 5. A test-tree is *ideal* if and only if it satisfies the following criteria:

(a) Every abstract state in the state-space partition is generated at least once in the test-tree.
(b) Each edge label in the test-tree contains only one element. ∎

For a given set of ideal test-trees for a class, Rule 1 will serve to generate a finite set of test cases such that:

(i1) The set of test cases is minimal. In other words, none of the test cases can be removed without losing the test coverage.
(i2) Every abstract state is exercised at least once.
(i3) Every method invocation with every element of its test-data set is exercised at least once at each abstract state.
(i4) All method sequences that depend on the abstract states are exercised.

Algorithm TestTree:

```
Unprocessed := {NewNode(s_0)}
Processed := ∅
repeat
   Select some node x ∈ Unprocessed
   if ¬( s_x ~ s_y  for some node y ∈ Processed ) then
      begin
      for each transition t with every substitution θ do
         begin
         if t(θ) is enabled at s_x then
            s := NewState(s_x, t, θ)
         else
            s := Abnormal
         if s ~ s_u  for a child node u of x then
            Append(l_xu, "t(θ)")
         else
            begin
            v := NewNode(s)
            NewEdge(x, v)
            Append(l_xv, "t(θ)")
            if t(θ) is enabled at s_x then
               Unprocessed := Unprocessed ∪ {v}
            end
         end
      end
   Unprocessed := Unprocessed – {x}
   Processed := Processed ∪ {x}
until Unprocessed = ∅
```

Figure 7. Test-Tree Generation Algorithm

It is, however, difficult to find a state-space partition and test-data sets to generate an ideal test-tree. In practice, we find the concepts of *sound* and *simplified* test-trees more useful.

Definition 6. A test-tree is *sound* if and only if every abstract state in the state-space partition is generated at least once in the test-tree. ∎

The test-tree in Figure 8, for example, is sound.

A sound test-tree has the same coverage as an ideal test-tree. For a given set of sound test-trees for a class, Rule 1 will serve to generate a finite set of test cases such that only (i2), (i3), and (i4) above are satisfied. The set of test cases thus generated, however, is no longer minimal.

To strike a balance between excessive test cases and full coverage, we introduced the notion of simplified test-trees. The idea behind is that every individual method in a class has already been tested prior to class-level testing. We can therefore concentrate only on the testing of distinct interactions among the components of the class.

Consider an edge label in a typical sound test-tree. Several elements of the edge label may be related to the same transition, albeit with different substitutions. To simplify the sound test-tree, only one of these elements is allowed to remain with the edge label to represent the transition. The other elements related to the same transition are removed.

Definition 7. A *simplified test-tree* is a reduced sound test-tree such that all the elements of each edge label in the test-tree are related to distinct transitions. ∎

A simplified test-tree for the class *Account* is shown in Figure 9.

For a given set of simplified test-trees for a class, Rule 1 will serve to generate a finite number of test cases such that:

(s1) Every abstract state is exercised at least once.
(s2) Every method is exercised at least once at each abstract state.
(s3) All possible interactions among the components of the class, depending on the abstract states, are exercised.

Figure 10 shows some test cases derived from the simplified test-tree in Figure 9 according to Rule 1. Case (A) shows the invocation of the constructor `account()` to create an object, Case (B) shows the invocation of the destructor `~account()` at the node $[S_1]$ to destroy an object, Case (C) shows the invocation of the observer `findBalance()` at the node $[S_2]$, and Cases (D) and (E) show the method sequences along the path from the root node $[S_0]$ through the internal node $[S_4]$ to the leaf node [*Abnormal*].

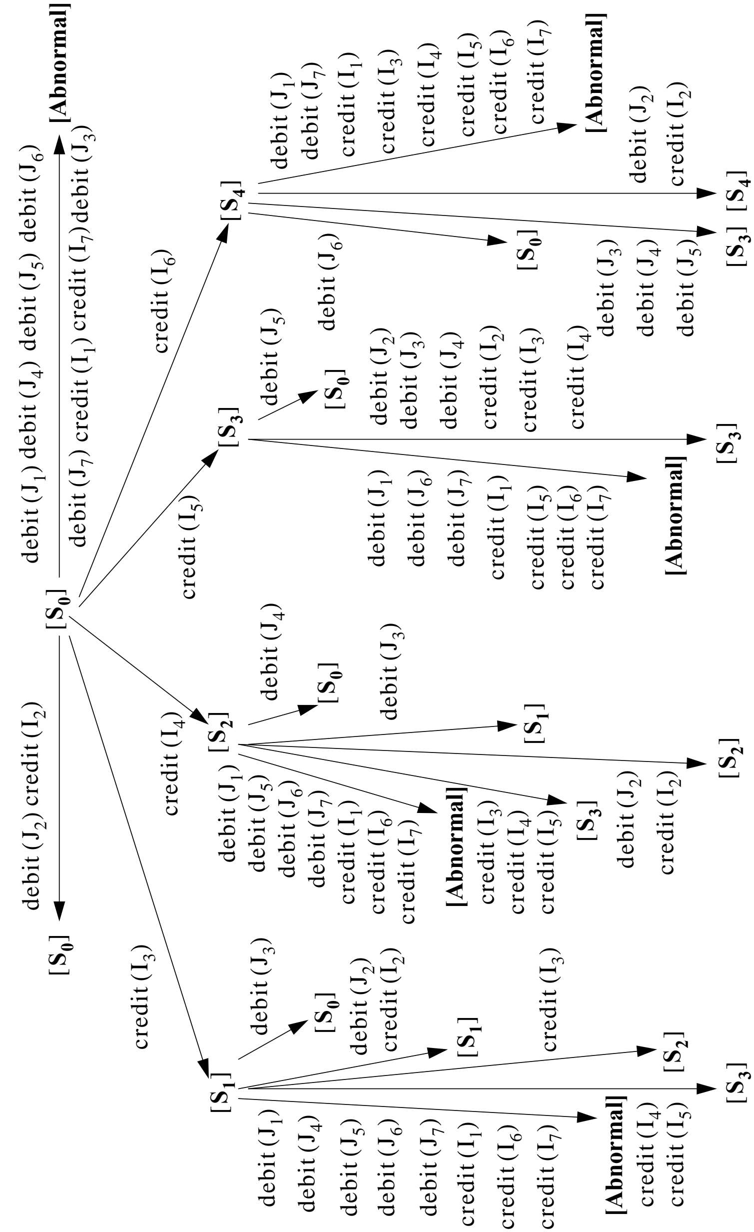

Figure 8. Test-Tree for Class Account

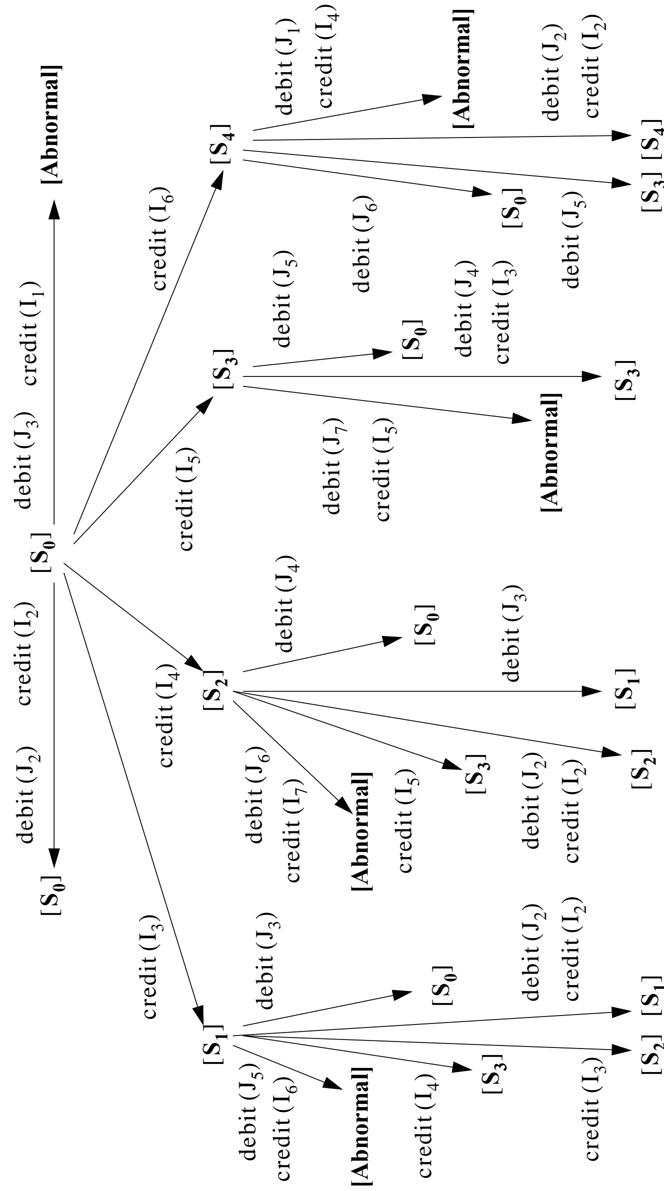

Figure 9. Simplified Test-Tree for Class Account

Case (A):	$\varnothing$ → `account()` → [S_0(`balance` = 0)].
Case (B):	[S_1(`balance` = 500)] → `~account()` → $\varnothing$.
Case (C):	[S_2(`balance` = 1000)] → `findBalance()` → [S_2(`balance` = 1000), output = 1000].
Case (D):	[S_0(`balance` = 0)] → `credit`(I_6) → [S_4(`balance` = 5000)] → `credit`(I_4) → [Abnormal].
Case (E):	[S_0(`balance` = 0)] → `credit`(I_6) → [S_4(`balance` = 5000)] → `debit`(J_1) → [Abnormal].

Figure 10. Some Test Cases for Class Account

3.5 Class-Level Testing Strategy

The procedure for our class-level object-oriented state testing approach can be summarized as follows:

Step 1. Analyze the specification and implementation of the class under test to identify
- the OTM of the class,
- a state-space partition of the class,
- a test-data set for each method of the class.

Step 2. Generate test-trees using the OTM, state-space partition, and test-data sets.

Step 3. Derive test cases from the test-trees.

Step 4. Execute the test cases and verify the test results.

Steps 1 and 2 occur repeatedly as testers refine test-trees. Steps 2, 3, and 4 can be done by testing tools.

4. Conclusion

Our approach has the following advantages:

(a) Because of the formal background from predicate/transition nets,
- The process of transforming a class specification into an OTM helps to expose errors in the specification, design, and implementation of the class.
- An OTM represents the relationships among the states of a class,

leading to important insights into the interactions among the components of the class.
- An OTM covers both states and method sequences, leading to the generation of test cases that simulate scenarios on the behaviors of objects.
- An OTM provides a theoretical basis for integrating existing testing techniques for conventional programming languages into class-level testing.

(b) A formal algebraic specification in FOOPS helps to alleviate the test oracle problem.

(c) The ideal, sound, and simplified test-trees provide a measure for assessing the effectiveness of test cases at the class level, thus supplementing other test coverage criteria currently available.

The approach reported in this paper lays a promising foundation for an integrated object-oriented testing methodology. Directions for future research include the following:

(a) The development of object-oriented test tools based on the method.
(b) Case studies on real-life applications to evaluate the proposal.
(c) The development of strategies for cluster-level and system-level testing by extending the class-level process.

References

[1] D.C.H. Kung, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen, "Design recovery for software testing of object-oriented programs", in *Proceedings of IEEE Working Conference on Reverse Engineering*, IEEE Computer Society, Los Alamitos, California, pp. 202-211 (1993).

[2] D.C.H. Kung, N. Suchak, J. Gao, and P. Hsia, "On object state testing", in *Proceedings of 18th IEEE Annual International Computer Software and Applications Conference* (*COMPSAC '94*), IEEE Computer Society, Los Alamitos, California, pp. 222-227 (1994).

[3] C.D. Turner and D.J. Robson, "Guidance for the testing of object-oriented programs", Technical Report TR-2/93, Computer Science Division, School of Engineering and Computer Science, University of Durham, Durham, UK (1993).

[4] C.D. Turner and D.J. Robson, "The state-based testing of object-oriented programs", in *Proceedings of 9th IEEE International Conference on Software Maintenance*, IEEE Computer Society, Los Alamitos, California, pp. 302-310 (1993).

[5] M.D. Smith and D.J. Robson, "A framework for testing object-oriented programs", *Journal of Object-Oriented Programming* 5 (3): 45-53 (1992).

[6] R.V. Binder (ed.), "Special Issue on Object-Oriented Software Testing", *Communications of the ACM* 37 (9) (1994).

[7] R.J. D'Souza and R.J. LeBlanc, "Class testing by examining pointers", *Journal of Object-Oriented Programming* 7 (4): 33-39 (1994).

[8] R.-K. Doong and P.G. Frankl, "The ASTOOT approach to testing object-oriented programs", *ACM Transactions on Software Engineering and Methodology* 3 (2): 101-130 (1994).

[9] T.H. Tse, F.T. Chan, and H.Y. Chen, "An axiom-based test case selection strategy for object-oriented programs", in *Software Quality and Productivity: Theory, Practice, Education, and Training*, M. Lee, B.-Z. Barta, and P. Juliff (eds.), Chapman and Hall, London, pp. 107-114 (1995).

[10] B. Beizer, *Software Testing Techniques*, Van Nostrand Reinhold, New York (1990).

[11] M. Roper, *Software Testing*, McGraw-Hill, London (1994).

[12] J.A. Goguen and J. Meseguer, "Unifying functional, object-oriented, and relational programming with logical semantics", in *Research Directions in Object-Oriented Programming*, B. Shriver and P. Wegner (eds.), MIT Press, Cambridge, Massachusetts, pp. 417-477 (1987).

[13] T.H. Tse and C.P. Cheng, "Towards a 3-dimensional net-based object-oriented development environment (NOODLE)", 5th International Congress on Computational and Applied Mathematics, Leuven, Belgium (1992). Also Technical Report TR-92-05, Department of Computer Science, The University of Hong Kong, Hong Kong (1992).

[14] T.H. Tse and C.P. Cheng, "NOODLE++: a 3-dimensional net-based object-oriented development model", Technical Report TR-95-04, Department of Computer Science, The University of Hong Kong, Hong Kong (1995).

[15] P. Huber, A.M. Jensen, L.O. Jepsen, and K. Jensen, "Reachability trees for high-level Petri nets", *Theoretical Computer Science* 45 (3): 261-292 (1986).

[16] K. Jensen, *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, Vol. 2, EATCS Monographs on Theoretical Computer Science, Springer-Verlag, Berlin (1995).

[17] M.-C. Gaudel, "Testing can be formal, too", in *Theory and Practice of Software Development (TAPSOFT '95): Proceedings of 6th International Joint Conference of CAAP/FACE*, P.D. Mosses, M. Nielsen, and M.I. Schwartzbach (eds.), Lecture Notes in Computer Science, Vol. 915, Springer-Verlag, Berlin, pp. 82-96 (1995).

[18] J.E. Richardson, S.L. Aha, and T.O. O'Malley, “Specification-based test oracles for reactive systems”, in *Proceedings of 14th IEEE International Conference on Software Engineering* (*ICSE '92*), IEEE Computer Society, Los Alamitos, California, pp. 105-118 (1992).