Postprint of article in *Quality Process Convergence: Proceedings of the 9th International Software Quality Week* (*QW '96*)

Test Case Generation for Class-Level Object-Oriented Testing*

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

Abstract:

In this paper, we discuss a new testing process to generate test cases for object-oriented programs. We focus on classes with mutable objects. The test case construction process is guided by formal object-oriented specifications. In our approach, testers first analyze the formal specification of a class to partition the state space of the class and identify a test model that is based on finite-state machines, then analyze the class specification and the test model to select a set of test data for each method of the class, and finally prepare the test cases of the class from the test model by following various well-developed testing criteria. This paper also extends the subtype relationship to allow testing information on superclasses to be inherited by subclasses. The subtype relation is defined on the formal specification and test model, and improves on the effectiveness of testing using the class inheritance hierarchy.

1. Introduction

Object-oriented software development raises important and challenging testing issues that cannot be solved directly by existing testing techniques [1,2,26] for conventional programming languages. The enhancement of the existing techniques is necessary, and new testing processes specific to object-oriented programs need to be established [4,10,16-20,23,28-31].

Object-oriented software testing is generally performed bottom-up at four levels:

(a) Method-level testing refers to the testing of an individual method in a class.
(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.

Formal methods are gaining recognition in software testing with a view to improving the quality and effectiveness of the process [3,8-10,14,20,29,31]. Following this trend, we have developed an object-oriented software testing process based on formal methods. Our objectives are to generate test cases systematically and effectively, help alleviate the test oracle problem, and develop automatic testing tools to support the entire testing process.

* This project is supported in part by a grant of the Research Grants Council and a CRCG grant of the University of Hong Kong.
** On leave from the Department of Computer Science, Jinan University, Guangzhou, China.

The majority of specification-based class testing techniques in the literature are based on one of three models: algebraic specifications [3,10,29], model-based specifications [20,31], or finite-state machines [18,19,30]. Recent trends in the research and practice in object-oriented specifications has shown the need to combine the three models together to provide a clear and consistent concept on the static and dynamic views of objects, especially on class inheritance [6,7,11,12,21,22,27]. In this paper, we will focus on the generation of test cases for a class with mutable objects and discuss the class-level testing techniques based on the unification of the three models.

A subclass inherits the features of its superclasses. A composite class imports other classes to help define its context. We assume that class-level testing is also performed bottom-up. In other words, a class is tested after its superclasses and its imported classes have been tested.

2. Process for Generating Test Cases at the Class Level

Test models are abstract representations of actual tests. They can be extracted from the design specifications (in the style of black-box testing) or the source code (in the style of white-box testing). Our test model is extracted from the formal specifications. A set of effective test cases satisfying certain coverage criteria is then generated from the test model.

The process of generating test cases at the class level is illustrated schematically in Figure 1.

3. Formal Specifications of Classes*

Our class specifications follow the style of Larch [7,15,21,22]. A class specification consists of two tiers: a functional tier and an object tier. The functional tier is an algebraic specification, which is used to define the abstract values of objects. An algebraic specification may consist of a number of modules. Each module is used to specify a collection of related sorts**. The properties of the operations (or functions) related to a particular sort is specified by a set of equational axioms. In particular, the sort for the abstract values of objects of a class is known as a base sort of the class.

Figure 2 shows a module `ACCOUNT` that specifies the set of abstract values for objects of a simple class `Account`. It begins by importing another module `MONEY`, in which a sort `Money` and the related operations have been specified. `AccountSort` is the base sort of the class `Account`. The carrier set of `AccountSort` is the set of values of objects in the class. An `op` clause declares the signature (or syntax) of an operation. An `eq` or `ceq` clause defines the properties of the operations by means of equations or conditional equations, respectively. In this example, a value of `AccountSort` is constructed by the operations `empty` and `creditS`. The operation `balance` is a function from the base sort to some other sort. We use this kind of operations to specify the attributes of a class.

* Objects in object-oriented programming are based on data abstraction, which can be defined at two different levels. At the more abstract level, known as the specification level, objects are represented by object types. They specify only the expected interfaces and behaviors of objects. At the more concrete level, known as the implementation level, objects are represented by classes. They define also how the data and the behaviors of objects are represented internally. As the two concepts are merged in many of the object-oriented languages, we decide to proceed likewise and use only the term “class” for both interpretations.

** The term “sort” is used in algebraic specifications in the same way that the term “type” is used in programming.

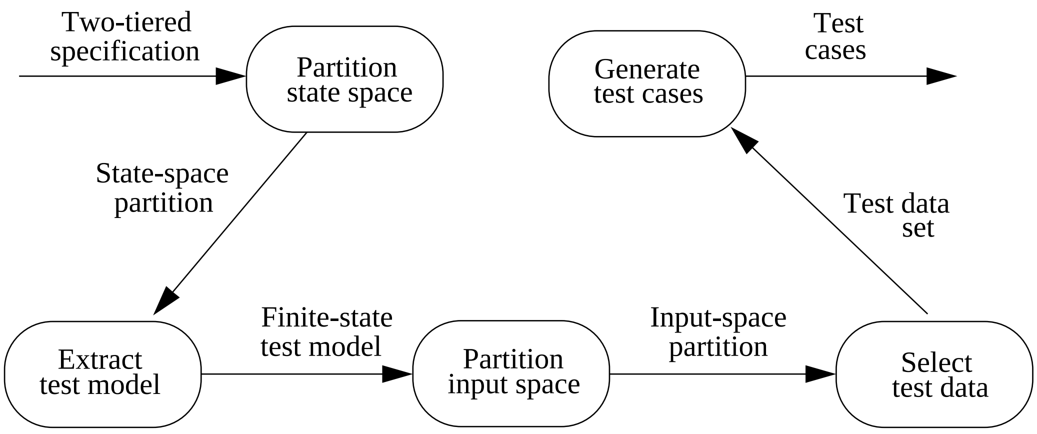


Figure 1. Process for Generating Test Cases

```
module ACCOUNT is
including MONEY .
sort AccountSort .

var X : AccountSort .
var M : Money .
var N : Money .

let creditLimit = 1000 .

op empty : -> AccountSort .
op creditS : AccountSort Money -> AccountSort .

op balance : AccountSort -> Money .
eq balance(empty) = 0 .
eq balance(creditS(X, M)) = M + balance(X) .

op charge : AccountSort Money -> Money .
eq charge(empty, N) = 0 .
ceq charge(creditS(X, M), N) = N * 0.05
   if N < 200 and balance(creditS(X, M)) < 0 .
ceq charge(creditS(X, M), N) = 10
   if N >= 200 and balance(creditS(X, M)) < 0 .
ceq charge(creditS(X, M), N) = 0
   if balance(creditS(X, M)) >= 0 .

op debitS : AccountSort Money -> AccountSort .
eq debitS(empty, N) = creditS(empty, - N) .
eq debitS(creditS(X, M), N) =
          creditS(X, (M - N - charge(creditS(X, M), N))) .

endmodule
```

Figure 2. The Functional Tier of Class Account

In the object tier, the class name, invariant, historical constraint, and methods of each class are specified. A class is a template to define the attributes, services, and behaviors of a collection of objects. A state of an object is a named abstract value, which is a collection of the values of attributes at a certain moment. An object encapsulates its state and methods which manipulate on the state. An invariant is the properties true of all possible states. A historical constraint is the properties true of all possible sequences of states. Each method has its signature and pre- and post-conditions. The invariants, historical constraints, pre- and post-conditions are predicates constructed by boolean terms of the functional tier.

Figure 3 shows the object tier of the class `Account`. In this example, a `based on` clause is used to introduce the base sort. The `invariant` clause introduces an invariant for the class `Account`. In this clause, `self` represents the value of object **self** in some state. A `method` clause declares a signature for each method. In the `require` and `ensure` clauses which follow, the pre- and post-conditions of the method are introduced. The output of the method is denoted by `result.` The names `pre-self` and `post-self` denote the values of the object **self** in the initial and final states, respectively, of the method.

4. State-Space Partition and Test Models

Partition analysis is a standard strategy in software testing. We enhance this strategy so that it operates effectively in class-level testing with formal specifications. In our approach, the state space of a class is partitioned into substates, and the input space of each method is partitioned into sub-domains. Partition analysis mainly involves the reduction of the predicates into a disjunctive normal form, which gives a disjoint partition of the value spaces [8,9], and the use of *uniformity* and *regularity hypotheses* [3,14].

```
class Account is
based on sort AccountSort .

invariant {balance(self) >= - creditLimit} .

constructor Account()
ensure {balance(result) == balance(empty)} .

method credit(M : Money)
require {M > 0} .
ensure {balance(post-self) == balance(creditS(pre-self, M))} .

method debit(M : Money)
require {M > 0 and balance(debitS(pre-self, M)) >= - creditLimit} .
ensure {balance(post-self) == balance(debitS(pre-self, M))} .

method findBalance() : Money
ensure {result == balance(pre-self) and
        balance(post-self) == balance(pre-self)} .

endclass
```

Figure 3. The Object Tier of Class Account.

We first consider the partition analysis of state space. 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 substates based on the class specification, such that all the states within a substate have similar behaviors for the purpose of testing. We shall call each substate an *abstract state* and each individual state a *concrete state*. In our approach, a boolean term in the functional tier is used to describe an (abstract) state. From the state-space partition, we construct a test model for the class. Test cases can then be generated from the test model.

Consider, for example, the class `Account`. Based on the properties of the base sort specified in Figure 2, we partition the state space into two substates: the account being empty and non-empty. Suppose we consider an object **self** and its value in some state is `self`. We can then specify the state-space partition precisely as follows:

`balance(self)` < `0`, `balance(self)` = `0`, `0` < `balance(self)`.

Taking into account the invariant in Figure 3, the state space is further partitioned into four substates. In Figure 4, five parts are shown, including an invalid part (or error state).

`balance(self)` < `- creditLimit (invalid)`, `balance(self)` = `- creditLimit`,
`- creditLimit` < `balance(self)` < `0`, `balance(self)` = `0`, `0` < `balance(self)`.

Figure 4. The State-Space Partition of Class Account

In this simple example, no further partitioning is necessary. A uniformity hypothesis is assumed in each substate. In other words, we assume that an object behaves uniformly in each substate.

Our test model for a class is a finite-state machine, which describes the state-dependent behaviors of individual objects of the class. A test model is composed of a set of states and a set of transitions among the states. Each state is obtained through the state-space partition of the class. Each transition consists of a method, which may change the value of an object from the source state of the transition to the target state of the transition, and a guard predicate derived from the pre-condition of the method. There are two special states: initial and final states. The former represents the state before an object is created. The latter represents the state after an object is destroyed. The test model of class `Account` is shown in Figure 5, where the error states are not depicted. There are five states:

$S_0 = \{unborn\}$, $S_1 = \{b == -1000\}$, $S_2 = \{-1000 < b < 0\}$, $S_3 = \{b == 0\}$, $S_4 = \{0 < b\}$,

where S_0 is the initial state, *b* stands for the attribute *balance* as specified by `balance`, and 1000 is the `creditLimit` as shown in Figure 4. There are 21 transitions:

$t_0 = Account()$, $t_1 = credit(m)$ $\{0 < m < 1000\}$, $t_2 = credit(m)$ $\{m == 1000\}$, $t_3 = credit(m)$ $\{1000 < m\}$, $t_4 = credit(m)$ $\{0 < m < 1000$ and $b + m == 0\}$, $t_5 = credit(m)$ $\{(0 < m < 1000$ and $0 < b + m)$ or $1000 \leq m\}$, $t_6 = credit(m)$ $\{0 < m < 1000$ and $-1000 < b + m < 0\}$, $t_7 = credit(m)$ $\{0 < m\}$, $t_8 = credit(m)$ $\{0 < m\}$, $t_9 = debit(m)$ $\{1000 < m$ and $b - m == -1000\}$, $t_{10} = debit(m)$ $\{0 < m$ and $-1000 < b - m < 0\}$, $t_{11} = debit(m)$ $\{0 < m$ and $b - m == 0\}$, $t_{12} = debit(m)$ $\{0 < m$ and $0 < b - m\}$, $t_{13} = debit(m)$ $\{0 < m < 1000\}$, $t_{14} = debit(m)$ $\{m == 1000\}$, $t_{15} = debit(m)$ $\{(200 \leq m < 990$ and $b - m == -990)$ or $(0 < m < 200$ and $b - 1.05 * m == -1000)\}$, $t_{16} = debit(m)$ $\{(200 \leq m < 990$ and $-990 < b - m < 10)$ or $(0 < m < 200$ and $-1000 < b - 1.05 * m < 0)\}$, $t_{17} = findBalance()$, $t_{18} = findBalance()$, $t_{19} = findBalance()$, $t_{20} = findBalance()$, where *credit(m)* stands for the method `credit(M : Money)` in Figure 3, *debit(m)* for `debit(M : Money)`, *findBalance*() for `findBalance()`, and *Account*() for `Account()`. Consider t_1, for example. The attached method is *credit*(*m*). The guard is {0 < *m* < 1000}.

During partition analysis, we need to distinguish four kinds of methods: mutators (such as `credit(M : Money)` and `debit(M : Money)`), observers (such as `findBalance()`), constructors (such as `Account()`), and destructors. We need to unfold the complex denotations by introducing *observable contexts* [3,14]. For example, we unfold `balance(creditS(pre-self, M))` into `balance(pre-self) + M` and `balance(debitS(pre-self, M))` into `balance(pre-self) - M - charge(pre-self, M)`. Assuming regularity hypotheses, the unfolding process can be completed in a finite number of steps. We can turn our test model into a complete finite-state machine by adding error states, error transitions, undefined states, and undefined transitions.

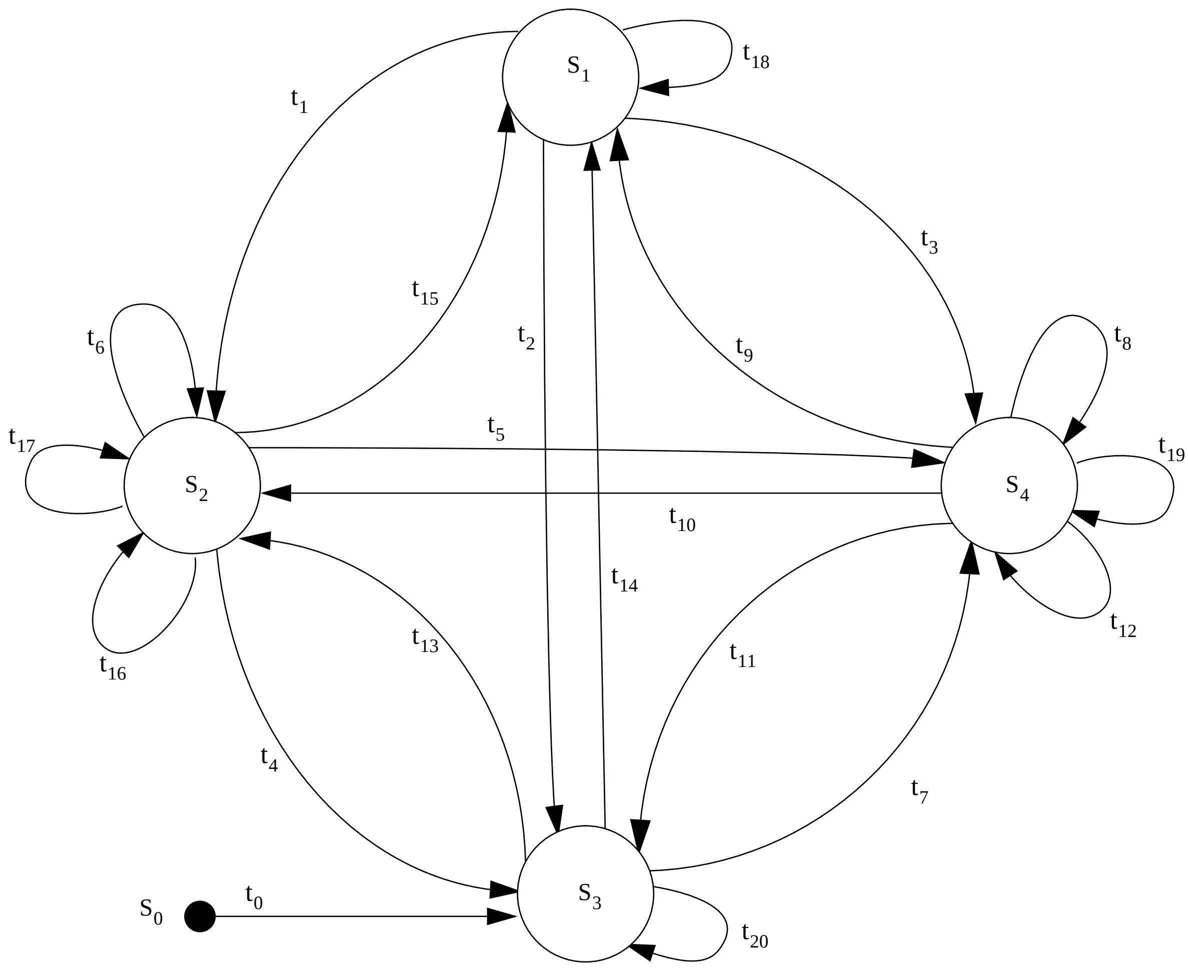


Figure 5. The Test Model of Class Account

5. Input-Space Partition and Test Data Sets

While partitioning the state space of a class, we should also partition the input space of each method, which is the sets of values for the input parameters of the method. A valid input space for a method is the subset of the input space satisfying the pre-condition of the method. Thus, we can at least partition the input space of a method into two sub-domains according to the validity of the input values. Test data can then be drawn from each sub-domain.

When we partition the input spaces of methods, the sort of each input variable is the actual subject of consideration. In our example of class `Account`, the input space of `credit(M : Money)` or `debit(M : Money)` is the carrier set of sort `Money`. The pre-condition of `credit(M : Money)` gives its valid input space as `M > 0` for the variable `M` of the sort `Money`. When we consider the pre-condition of `debit(M : Money)`, we unfold `balance(debitS(pre-self, M))` into `balance(pre-self) - M - charge(pre-self, M)`. Hence the valid input space of `debit(M : Money)` is `(0, balance(pre-self) + creditLimit]`. Since methods are used to manipulate the states of objects, the input-space partition of each method is also related with the state-space partition of the class. During input-space partition of a method, therefore, we also consider the state or some attributes of the class as implied parameters of the method. Based on the functional tier, the object tier, and the test model in Figures 2, 3, and 5, we can partition the input space of each method as shown in Figure 6, where *b* stands for the attribute *balance* and only the valid sub-domains are shown.

We can then choose test data for each input parameter of a method from every sub-domain using existing testing techniques. The simplest way is to assume a uniformity hypothesis in each sub-domain. In other words, we assume that the method under test behaves uniformly in each sub-domain. Based on the hypothesis, we need only select one value randomly from each sub-domain, and obtain a set of test data for each method.

6. Test Case Generation and Test Oracles

In our approach, test cases contain various sequences of method invocations with various test data. The test cases are generated from our test model to test the state-dependent scenarios on the behaviors of interactions among methods of a single class.

First of all, we use finite-state testing techniques to generate test cases. A test case is generated by traversing the test model from the initial state. Method sequences are derived from the traversed transitions. We are interested in a set of test cases that in some way cover the test model, such as in the form of state coverage, transition coverage, and path coverage. In our example, the test case

Account(); *credit*(5000); *debit*(6000); *credit*(1000); *debit*(5); *findBalance*()

covers the five states in the following sequence: S_0, S_3, S_4, S_1, S_3, S_2, S_2.

Secondly, we use data-flow testing techniques to generate test cases. Test cases are generated according to the definitional and computational-use occurrences of attributes in the test model. Each attribute is classified as being defined or used. An attribute is said to be defined at a transition if the method of the transition changes the value of the attribute. An attribute is said to be used at a transition if the method of the transition refers to the value of the attribute. For example, the attribute *balance* in Figure 5 is defined at transition t_0, used at transitions t_{17} to t_{20}, and defined and used at transitions t_1 to t_{16}. A set of test cases is generated by covering d-u paths with regard to each attribute according to certain data-flow testing criteria, such as the all-definitions strategy, the all-c-uses strategy, and the all-du paths strategy. We can also define the predicate-use of an attribute according to the predicates of states and the guards of transitions.

Additional testing techniques based on finite-state machines can be found in [5,13,24].

Our strategy for test oracles is based on the class formal specifications. In our approach, we test a class implementation by determining whether it conforms to its formal specification. The formal specification can therefore serve as a reliable reference to help alleviate the test oracle problem.

7. Subtyping and Testing

Inheritance is an important aspect of object-oriented programming. It also raises the issue of how subclasses should be tested. Unfortunately, the term "inheritance" has been used to mean anything from the preservation of given characteristics with addition of new ones to the reuse of parts of a class without any constraints [25]. In order to facility testability, we restrict the interpretation of inheritance to behavioral subtyping [7,11,12,21,22].

Input-space partition of *credit*(*m*):
For state S_1:
$d_1 = \{0 < m < 1000\}$, $d_2 = \{m == 1000\}$, $d_3 = \{1000 < m\}$.
For state S_2:
$d_4 = \{0 < m < 1000 \text{ and } 0 < b + m\}$, $d_5 = \{0 < m < 1000 \text{ and } b + m == 0\}$, $d_6 = \{0 < m < 1000 \text{ and } b + m < 0\}$, $d_7 = \{1000 < m\}$, $d_8 = \{m == 1000\}$.
For state S_3:
$d_9 = \{0 < m\}$.
For state S_4:
$d_{10} = \{0 < m\}$.

Input-space partition of *debit*(*m*):
For state S_2:
$d_{11} = \{200 < m < 990 \text{ and } b - m == -990\}$, $d_{12} = \{0 < m < 200 \text{ and } b - 1.05 * m == -1000\}$, $d_{13} = \{200 < m < 990 \text{ and } -990 < b - m\}$, $d_{14} = \{0 < m < 200 \text{ and } -1000 < b - 1.05 * m\}$, $d_{15} = \{m == 200 \text{ and } b == -790\}$, $d_{16} = \{m == 200 \text{ and } -790 < b < 0\}$.
For state S_3:
$d_{17} = \{m == 1000\}$, $d_{18} = \{0 < m < 1000\}$.
For state S_4:
$d_{19} = \{1000 < m \text{ and } b - m == -1000\}$, $d_{20} = \{0 < m \text{ and } -1000 < b - m < 0\}$, $d_{21} = \{0 < m \text{ and } b - m == 0\}$, $d_{22} = \{0 < m < b\}$.

Figure 6. The Input-Space Partition for Each Method of Class Account

Subtyping is a relationship among classes. The class *A* is defined as a subtype* of class *B* if and only if any object of *A* behaves as if it were an object of *B*. We extend the notion of subtyping to be a family of relationships between the functional tiers, the object tiers, as well as the test models of *A* and *B*. First, a simulation function [7,21,22] is defined as a mapping from the set of abstract values of *A* to that of *B*. In other words, a simulation function maps the carrier set of the base sort of *A* to that of *B* in the functional tier. Then, in the object tier, *A* must provide all the methods of *B*. We refer to these as inherited methods. Inherited methods must satisfy the *signature* and *method rules* [22], but can be renamed if necessary. In addition to inherited methods, *A* may also have some extra methods. The invariant and historical constraint of *B* must be preserved by *A*. Finally, there is a homomorphism *h* from the test model of *A* to the test model of *B*. *h* consists of a pair of functions: h_s from the set of states of *A* to the set of states of *B* and h_t from the set of transitions of *A* to the set of transitions of *B*. h_s must follow the simulation function in the functional tier.

The example in Figure 7 may provide readers with an intuitive idea of a homomorphism. The transition *t* in (a) and (b) contains an inherited method, obtained by renaming the method in $h_t(t)$. It is called an inherited transition. The inherited transitions must follow the method rule in the object tier. The transition *t* in (c) is newly created in the test model of *A* and is not mapped to any transition in the test model of *B*. (Mathematically, it is mapped to the transition ε, which does nothing.) It is called an extra transition.

If the functional and object tiers as well as the test models of *A* and *B* satisfy the above properties of the extended subtyping, the properties, code, and testing information of *B* can be inherited by *A*. *A* inherits the test information on *B* in the following sense:

Let TC_A be the set of test cases of *A* and TC_B be the set of test cases of *B*. Then the following holds:

$$R((TC_A) \upharpoonright T) \subseteq TC_B$$

where *T* is the set of extra transitions of *A*. That is to say, if we filter out all methods in the extra transitions of *A* from every test case of *A*, we get a subset of the test cases of *B* after renaming. Hence we can obtain a test case of *A* by inserting into a test case of *B* the appropriate methods in the extra transitions of *A*. Thus, testing information on a superclass can be inherited by its subclasses to reduce the cost of testing.

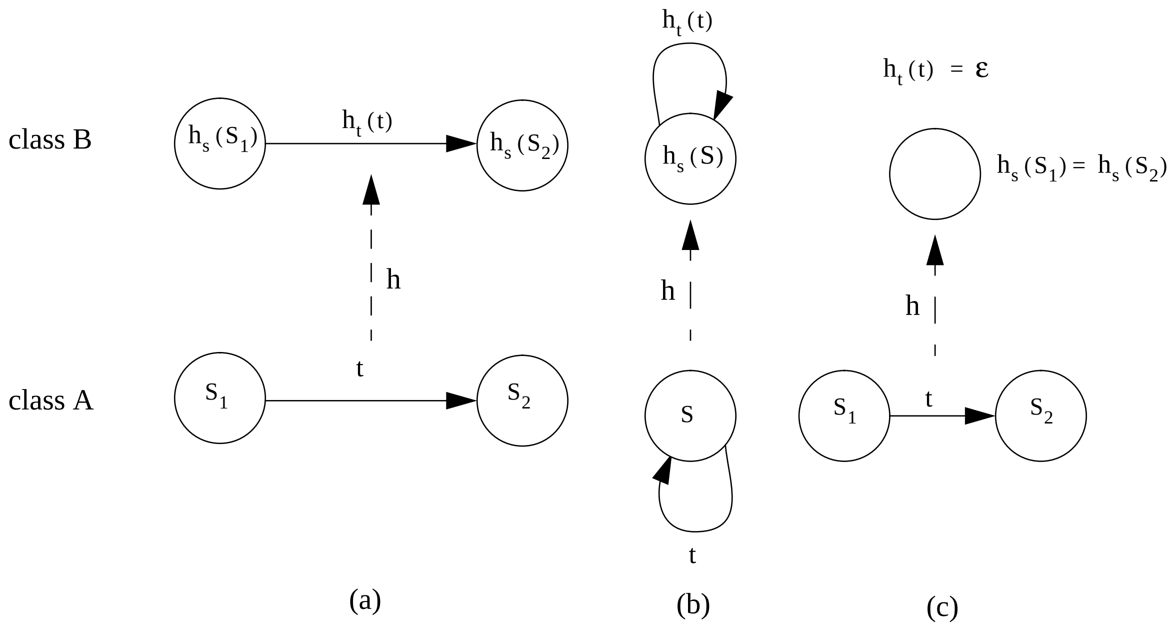
8. Conclusion

We have presented a new technique to generate test cases for class-level object-oriented software testing. It integrates the testing techniques based on algebraic specifications, model-based specifications, and finite-state machines. It supports a systematic testing process, which makes it easier to choose test cases, trace test executions, and analyze test results. Our test models respect the intended behaviors of the original specifications. The test case generation is based on well-developed testing criteria. We have also discussed the inheritance of test information by extending the concept of subtypes.

* For the reason explained earlier, we shall still call A a subclass and B a superclass.

Directions for future research include the following:

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



Finger 7. Homomorphism of Test Models

References

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

[2] B. Beizer, *Black Box Testing: Techniques for Functional Testing of Software and Systems,* John Wiley, New York (1995).

[3] G. Bernot, M.-C. Gaudel, and B. Marre, "Software testing based on formal specifications: a theory and a tool", *Software Engineering Journal* **6** (6): 387-405 (1991).

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

[5] G. von Bochmann and A. Petrenko, "Protocol testing: review of methods and relevance for software testing", in *Proceedings of ACM International Symposium on Software Testing and Analysis (ISSTA '94)*, ACM Press, New York, pp. 109-124 (1994).

[6] G. Booch, *Object-Oriented Analysis and Design with Applications*, Benjamin/Cummings, Redwood City, California (1994).

[7] R.H. Bourdeau and B.H.C. Cheng, "A formal semantics for object model diagrams", *IEEE Transactions on Software Engineering* **21** (10): 799-821 (1995).

[8] D. Carrington and P. Stocks, “A tale of two paradigms: formal methods and software testing”, in *Z User Workshop: Proceedings of 8th Annual Z User Meeting (ZUM '94)*, J.P. Bowen and J.A. Hall (eds.), Workshops in Computing, Springer-Verlag, Berlin, pp. 51-68 (1994).

[9] J. Dick and A. Faivre, “Automating the generation and sequencing of test cases from model-based specifications”, in *Industrial Strength Formal Methods: Proceedings of 1st International Symposium of Formal Methods Europe (FME '93)*, J.C.P. Woodcock and P.G. Larsen (eds.), Lecture Notes in Computer Science, Vol. 670, Springer-Verlag, Berlin, pp. 268-284 (1993).

[10] 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).

[11] J. Ebert and G. Engels, “Structural and behavioural views on OMT-classes”, in *Object-Oriented Methodologies and Systems: Proceedings of International Symposium (ISOOMS '94)*, E. Bertino and Susan Urban (eds.), Lecture Notes in Computer Science, Vol. 858, Springer-Verlag, Berlin, pp. 142-157 (1994).

[12] J. Ebert and G. Engels, “Specification of object life cycle definitions”, Technical Report 19-95, Department of Computer Science, Koblenz University, Germany (1995).

[13] S. Fujiwara, G. von Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi, “Test selection based on finite state models”, *IEEE Transactions on Software Engineering* **17** (6): 591-603 (1991).

[14] 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).

[15] J.V. Guttag and J.J. Horning (eds.), *Larch: Languages and Tools for Formal Specification*, Texts and Monographs in Computer Science, Springer-Verlag, Berlin (1993).

[16] M.J. Harrold, J.D. McGregor, and K.J. Fitzpatrick, “Incremental testing of object-oriented class structures”, in *Proceedings of 14th IEEE International Conference on Software Engineering (ICSE '92)*, IEEE Computer Society, Los Alamitos, California, pp. 68-80 (1992).

[17] M.J. Harrold and G. Rothermel, “Performing data flow testing on classes”, *ACM software Engineering Notes* **19** (5): 154-163 (1994).

[18] D.M. Hoffman and P.A. Strooper, “Graph-based class testing”, *Australian Computer Journal* **26** (4): 158-163 (1994).

[19] H.S. Hong, Y.R. Kwon, and S.D. Cha, “Testing of object-oriented programs based on finite state machines”, in *Proceedings of 2nd IEEE Asia-Pacific Software Engineering Conference (APSEC '95)*, IEEE Computer Society, Los Alamitos, California (1995).

[20] X. Jia, “Model-based formal specification directed testing of abstract data types”, in *Proceedings of 17th IEEE Annual International Computer Software and Applications Conference (COMPSAC '93)*, D. Cooke (ed.), IEEE Computer Society, Los Alamitos, California, pp. 360-366 (1993).

[21] G.T. Leavens and W.E. Weihl, “Specification and verification of object-oriented programs using supertype abstraction”, *Acta Informatica* **32** (8):705-778 (1995).

[22] B.H. Liskov and J.M. Wing, “A behavioral notion of subtyping”, *ACM Transactions on Programming Languages and Systems* **16** (6): 1811-1841 (1994).

[23] J.D. McGregor and D.M. Dyer, “Inheritance and state machines”, *ACM Software Engineering Notes* **18** (4): 61-69 (1993).

[24] M. Mullerburg, “Systematic testing: a means for validating reactive systems”, *Journal of Software Testing, Verification, and Reliability* **5** (3): 163-179 (1995).

[25] F. Parisi-Presicce and A. Pierantonio, “An algebraic theory of class specification”, *ACM Transactions on Software Engineering and Methodology* **3** (2): 166-199 (1994).

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

[27] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall, Englewood Cliffs, New Jersey (1991).

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

[29] 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).

[30] 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 (CSM '93)*, IEEE Computer Society, Los Alamitos, California, pp. 302-310 (1993).

[31] S.H. Zweben, W.D. Heym, and J. Kimmich, "Systematic testing of data abstractions based on software specifications", *Journal of Software Testing, Verification, and Reliability* **1** (4): 39-55 (1992).