HKU CSIS Tech Report TR-94-02

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)

An Axiom-Based Test Case Selection Strategy for Object-Oriented Programs*†

T.H. Tse[a], F.T. Chan[b], H.Y. Chen[c]

[a] Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong.
Email: thtse@cs.hku.hk.

[b] School of Professional and Continuing Education, The University of Hong Kong, Pokfulam, Hong Kong.

[c] Department of Computer Science, Jinan University, Guangzhou, China

ABSTRACT

Because of the growing importance of object-oriented programming, a number of testing approaches have been proposed. Frankl et al. propose the application of the functional approach, using algebraic specifications for the generation of test cases and the validation of methods. Given a specification, Frankl et al. propose that equivalent terms should give observably equivalent objects, and offer general heuristics on the selection of equivalent terms for testing. Their guidelines, however, are only supported by limited empirical results, do not have a theoretical basis, and provide no guarantee of effectiveness.

In this paper, we define the concept of a fundamental pair as a pair of equivalent terms which are formed by replacing all the variables on both sides of an axiom by normal forms. We prove that an implementation is consistent with respect to all equivalent terms if and only if it is consistent with respect to all fundamental pairs. In other words, the testing coverage of fundamental pairs is identical to that of all equivalent terms, and hence we need only concentrate on the testing of fundamental pairs. Our strategy is mathematically based, simple, and much more efficient. Furthermore, it underscores the usefulness of axiom-based specifications.

Keyword Codes: D.2.1; D.2.5; D.1.5

Keywords: algebraic specifications, functional testing, object-oriented

1. INTRODUCTION

Object-oriented programming is considered as an increasingly popular software development method for the 1990s. Since testing is one of the most quality critical and time-consuming phases of the software development process, it is important to investigate into the testing of object-oriented

---

* © 1995 Chapman and Hall. This material is presented to ensure timely dissemination of scholarly and technical work. Personal use of this material is permitted. Copyright and all rights therein are retained by authors or by other copyright holders. All persons copying this information are expected to adhere to the terms and constraints invoked by each authors copyright. In most cases, these works may not be reposted without the explicit permission of the copyright holder. Permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from Chapman and Hall.

† This research is supported in part by a Research and Conference Grant of The University of Hong Kong.

programs. As indicated in [1], the important change from the testing of conventional programs to the testing of object-oriented programs is that the latter programs are not necessarily executed in a predefined order. The sequence of invocation of methods in a class is not specified explicitly. Operations are allowed to be called in any order once an object has been created. Methods across different classes can be combined in various ways. Furthermore, many new testing problems related to inheritance, polymorphism, overloading, encapsulation, aggregation, and state-dependent behaviour have arisen [2]. It is generally accepted, therefore, that object-oriented program testing is quite different from the conventional counterpart.

A number of papers on the testing of object-oriented programs have been published [3−14]. Most of the papers discuss the nature of the problem and potential solutions, using approaches such as functional testing, structural testing [5, 9], and state-based testing [13, 14]. In particular, Frankl et al. [4, 7, 8] propose the application of the functional approach [15, 16] (or black-box testing) to object-oriented programming. They propose the use of algebraic specifications both as a means of selecting test cases for objects and for validating the results of testing. Different test cases are generated from algebraic ''terms'' satisfying the axioms in the specification. A set of automatic tools have been developed integrating test case generation with the execution and verification of test results. Unfortunately, there is a fundamental problem in the theory behind the equivalence of terms, and the proposed heuristics for the selection of terms is only formulated from limited empirical results.

In this paper, we propose an improved theoretical foundation for Frankl's functional approach, and a more efficient strategy for selecting equivalent terms as test cases. Our strategy reduces the domain of test case selection from a polynomial function of the set of axioms to a linear function. On the other hand, we formally prove that the testing coverage is as good as that proposed by Frankl et al.

2. A SUMMARY OF FRANKL'S FUNCTIONAL TESTING APPROACH

In their functional testing approach, Frankl et al. use algebraic specifications [17, 18, 19] of abstract data types (ADTs) to specify classes in target programs. A series of operations on an ADT is known as a term (or called a ''word'' in Frankl et al.) Let u_1 and u_2 be two terms of an ADT. Let s_1 and s_2 be the respective sequences of operations in a given implementation. According to Frankl et al., one term is equivalent to another if and only if one can be transformed into the other using axioms as rewriting rules. The test suite (s_1, s_2) reveals an error of the implementation if u_1 is equivalent to u_2 but the operation sequences s_1 and s_2 produce observationally different objects.

Based on the above idea, Frankl et al. constructed a set of testing tools called ASTOOT. First of all, it accepts an algebraic specification and an implementation of the given class. It then accepts a term u_1, and generates an equivalent term u_2. Test drivers are created automatically by a driver generator to check and execute the operation sequences s_1 and s_2 corresponding to u_1 and u_2, respectively. Finally, the results of the executions of s_1 and s_2 are compared. If they do not give the same observational result, an error is found.

In Frankl's approach, test cases are generated by a compiler and a simplifier. The compiler translates the two sides of each axiom of an algebraic specification into a pair of ADT trees as a transformation rule. A conditional axiom in the specification will give rise to a branch in the ADT tree. The simplifier inputs an original term u_1 provided by the user, translates it into an ADT tree, and applies the transformation rules in the form of the pairs of ADT trees to obtain an equivalent term u_2.

Two important problems are raised: How do we select an original term to be input to the simplifier, and how do we select paths through the resulting ADT trees, in order to increase the likelihood of exposing errors? To explore these problems, Frankl et al. performed two case studies, testing erroneous implementations of priority queues and sorted lists. After running several thousand test cases, they recommended the following tentative guidelines on the generation of test cases:

(i) Select long original terms, with various ratios of different operations.

(ii) For the case of conditional axioms, choose a variety of parameters so that all possible paths in the ADT tree would be traversed.

3. EVALUATION OF FRANKL'S APPROACH

There are a number of merits in Frankl's functional approach and the associated tools:

(a) An integrated set of automatic tools are provided for test case generation, test driver generation, test execution, and test checking.

(b) After running a program with a test case, we must examine the result of execution against the specification of the program. This so called oracle problem is a major concern in program testing. The use of algebraic specifications is an elegant solution.

(c) Frankl's approach takes a suite of methods as a test case, instead of taking an individual method. This concept is especially useful in object-oriented programming, where the sequence of events does not depend on a predefined calling method but on a suite of messages passing among objects.

There are, however, some problems in Frankl's approach:

(i) The definition of the equivalence of terms has a fundamental problem. Consider, for example, the two terms "$([5] \mid [1] \mid [4] \mid [2]).sorting$" and "$([4] \mid [5] \mid [2] \mid [1]).sorting$" in a specification for bubble sort (see Example 1 in the next section). They are regarded by most people as equivalent, and they also produce equivalent results when implemented correctly. However, they cannot be transformed into one another by left-to-right rewriting rules. Furthermore, transformation using axioms as rewriting rules is uni-directional. Thus a term u_1 being equivalent to u_2 would imply that u_2 is not equivalent to u_1. Frankl's papers are based on this fallacious definition.

(ii) The guidelines on the selection of equivalent test cases are supported only by two empirical studies, do not have any theoretical basis, and hence provide no guarantee of effectiveness.

4. AN AXIOM-BASED STRATEGY FOR SELECTING EQUIVALENT TEST CASES

In this paper, we propose a simple mathematically-based strategy for selecting equivalent test cases.

An algebraic specification of ADTs consists of a syntax declaration and a semantic specification [18, 19]. The syntax declaration lists the functions involved, plus their domains and co-domains, corresponding to the input and output variables of methods. The semantic specification consists of axioms which describe the behavioural properties of the functions. The following is an example of an algebraic specification. Bubble sort is chosen because it is familiar to most readers.

Example 1
object *BUBBLESORT* **is**
importing *NAT*
sorts *Bool List*
operations
nil : → *List*
[_] : *Nat* → *List*
sorted : *List* → *Bool*
sorting : *List* → *List*
_ | _ : *List List* → *List* [*associative identity*: *nil*]

variables
N N' : *Nat*
L L' : *List*
axioms
nil.*sorted* = *true*
[*N*].*sorted* = *true* ([*N*] | [*N'*] | *L*).*sorted* = ([*N'*] | *L*).*sorted* and *N* <= *N'*
nil.*sorting* = *nil*
[*N*].*sorting* = [*N*]
L.*sorting* = *L* if *L*.*sorted*
(*L* | [*N*] | [*N'*] | *L'*).*sorting* = (*L* | [*N'*] | [*N*] | *L'*).*sorting* if *N'* < *N*
end

A list of functions on an ADT is called a term if and only if it conforms to the standard syntax requirements of term algebra [17, 18]. A term can be transformed into another using the axioms of the specification as progressive left-to-right rewriting rules. A term is said to be in normal form if and only if no further axiom is applicable. A system of axioms is said to be canonical if and only if every sequence of rewriting on the same ground term (that is, a term without variables) eventually reaches a unique normal form, independent of the choice and sequence of rewriting rules used. Thus, any consistent specification should be canonical.

The following is an improved version of the concept of equivalence:

Definition 4.1. Two terms u_1 and u_2 are said to be *equivalent* (denoted by $u_1 \sim u_2$) if and only if both of them can be transformed by canonical axioms into the same normal form.

For instance, the terms "([5] | [1] | [4] | [2]).*sorting*" and "([4] | [5] | [2] | [1]).*sorting*" of Example 1 are equivalent because they can both be transformed into the same normal form [1] | [2] | [4] | [5].

Furthermore, we would like to introduce a new concept in our paper:

Definition 4.2. Given a canonical system of axioms, a pair of equivalent terms, formed by replacing all the variables on both sides of an axiom by normal forms, is called a *fundamental pair* induced from the axiom.

The following two definitions are adapted from Frankl for completeness:

Definition 4.3. Given a canonical system of axioms, the series of methods corresponding to the functions in a term is called a sequence of operations corresponding to the term. Two such sequences s_1 and s_2 are *equivalent* (denoted by $s_1 \approx s_2$) if and only if their operations on the same object give observationally equivalent results.

Definition 4.4. An implementation Ψ, which maps ADTs and functions of a canonical specification to classes and methods in a programming language, is said to be *consistent with respect to* the equivalent terms $u_1 \sim u_2$ if and only if the corresponding sequences of operations satisfy $\Psi(u_1) \approx \Psi(u_2)$.

Having defined the fundamental concepts, we recommend a strategy for test case selection: in order to test whether an implementation is consistent with respect to equivalent terms, we need only test fundamental pairs. The testing coverage of fundamental pairs is the same as that of equivalent terms. In other words, any error revealable by equivalent terms can be revealed by some fundamental pair.

For instance, in Example 1, we need only test fundamental pairs such as

([5] | [6] | [3] | [1] | [7] | [9]).*sorting* ~ ([5] | [6] | [1] | [3] | [7] | [9]).*sorting*

and

([1] | [3] | [5] | [6] | [7] | [9]).*sorting* ~ [1] | [3] | [5] | [6] | [7] | [9].

We need not test other equivalent pairs such as

$$([6] \mid [5] \mid [3] \mid [1] \mid [9] \mid [7]).sorting \sim [1] \mid [3] \mid [5] \mid [6] \mid [7] \mid [9]$$

or

$$(([6] \mid [5]).sorting \mid ([3] \mid [1]).sorting \mid ([9] \mid [7]).sorting).sorting$$
$$\sim ([5] \mid [6] \mid [3] \mid [1] \mid [7] \mid [9]).sorting.$$

Our strategy has a sound mathematical foundation, simple, and efficient. The testing coverage claim is based on Theorem 4.6 below. The following lemma is required in proving the theorem.

Lemma 4.5. Given a canonical specification, any term u_1 which is not in normal form can be transformed into a unique normal form u_0 via a series of axioms $a_1, a_2, ..., a_k$:

$$u_1 \xrightarrow{a_1} u_2 \xrightarrow{a_2} u_3 \xrightarrow{a_3} ... \xrightarrow{a_{k-1}} u_k \xrightarrow{a_k} \approx u_0$$

such that all the binding variables involved with each step of the transformation are in normal forms.

For example, the term "$([5] \mid [1] \mid [4] \mid [2]).sorting$" of BUBBLESORT in Example 1 can be transformed into its normal form $[1] \mid [2] \mid [4] \mid [5]$ as follows:

$([5] \mid [1] \mid [4] \mid [2]).sorting$
$\rightarrow ([5] \mid [1] \mid [2] \mid [4]).sorting$ {using the last axiom with $L = [5] \mid [1]$, $N = [4]$, $N' = [2]$, $L' = nil$}
$\rightarrow ([1] \mid [5] \mid [2] \mid [4]).sorting$
$\rightarrow ([1] \mid [2] \mid [5] \mid [4]).sorting$
$\rightarrow ([1] \mid [2] \mid [4] \mid [5]).sorting$
$\rightarrow [1] \mid [2] \mid [4] \mid [5]$ {using the second last axiom}

The proof of the general case is straightforward.

Theorem 4.6. For a given canonical specification, an implementation is consistent with respect to all equivalent terms if and only if it is consistent with respect to all fundamental pairs.

Proof of Theorem 4.6. Obviously if an implementation is consistent with respect to all equivalent terms, then it is consistent with respect to all fundamental pairs.

On the contrary, assume that an implementation is consistent with respect to all fundamental pairs. Let $u_1 \tilde{} u_2$ be any two equivalent terms. Suppose the mapping Ψ denotes the implementation, and suppose $\Psi(u_1) = s_1$ and $\Psi(u_2) = s_2$. We would like to prove that $s_1 \approx s_2$.

By Definition 4.1, u_1 and u_2 can be transformed into the same normal form u_0. Then there exists a series of axioms $a_1, a_2, ..., a_k$:

$$u_1 \xrightarrow{a_1} u_{12} \xrightarrow{a_2} u_{13} \xrightarrow{a_3} ... \xrightarrow{a_{k-1}} u_{1k} \xrightarrow{a_k} \approx u_0$$

satisfying Lemma 4.5.

Let $\Psi(u_{1j}) = s_{1j}$ for $j = 2, ..., k$, and $\Psi(u_0) = s_0$.

Suppose (without loss of generality) that*

(1) $u_1 = f_0(A_0).f_1(A_1) ... f_i(A_i).f_{i+1}(A_{i+1}) ... f_t(A_t)$

where A_j is either null or a tuple of parameters, containing only ground terms,

---

* Consider, for example, the class of stacks. Suppose u_1 in (1) is $new.push(4).pop.push(2)$. Then $f_0(A_0) = new$, $f_1(A_1) = push(4)$, $f_2(A_2) = pop$, $f_3(A_3) = push(2)$. Suppose a_1 in (2) is $O_x.push(I).pop = O_x$. Then O_x matches new, $push(I)$ matches $push(4)$, and pop matches pop. Hence, $u_{12} = new.push(2)$.

(2) a_1 is $O_x.f_1(X_1)...f_i(X_i) \ ... \ O_x.g_1(Y_1) \ ... \ g_j(Y_j)$

where O_x is null or an object variable, X_2, ... , X_i, Y_1, ... , Y_j are either null or tuples of parameters containing variables and/or ground terms.

Then

$$u_{12} = f_0(A_0).g_1(B_1) \ ... \ g_j(B_j).f_{i+1}(A_{i+1}) \ ... f_t(A_t)$$
$$s_1 = \Psi(f_0(A_0)).\Psi(f_1(A_1)) \ ... \ \Psi(f_i(A_i)).\Psi(f_{i+1}(A_{i+1})) \ ... \ \Psi(f_t(A_t))$$
$$s_{12} = \Psi(f_0(A_0)).\Psi(g_1(B_1)) \ ... \ \Psi(g_j(B_j)).\Psi(f_{i+1}(A_{i+1})) \ ... \ \Psi(f_t(A_t))$$

Since the series of axioms a_1, ..., a_k satisfy Lemma 4.5,

$$f_0(A_0).f_1(A_1)...f_i(A_i) \ \sim \ f_0(A_0).g_1(B_1) \ ... \ g_j(B_j)$$

is a fundamental pair induced from a_1. According to the assumption that the implementation is correct with respect to all fundamental pairs, we have

$$\Psi(f_0(A_0)).\Psi(f_1(A_1)) \ ... \ \Psi(f_i(A_i)) \ \approx \ \Psi(f_0(A_0)).\Psi(g_1(B_1)) \ ... \ \Psi(g_j(B_j)).$$

Thus,

$$\Psi(f_0(A_0)).\Psi(f_1(A_1)) \ ... \ \Psi(f_i(A_i)).\Psi(f_{i+1}(A_{i+1})) \ ... \ \Psi(f_t(A_t))$$
$$\approx \ \Psi(f_0(A_0)).\Psi(g_1(B_1)) \ ... \ \Psi(g_j(B_j)).\Psi(f_{i+1}(A_{i+1})) \ ... \ \Psi(f_t(A_t))$$

That is, $s_1 \approx s_{12}$. By the same argument, $s_{12} \approx s_{13} \approx ... \approx s_{1k} \approx s_0$. Therefore, $s_1 \approx s_0$.

Similarly, we can prove that $s_2 \approx s_0$. Hence $s_1 \approx s_2$.

5. CONTRIBUTIONS OF OUR STRATEGY

Compared with Frankl's guidelines for selecting equivalent test cases, our strategy has the following advantages:

(a) Our strategy is based on a sound theoretical foundation and mathematical proof.

(b) The domain of test case selection in Frankl's guidelines is the whole set of all pairs of equivalent terms, but the domain of selection in our strategy is only the set of fundamental pairs induced from the axioms. The original domain is a polynomial function of the set of axioms, whereas the proposed domain is only a linear function of the set. Although our domain of selection is much less than that of Frankl's, our testing coverage is the same. Hence our strategy is more efficient. Furthermore, our strategy is more precise than Frankl's guidelines, and hence easier to be performed.

(c) Our recommendation is very natural since experienced implementors would start their tests with fundamental pairs even in the absence of formal theory. Our strategy simply says that when implementors are satisfied with the test results with respect to fundamental pairs, they do not need to test other equivalent terms.

(d) Given an original term u_1, Frankl's approach searches and reduces an ADT tree to generate a term u_2 such that $u_1 \sim u_2$. "The size of the tree may be exponential in the size of the initial sequence. To deal with this complexity, the simplifier can operate either in batch mode, which builds the entire tree, or in interactive mode, which allows the user to selectively guide the construction of a subtree." [7] The interactive method can only give partial solution, whereas the batch method simply hides the complexity problem from the user. Our strategy avoids the complexity problem entirely by generating equivalent test cases directly from the two sides of each axiom, rather than searching and reducing the ADT trees.

(e) Using our strategy, two of Frankl's tools, the compiler and simplifier, can be replaced by a simple generator which induces equivalent test cases directly from the two sides of each axiom. This will

greatly simplify the system.

6. CONCLUSION

In this paper, we take a fresh look at Frankl's functional approach for the testing of object-oriented programs. We find that, although the approach has many general merits, it is based on a fallacious foundation, and the heuristics for the selection of equivalent test cases are supported only by two empirical studies. Hence the correctness and effectiveness of the guidelines are not guaranteed. We define the concept of a fundamental pair as a pair of equivalent terms which are formed by replacing all the variables on both sides of an axiom by normal forms. We prove that an implementation is consistent with respect to all equivalent terms if and only if it is consistent with respect to all fundamental pairs. In other words, the testing coverage of fundamental pairs is identical to that of all equivalent terms, and hence we need only concentrate on the testing of fundamental pairs. We have reduced the domain of test cases from a polynomial function of the set of axioms to a linear function. Our strategy is mathematically based, simple, and much more efficient. Furthermore, it underscores the usefulness of axiom-based specifications.

We note that there may be infinitely many fundamental pairs induced from the same axiom by assigning different normal forms to the variables. Exhaustive testing is of course impossible. How should we select the fundamental pairs, and how does the selection affect the testing coverage? How do we verify whether two resulting objects are observationally equivalent? We are investigating into the application of regularity and uniformity hypotheses as proposed in [15, 16] to further enhance our strategy.

REFERENCES

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

[2] Kung, D.C., Gao, J.Z., Lin, J., Hsia, P., and Toyoshima, Y.: Design recovery for software testing of object-oriented programs, Proceedings of the IEEE Working Conference on Reverse Engineering, IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 202−211.

[3] Cheatham, T.E., Jr. and Mellinger, L.: Testing object-oriented software systems, Proceedings of the 18th ACM Annual Computer Science Conference (CSC 1990), ACM Press, New York, NY, 1990, pp. 161−165.

[4] Doong, R.-K. and Frankl, P.G.: Case studies on testing object-oriented programs, Proceedings of the 4th ACM Annual Symposium on Testing, Analysis, and Verification (TAV 4), ACM Press, New York, NY, 1991, pp. 165−177.

[5] Fiedler, S.P.: 'Object-oriented unit testing', Hewlett-Packard Journal, 1989, 40 (4): pp. 69−74.

[6] Firesmith, D.G.: Testing object-oriented software, Proceedings of the 11th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 11), Prentice Hall, Englewood Cliffs, NJ, 1993, pp. 407−426.

[7] Frankl, P.G. and Doong, R.-K.: Tools for testing object-oriented programs, Proceedings of the 8th Pacific Northwest Conference on Software Quality, 1990, pp. 309−324.

[8] Frankl, P.G.: A framework for testing object-oriented programs, Computer Science Technical Report PUCS-105-91: Department of Electrical Engineering and Computer Science, Polytechnic University, Brooklyn, NY, 1991.

[9] Harrold, M.J., McGregor, J.D., and Fitzpatrick, K.J.: Incremental testing of object-oriented class structures, Proceedings of the 14th International Conference on Software Engineering (ICSE 1992), ACM Press, New York, NY, 1992, pp. 68−80.

[10] Hoffman, D.M. and Strooper, P.A.: 'Graph-based class testing', Australian Computer Journal, 1994, 26 (4): pp. 158−163.

[11] Perry, D.E. and Kaiser, G.E.: 'Adequate testing and object-oriented programming', Journal of Object-Oriented Programming, 1990, 2 (5): pp. 13−19.

[12] Smith, M.D. and Robson, D.J.: Object-oriented programming: the problems of validation, Proceedings of the 6th IEEE International Conference on Software Maintenance (ICSM 1990), IEEE Computer Society Press, Los Alamitos, CA, 1990, pp. 272−281.

[13] Turner, C.D. and Robson, D.J.: State-based testing and inheritance, Technical Report TR-1/93, Computer Science Division, School of Engineering and Computer Science, University of Durham, Durham, UK, 1993.

[14] Turner, C.D. and Robson, D.J.: 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.

[15] Bouge, L., Choquet, N., Fribourg, L., and Gaudel, M.-C.: 'Test sets generation from algebraic specifications using logic programming', Journal of Systems and Software, 1986, 6: pp. 343−360.

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

[17] Ehrig, H. and Mahr, B.: Fundamentals of Algebraic Specifications 1: Equations and Initial Semantics, EATCS Monographs on Theoretical Computer Science, Vol. 6, Springer, Berlin, Germany, 1985.

[18] Goguen, J.A., Thatcher, J.W., and Wagner, E.G.: An initial algebra approach to specification, correctness and implementation of abstract data types, Current Trends in Programming Methodology, vol. IV: Data Structuring, R.T. Yeh (ed.), Prentice Hall, Englewood Cliffs, NJ, 1978, pp. 80−149.

[19] Goguen, J.A. and Winkler, T.: Introducing OBJ, Technical Report SRI-CSL-88-9, Computer Science Laboratory, SRI International, Menlo Park, CA, 1988.