Postprint of article in *IEEE Transactions on Software Engineering* **39** (11): 1549–1563 (2013)

Equality to Equals and Unequals: A Revisit of the Equivalence and Nonequivalence Criteria in Class-Level Testing of Object-Oriented Software*

Huo Yan Chen and T.H. Tse, *Senior Member, IEEE*

Abstract—Algebraic specifications have been used in the testing of object-oriented programs and received much attention since the 1990s. It is generally believed that class-level testing based on algebraic specifications involves two independent aspects: the testing of equivalent and nonequivalent ground terms. Researchers have cited intuitive examples to illustrate the philosophy that even if an implementation satisfies all the requirements specified by the equivalence of ground terms, it may still fail to satisfy some of the requirements specified by the *non*equivalence of ground terms. Thus, both the testing of equivalent ground terms and the testing of nonequivalent ground terms have been considered as significant and cannot replace each other.

In this paper, we present an innovative finding that, given any canonical specification of a class with proper imports, a complete implementation satisfies all the observationally equivalent ground terms if and only if it satisfies all the observationally nonequivalent ground terms. As a result, these two aspects of software testing cover each other and can therefore replace each other. These findings provide a deeper understanding of software testing based on algebraic specifications, rendering the theory more elegant and complete. We also highlight a couple of important practical implications of our theoretical results.

Index Terms—Software testing, equivalence criterion, nonequivalence criterion, algebraic specification, object-oriented software

1 Introduction

As a major formal method for defining the functional requirements of object-oriented software, algebraic specifications are very useful in the testing of their implementations with many benefits, including improvements in the automation and effectiveness of test case generation.

In particular, class-level testing of object-oriented software based on algebraic specifications has been studied extensively. Previous work treats the testing of the correctness of implementation in two aspects: whether two sequences of operations (known formally as *ground terms*) that are proven to be equivalent according to the specification will result in equivalent objects in the implementation, and whether two sequences of operations that are proven to be nonequivalent according to the specification will result in nonequivalent objects in the implementation.

Previous research argues that, generally speaking, the testing of all equivalent ground terms is not sufficient to reveal the possible failures due to all the faults in the implementation. The testing of all nonequivalent ground terms is not sufficient either. Hence, one must conduct both kinds of tests. Please refer to Example 2 in Section 4 for a commonly adopted intuitive illustration.

In this paper, we present our innovative finding that, given a canonical specification of a class with proper imports and a complete implementation, the testing of observationally equivalent ground terms and the testing of observationally nonequivalent ground terms cover each other.[1] In other words, if a failure due to a certain fault in the implementation can be revealed by testing observational nonequivalence, another failure due to the same fault can also be revealed by testing observational equivalence, and vice versa. As a result, contrary to the general belief in previous work, we do *not* need to conduct both kinds of tests. This finding deepens the understanding of software testing based on algebraic

- *Huo Yan Chen is with the Department of Computer Science, Jinan University, Guangzhou, 510632, China.*
- *T.H. Tse is with the Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong. Email: thtse@cs.hku.hk.*

[1] Thus, we attain "equality to equals and unequals," quoted from Plato, *The Republic* (380 BC).

specifications, and renders the theory more elegant and complete. Furthermore, the theory has important practical implications in real-world software testing. The remaining sections describe our new theoretical and practical findings in detail.

This paper is organized as follows: Section 2 describes previous work related to the testing of object-oriented software based on algebraic specifications. Section 3 outlines the basic concepts used in the paper. Section 4 investigates an innovative relationship between the equivalence criterion and the nonequivalence criterion. Section 5 applies the results of Section 4 to class-level testing of object-oriented software and gives examples for illustration. Section 6 discusses the theoretical implications in relation to previous work on class-level testing based on algebraic specifications. Section 7 highlights two important practical implications. Section 8 concludes the paper.

2 Related Previous Work

The idea of algebraic specifications originated from the work of Zilles [33], Goguen et al. [21], and Guttag and Horning [22]. There have been numerous proposals. For example, Goguen and others introduced *OBJ3* [20] and extended it to *FOOPS* [6] for the object-oriented paradigm, while Bidoit and Mosses [5] designed the Common Algebraic Specification Language (CASL), which consolidates various algebraic specification features into a general purpose language.

A general theory for software testing based on algebraic specifications was proposed by Bernot et al. [3], [4]. It includes a regularity hypothesis, a uniformity hypothesis, and other oracle hypotheses to formalize the oracle problem. The main advantage of the framework is to express in explicit terms the abilities and limitations of testing as well as the gap between the testing and proving of programs. Based on the framework, a tool has been developed to generate test cases by replacing all the variables in the axioms of the specification by ground terms according to the uniformity and regularity hypotheses [7], [14].

The DAISTS approach by Gannon et al. [17] suggests selecting a tuple of argument values as inputs to the left- and right-hand sides of an axiom, and then invokes a user-supplied equality function to examine the outputs. A failure is revealed if the outputs from both sides are not equal. The testing approach proposed by Antoy and Hamlet [2] uses proof techniques. Unlike DAISTS, they test the equality of values of abstract date types in the abstract domain rather than in the concrete domain.

Machado [27] proposed oracles derived from flat algebraic specifications expressed in first-order logic. Machado [28] presented an extension of the framework in Bernot et al. [3], [4] to first-order logic with restrictions on quantifiers. Machado and Sannella [29] further extended the framework for CASL architectural specifications.

Jalote [24] considered the axioms as rewrite rules, suggested choosing test cases from all legal combinations of operations (that is, all terms), and derived corresponding equivalent terms by means of the rewrite rules. They checked whether the execution results corresponding to every selected pair of equivalent terms are equivalent. If not, a failure is revealed.

To facilitate testing object-oriented software at the class level, Doong and Frankl [15], [16] proposed *LOBAS*, an algebraic specification language whose syntax is similar to object-oriented programming languages and whose semantics is similar to that of *OBJ3* and *FOOPS*.
The *LOBAS* language for algebraic specifications proposed by Doong and Frankl [15], [16] is more suitable for the class-level testing of object-oriented software. Based on *LOBAS*, they extended the work of Jalote [24] and developed a tool known as *ASTOOT* to generate test cases from pairs of equivalent terms through rewriting, and from pairs of nonequivalent terms by "exchanging path conditions."

In algebraic specifications, a ground term is said to be a *normal form* if and only if it cannot be further rewritten to another term using any axiom in the given specification. In our earlier work [9], [10], two ground terms are said to be *equivalent* if and only if both of them can be written to the same normal form according to the given canonical specification. If a pair of ground terms is formed by replacing all the variables on both sides of an axiom by normal forms of the given specification, we refer to them as a *fundamental pair of equivalent terms* or simply a *fundamental pair* (denoted by "$u_1 \sim_{fun} u_2$"). We proved in [9] that, given any canonical specification of a class with proper imports, a complete implementation satisfies all the fundamental pairs if and only it satisfies all pairs of observationally equivalent ground terms.

Further descriptions of the *TACCLE* methodology and the results of [9], [10] are given in Sections 3 and 6.

According to previous work such as [10], [16], class-level testing based on algebraic specifications involves two aspects, namely, the testing of observationally equivalent ground terms and the testing of observationally nonequivalent ground terms. Intuitive examples[2] have been used to illustrate the philosophy that even if the testing of observationally equivalent ground terms is exhaustive, it cannot reveal failures in which two different states are being confused as a single state, that is, in which two observationally nonequivalent ground terms erroneously generate two equivalent objects [10], [16]. It has been pointed out therefore that the testing of observationally nonequivalent ground terms is necessary and cannot be ignored even after exhaustive testing of observationally equivalent ground terms.

Hierons et al. [23] and Zhu [32] have surveyed these aspects. Kong et al. [25] and Yu et al. [31] extended the algebraic testing approach to cover software components, and presented an automated testing tool called *CASCAT* for Java components.

[2] Such as Example 2 in Section 4, which is refuted by Examples 3 and 4 in Section 4.

3 Basic Concepts of Class-Level Testing of Object-Oriented Software Based on Algebraic Specifications

This paper considers the class-level testing of object-oriented software based on algebraic specifications. In order that this paper may be self-contained, we first summarize the basic concepts of algebraic specifications. Readers may refer to existing work [9], [10], [16] for more details of algebraic specifications for object-oriented software. For fairness of comparison with previous work, we will use the same level of constraint on algebraic specifications as that in Aiguier et al. [1], Doong and Frankl [15], [16], and Chen et al. [9], [10]. Readers may also refer to Section 4.6 of [10] for an explanation of why the constraints are reasonably justified in object-oriented software testing. We concede that, like the related work on the testing of object-oriented software based on algebraic specifications, we cannot cover nondeterministic systems using canonical specifications. While this is an interesting topic that is worth further research, it is beyond the scope of the present paper.

An *algebraic specification* of a class consists of two parts. The first part defines the *syntax* of the class by declaring the *operations* in terms of their input and output classes. The second part defines the *semantics* of the class by listing the *axioms* in the form of *conditional equations*, which describe the functional requirements of the operations. The following is an example of an algebraic specification. For ease of reading, we will reuse the same example for different purposes in the paper.

Example 1 (adapted from Chen et al. [9]). Algebraic specification of a class *IntStack* of stacks of integers.

module *INTSTACK*
class *IntStack*
import classes *Int* // the class of integers
Bool // the class of Boolean values
operations
new: → *IntStack*
_.isEmpty: *IntStack* → *Bool*
_.push(*_*): *IntStack Int* → *IntStack*
_.pop: *IntStack* → *IntStack*
_.top: *IntStack* → *Int* ∪ {*nil*}
variables
S: *IntStack*; *N*: *Int*
axioms
a_1: *new.isEmpty* = *true*
a_2: *S.push*(*N*).*isEmpty* = *false*
a_3: *new.pop* = *new*
a_4: *S.push*(*N*).*pop* = *S*
a_5: *S.top* = *nil if S.isEmpty*
a_6: *S.push*(*N*).*top* = *N*

The statement "*_.push*(*_*): *IntStack Int* → *IntStack*" means that the operation *push* has an input class *IntStack*, a parameter class *Int*, and an output class *IntStack*.

Given an algebraic specification, a syntactically valid sequence of operations is called a *term*. A term without variables is known as a *ground term*. In Example 1, for instance, *S.push*(*N*).*pop.top* is a term and *new.push*(2).*pop.top* is a ground term.

Suppose a ground term u contains a subterm v that is an instance of the left-hand side v_i of an axiom a_i: $v_i = v_i'$. If we replace the subterm v by the corresponding instance v' of the right-hand side v_i', and if the result is a ground term u', then u is said to be *rewritten* into u' using axiom a_i as a *left-to-right rewrite rule*. For instance, a ground term *new.push*(2).*pop.top* in Example 1 can be rewritten to *new.top* using axiom a_4 as a rewrite rule.

Let $u_1, u_2, ..., u_n$ be ground terms. If u_1 can be rewritten to u_2, and u_2 can be rewritten to u_3, and …, and u_{n-1} can be rewritten to u_n, then we say that there is a *rewriting relation* from u_1 to u_n (denoted by "$u_1 \sim_{rew} u_n$").

We say that a ground term is in a *normal form* if and only if it cannot be further rewritten by any axiom in the specification. An algebraic specification is said to be *canonical* if and only if every ground term can be rewritten to *a unique* normal form through a *finite* number of rewrites.

We refer to the specified functions in a specification as *operations* and the implemented functions in a program as *methods*. The operations (or methods) that create objects in a class C are called the *creators* of C. An *attribute* of an object is a visible property of that object. For any given object, the collection of all the attributes and their respective values is called the *state* of the object. The operations (or methods) that return the values of attributes of objects in C without any change of state are called the *observers* of C. The operations (or methods) that change the states of objects in C are called the *constructors* or *transformers* of C. In particular, a constructor of an object can remain in a normal form after transformations by axioms, whereas a transformer will ultimately be eliminated by some rewrite rule and cannot appear in a normal form.

For any operation or method $\_.f(\_, \_, ..., \_)$: $C\ C_1\ C_2\ ...\ C_n \rightarrow D$, we refer to C as the *input class* of f, D as the *output class* of f, and $C_1, C_2, ..., C_n$ as the *parameter classes* of f. Let f and g be operations (or methods). If the output class of f is the same as the input class of g, we say that g is *applicable to* f and that $f.g$ is *syntactically valid*. Let f be the last operation in a term u (or the last method in a method sequence u) and g be the first operation in a term v (or the first method in a method sequence v). If g is *applicable to* f, we say that v is *applicable to* u and that $u.v$ is *syntactically valid*.

An *observable context* on a class C is either an observer or a syntactically valid sequence of constructors or transformers in C that ends with an observer.

Consider a canonical specification of a given class. Two ground terms u_1 and u_2 are said to be *normally equivalent* (denoted by "$u_1 \sim_{nor} u_2$") if and only if both of them have the same normal form. Two ground terms u_1 and u_2 are said to be *observationally equivalent* (denoted by "$u_1 \sim_{obs} u_2$") if and only if 1) $u_1.oc$ and $u_2.oc$ are observationally equivalent for any applicable observable context oc, and 2) u_1 and u_2 are normally equivalent when there is no applicable observable context oc. (Notice that this definition is recursive.) Two ground terms u_1 and u_2 are said to be *attributively equivalent* (denoted by "$u_1 \sim_{att} u_2$") if and only if 1) $u_1.ob$ and $u_2.ob$ are observationally equivalent for any applicable observer ob, and 2) u_1 and u_2 are normally equivalent when there is no applicable observer ob. From

the definition, normal equivalence is a special case of observational equivalence, which is a special case of attributive equivalence.

Consider a canonical specification and an implementation of a given class. Two objects O_1 and O_2 are said to be *observationally equivalent* (denoted by "$O_1 \approx_{obs} O_2$") if and only if 1) $O_1.oc$ and $O_2.oc$ are observationally equivalent objects for any applicable observable context oc, and 2) O_1 and O_2 are identical objects (denoted by $O_1 = O_2$) when there is no applicable observable context oc. Two objects O_1 and O_2 are said to be *attributively equivalent* (denoted by "$O_1 \approx_{att} O_2$") if and only if 1) $O_1.ob$ and $O_2.ob$ are observationally equivalent objects for any applicable observer ob, and 2) O_1 and O_2 are identical objects when there is no applicable observer ob.

If the output class of the last operation in the term u is a *primitive type* (that is, it corresponds to a built-in type in the implementation language, such as int in C++), then the normal form of u is a constant k of the primitive type. In such case, we write "$u \sim_{nor} k$," "$u \sim_{obs} k$," and "$u \sim_{att} k$" in the specification as "$u = k$," where "=" denotes equality in the primitive type. Similarly, we write "$O_1 \approx_{obs} k$" and "$O_1 \approx_{att} k$" in the implementation as "$O_1 = k$."

Given an algebraic specification of a class C, suppose oc_i (i = 1, 2, ..., n) is an observable context on C and every oc_j (j = 2, 3, ..., n) is applicable to oc_{j-1}. Then, the sequence $oc_1.oc_2.\cdots.oc_n$ is called an *observable context sequence* (or simply an *oc sequence*) on C, and its *length* is said to be n. If the output class of oc_n is a primitive type (that is, a built-in type in the implementation language), then $oc_1.oc_2.\cdots.oc_n$ is called a *primitive oc sequence* on C.

A module in an algebraic specification may *import* known classes to support the functional requirements of a specified class. We say that the specification of the class has *proper imports* if and only if every oc sequence is of finite length and can be extended to a primitive oc sequence in a finite number of steps. We say that an implementation P of a class is *complete* with respect to an algebraic specification Sp if and only if every operation f in Sp is implemented by a unique method $\Theta(f)$ in P such that the output class or type of $\Theta(f)$ is consistent with the output class or type of f, and the constants in P are consistent with the constants in Sp.

For a complete implementation, given any ground term u in Sp, we use $\Theta(u)$ to denote the unique object that results from executing the implemented method sequence corresponding to u. In particular, for a complete implementation, any constant k in Sp corresponds to a constant k in P. We write $\Theta(k) = k$.

Given a canonical specification Sp of a class with proper imports and a complete implementation P, we define P to be *correct with respect to Sp* if and only if *both* of the following criteria are satisfied:

1. **Equivalence Criterion.** For any pair of observationally equivalent terms u_1 and u_2, the objects $\Theta(u_1)$ and $\Theta(u_2)$ that result from executing the corresponding implemented method sequences are observationally equivalent. That is, $(\forall u_1)(\forall u_2)((u_1 \sim_{obs} u_2) \rightarrow (\Theta(u_1) \approx_{obs} \Theta(u_2)))$. In this case, we also say P *satisfies all the observationally equivalent ground pairs* specified in Sp.
2. **Nonequivalence Criterion.** For any pair of observationally nonequivalent terms v_1 and v_2, the objects $\Theta(v_1)$ and $\Theta(v_2)$ that result from executing the corresponding implemented method sequences are observationally nonequivalent. That is, $(\forall v_1)(\forall v_2)(\neg(v_1 \sim_{obs} v_2) \rightarrow \neg(\Theta(v_1) \approx_{obs} \Theta(v_2)))$. In this case, we also say P *satisfies all the observationally nonequivalent ground pairs* specified in Sp.

We say that a *failure* of P with respect to Sp is *revealed* if either of the above criteria is negated, that is, if one of the following conditions is satisfied:

1′. Equivalence Failure Criterion. $(\exists u_1)(\exists u_2)((u_1 \sim_{obs} u_2) \wedge \neg(\Theta(u_1) \approx_{obs} \Theta(u_2)))$.

2′. Nonequivalence Failure Criterion. $(\exists v_1)(\exists v_2)(\neg(v_1 \sim_{obs} v_2) \wedge (\Theta(v_1) \approx_{obs} \Theta(v_2)))$.

4 New Relationship between the Equivalence and Nonequivalence Criteria

Class-level testing based on algebraic specifications involves *two aspects*, namely, the testing of observationally equivalent ground terms and the testing of observationally nonequivalent ground terms. Two important questions immediately arise: What is the relationship between these two aspects? How much does one cover the other?

Previous work such as Doong and Frankl [16], Gaudel [18], Chen et al. [10], and Zhu [32] argues that successful exhaustive testing of one aspect does *not* entail successful testing of the other. As pointed out by Gaudel [18], "The definition of [the set of all ground instances of all the axioms] $Exhaust_{Sp}$ comes from the notion of satisfaction of [21]. However, it does not correspond exactly to initial semantics of algebraic specifications *since inequalities are not tested:* it rather corresponds to loose semantics. ... In [16], a bigger exhaustive test set is mentioned which includes for every ground term the inequalities with other normal forms, following the definition of initial semantics." Zhu [32] further recognizes that "One of [Doong and Frankl's] most important contributions ... is the extension of test cases to include negative test cases, which consists of two terms that are supposed to generate non-equivalent results."

The following is an intuitive example that is representative of this philosophy.

> **Example 2** (adapted from Doong and Frankl [16] and Chen et al. [10]). Suppose none of the operations changes the states of objects in a faulty implementation. Given any two observationally equivalent ground terms, the corresponding objects returned by the implementation will, of course, be equivalent. Intuitively, the failure cannot be revealed by testing equivalent terms only. One can conclude, therefore, that the testing of observationally nonequivalent ground terms is arguably necessary and cannot be ignored.

Although Example 2 appears intuitively to be valid, it is, in fact, not always the case. We can construct a simple counterexample as follows:

Example 3. Consider the algebraic specification of the class *IntStack* in Example 1 again. Suppose an implementation of this specification is as follows, where *array*[1] is the top of the stack and *array*[100] is the bottom.

```
# include <iostream>
# define SIZE 100
# define NIL 0
class intStack {
  int array[SIZE];
  public: ...
};
...
void intStack :: newStack() {
  for (int j = 1; j <= 100; j ++)
    array[j] = NIL;
}
void intStack :: push(int N) {
  for (int j = 100; j > 1; j – –)
    array[j] = array[j – 1];
  array[1] = NIL;
  /* There is a fault in the above statement. */
  /* It should be "array[1] = N;". */
}
void intStack :: pop () {
  for (int j = 1; j < 100; j++)
    array[j] = array[j + 1];
  array[100] = NIL;
}
int intStack :: top () {
  return array[1];
}
```

As a result of the fault in the implemented method *push*(*int N*), the states of all the objects in the implemented class are always (*NIL*, *NIL*, …, *NIL*), satisfying the precondition that "none of the operations changes the states of objects" of Example 2. However, the conclusion that "the failure cannot be revealed by testing equivalent terms only" of Example 2 is not true, because a pair of failure-revealing equivalent terms can be constructed thus:

Substituting *S* by "*new*" and *N* by any value (such as 6) in the axiom a_5: *S*.*push*(*N*).*top* = *N*, we obtain the following fundamental pair of equivalent terms:

new.*push*(6).*top* ~ 6

Under the above implementation, these two terms return different observable results, namely, "*nil*" and "6", respectively. Hence, this fundamental pair reveals a failure due to the fault in the implemented method *push*(*int N*). Thus, Example 2 is refuted.

Following standard practice in software testing, we use the term "fault" to mean an incorrect instruction in the program and the term "failure" to mean an incorrect execution result. In Example 3, for instance, *array*[1] = *NIL* is a fault and Θ(*new*.*push*(6).*top*) $\approx_{obs}$ Θ(*new*.*push*(8).*top*) is a failure due to that fault. In general, a fault may cause more than one failure. For instance, Θ(*new*.*top*) $\approx_{obs}$ Θ(*new*.*push*(4).*top*) is another failure due to the same fault. We note also that in Example 3 as well as Examples 4 and 5 that follow, we assume that there is a built-in type *int* in the implementation language for the type *Int* in the specification. This assumption is reasonable.

On one hand, Example 3 serves as a counterexample to refute the intuitive Example 2. On the other hand, Example 3 shows only one particular scenario where, given an observationally nonequivalent pair that reveals a failure, there is an observationally equivalent pair that also reveals a failure due to the same fault. To take a broader view, let us consider a slightly more general Example 4 first, and then follow up with generalized lemmas, theorem, and proofs.

Example 4. Take the algebraic specification of the class *IntStack* in Example 1 and the implementation in Example 3 again. Consider a pair of ground terms *new*.*push*(1).*push*(3).*push*(5) and *new*.*push*(2).*push*(3).*push*(5). They exhibit the following properties:

(a) There is an observable context in the specification, namely, *pop*.*pop*.*top*, which operates on the two ground terms to give different results:
new.*push*(1).*push*(3).*push*(5).*pop*.*pop*.*top* = 1 (4.1)
new.*push*(2).*push*(3).*push*(5).*pop*.*pop*.*top* = 2 (4.2)
Hence, the two original ground terms are observationally nonequivalent.

(b) Based on the two ground terms in the specification, two sequences of methods in the implementation in Example 3 will be executed, namely,
Θ(*new*).Θ(*push*(1)).Θ(*push*(3)).Θ(*push*(5)) and
Θ(*new*).Θ(*push*(2)).Θ(*push*(3)).Θ(*push*(5)).
The two objects that result from executing these sequences of methods corresponding to the two ground terms are observationally equivalent because they are both observationally equivalent to Θ(*new*). Hence, the two ground terms *new*.*push*(1).*push*(3).*push*(5) and *new*.*push*(2).*push*(3).*push*(5) reveal a failure. In particular, the observable context in the implementation corresponding to *pop*.*pop*.*top*, namely, Θ(*pop*).Θ(*pop*).Θ(*top*), operates on the two objects to give the same result:
Θ(*new*).Θ(*push*(1)).Θ(*push*(3)).Θ(*push*(5)).
Θ(*pop*).Θ(*pop*).Θ(*top*) = 0 (4.3)
Θ(*new*).Θ(*push*(2)).Θ(*push*(3)).Θ(*push*(5)).
Θ(*pop*).Θ(*pop*).Θ(*top*) = 0 (4.4)

(c) At least one of the right-hand values in (4.1) and (4.2) for the specification is inconsistent with the right-hand value in (4.3) and (4.4) for the implementation. In this particular example, Θ(1) = 1 and Θ(2) = 2, which are both different from 0. Without loss of generality, consider (4.1). It can be regarded as a pair of observationally equivalent ground terms
new.*push*(1).*push*(3).*push*(5).*pop*.*pop*.*top* and 1.
However, their corresponding objects are nonequivalent:
Θ(*new*).Θ(*push*(1)).Θ(*push*(3)).Θ(*push*(5)).
Θ(*pop*).Θ(*pop*).Θ(*top*) = 0 but
Θ(1) ≠ 0.

Hence, they also reveal a failure due to the same fault.

In summary, based on a given pair of observationally nonequivalent ground terms *new.push*(1).*push*(3).*push*(5) and *new.push*(2).*push*(3).*push*(5) that reveal a failure, we have identified a pair of observationally equivalent ground terms *new.push*(1).*push*(3).*push*(5).*pop.pop.top* and 1 that also reveal a failure due to the same fault.

Let us consider also the reverse scenario.

Example 5. Take the algebraic specification of the class *IntStack* in Example 1 again. Consider another faulty implementation in which *pop* is implemented as a null operation that does nothing while other operations are implemented correctly. The pair of observationally equivalent ground terms *new.push*(1) and *new.push*(1).*push*(3).*pop* reveal a failure because they are observationally equivalent but produce different objects in the given implementation. They also exhibit the following properties:

(a) There is an observable context in the specification, namely, *top*, which operates on the two ground terms to give identical results

new.push(1).*top* = 1 (4.5)

new.push(1).*push*(3).*pop.top* = 1 (4.6)

(b) The two objects that result from executing the operations in the implementation corresponding to *new.push*(1) and *new.push*(1).*push*(3).*pop* (namely, Θ(*new*).Θ(*push*(1)) and Θ(*new*).Θ(*push*(1)).Θ(*push*(3)).Θ(*pop*)) are observationally nonequivalent because the corresponding observable context in the implementation (namely, Θ(*top*)) operates on the two objects to give different results

Θ(*new*).Θ(*push*(1)).Θ(*top*) = 1 (4.7)

Θ(*new*).Θ(*push*(1)).Θ(*push*(3)).Θ(*pop*).Θ(*top*) = 3 (4.8)

(c) At least one of the right-hand values in (4.7) and (4.8) for the implementation is inconsistent with the right-hand value in (4.5) and (4.6) for the specification. In this particular example, (4.8) is inconsistent with the specification. Let us take the pair of observationally nonequivalent ground terms *new.push*(1).*push*(3).*pop.top* (that is, the left-hand side of (4.6)) and 3. Their corresponding objects

Θ(*new*).Θ(*push*(1)).Θ(*push*(3)).Θ(*pop*).Θ(*top*) and Θ(3)

are observationally equivalent by (4.8). Hence, they also reveal a failure due to the same fault.

In short, based on a given pair of observationally equivalent ground terms *new.push*(1) and *new.push*(1).*push*(3).*pop* that reveal a failure, we can identify a pair of observationally nonequivalent ground terms *new.push*(1).*push*(3).*pop.top* and 3 that also reveal a failure due to the same fault.

Is there a specific class of specifications and implementations such that if an observationally nonequivalent ground pair reveals a failure due to a certain fault, then there exists an observationally equivalent ground pair that will also reveal a failure due to the same fault, and vice versa? That is, under what conditions will the testing of observationally equivalent ground terms and observationally nonequivalent ground terms cover each other? The following lemmas and theorem help answer the questions.

Lemma 1. Given a canonical specification Sp of a class with proper imports and a complete implementation P, if P satisfies all the observationally equivalent ground pairs specified in Sp, it will also satisfy all the observationally nonequivalent ground terms specified in Sp. Formally, given such a specification and implementation,

$$(\forall u_1)(\forall u_2)((u_1 \sim_{obs} u_2) \rightarrow (\Theta(u_1) \approx_{obs} \Theta(u_2))) \Rightarrow (\forall u_1)(\forall u_2)(\neg(u_1 \sim_{obs} u_2) \rightarrow \neg(\Theta(u_1) \approx_{obs} \Theta(u_2))). \quad (4.9)$$

The following is the formal proof of Lemma 1. Example 4 above illustrates its train of thought.

Proof. We prove the lemma by reductio ad absurdum. Assume that the left-hand side of (4.9) is true but the right-hand side is false, that is, $(\exists u_1)(\exists u_2)(\neg(u_1 \sim_{obs} u_2) \wedge (\Theta(u_1) \approx_{obs} \Theta(u_2)))$. In this case, $\Theta(u_1) \approx_{obs} \Theta(u_2)$ is caused by an implementation fault, which we will denote by f.

Since $\neg(u_1 \sim_{obs} u_2)$ and the given class has proper imports, according to the definition of observational equivalence $\sim_{obs}$ and the definition of proper imports, there exists an oc sequence ocs on the specification such that the output class of the observer at the end of the ocs is a primitive type, and $u_1.ocs \neq u_2.ocs$.[3] Hence, there are two values k_1 and k_2 of the primitive type such that $k_1 \neq k_2$, $u_1.ocs = k_1$, and $u_2.ocs = k_2$. As the implementation is complete, there is a unique method sequence $\Theta(ocs)$ that implements ocs. Since the class has proper imports and the implementation is complete, $\Theta(ocs)$ must also end with a primitive type, so that $\Theta(u_1).\Theta(ocs) = k$ for some value k of the primitive type. Because $\Theta(u_1) \approx_{obs} \Theta(u_2)$, according to the definition of observational equivalence $\approx_{obs}$, we have $\Theta(u_2).\Theta(ocs) = \Theta(u_1).\Theta(ocs) = k$.

As $k_1 \neq k_2$, we must have $k \neq k_1$ or $k \neq k_2$. Without loss of generality, suppose $k \neq k_1$. In that case, we have $u_1.ocs = k_1$ but $\Theta(u_1).\Theta(ocs) = k \neq k_1$.

Let $\Theta(u_1.ocs)$ denote the execution result of implemented method sequence corresponding to the specified operation sequence $u_1.ocs$. Since the implementation is complete, we have $\Theta(u_1.ocs) = \Theta(u_1).\Theta(ocs) = k \neq k_1$. Let $\Theta(k_1)$ denote the execution result of k_1. Because k_1 is the value of a primitive type, we have $\Theta(k_1) = k_1 \neq k$. Based on these relations, we obtain $u_1.ocs = k_1$ but $\Theta(u_1.ocs) \neq \Theta(k_1)$.

According to the definition of observational equivalence, as the output class of the operation at the end of $u_1.ocs$ is a primitive type, $u_1.ocs = k_1$ means $u_1.ocs \sim_{obs} k_1$ and $\Theta(u_1.ocs) \neq \Theta(k_1)$ means $\neg(\Theta(u_1.ocs) \approx_{obs} \Theta(k_1))$. (See the basic concept of primitive types in Section 3.) Thus, we have $(u_1.ocs \sim_{obs} k_1) \wedge \neg(\Theta(u_1.ocs) \approx_{obs} \Theta(k_1))$. This contradicts the left-hand side of (4.9).

[3] If u_1 or u_2 ends with an observer of a primitive type, then ocs will be empty. Consider, for instance, u_1 = *new.push*(1).*top* and u_2 = 2 in Example 1. Since *new.push*(1).*top* ≠ 2 according to axiom a_6, we have $\neg(u_1 \sim_{obs} u_2)$. See the explanation of primitive types in Section 3. In this case, u_1 ends with an observer *top* and hence ocs is empty.

We note from the above that $\neg(\Theta(u_1.ocs) \approx_{obs} \Theta(k_1))$ is derived from $\Theta(u_1) \approx_{obs} \Theta(u_2)$, which is caused by the fault *f*. In other words, $\neg(\Theta(u_1.ocs) \approx_{obs} \Theta(k_1)$ and $\Theta(u_1) \approx_{obs} \Theta(u_2)$ are due to the same fault *f*. □

The converse of Lemma 1 is also true, as stated as follows:

> **Lemma 2**. Given a canonical specification *Sp* of a class with proper imports and a complete implementation *P*, if *P* satisfies all observationally nonequivalent ground pairs specified in *Sp*, it will also satisfy all observationally equivalent ground pairs specified in *Sp*. Formally, given such a specification and implementation,
>
> $$(\forall u_1)(\forall u_2)(\neg(u_1 \sim_{obs} u_2) \rightarrow \neg(\Theta(u_1) \approx_{obs} \Theta(u_2))) \Rightarrow (\forall u_1)(\forall u_2)((u_1 \sim_{obs} u_2) \rightarrow (\Theta(u_1) \approx_{obs} \Theta(u_2))). \quad (4.10)$$

The following is the formal proof of Lemma 2. Example 5 above illustrates its train of thought.

Proof. We also prove the lemma by reductio ad absurdum. Assume that the left-hand side of (4.10) holds but the right-hand side is false, that is, $(\exists u_1)(\exists u_2)((u_1 \sim_{obs} u_2) \wedge \neg(\Theta(u_1) \approx_{obs} \Theta(u_2)))$.

Since $\neg(\Theta(u_1) \approx_{obs} \Theta(u_2))$ and the given class has proper imports, according to the definition of observational equivalence $\approx_{obs}$ and the definition of proper imports, there exists an oc sequence $\Theta(ocs)$ in the implementation such that the output class of the observer at the end of $\Theta(ocs)$ is a primitive type, and $\Theta(u_1).\Theta(ocs) \neq \Theta(u_2).\Theta(ocs)$.[4] In other words, $\Theta(u_1).\Theta(ocs) = k_1$ and $\Theta(u_2).\Theta(ocs) = k_2$ for some values k_1 and k_2 of a primitive type such that $k_1 \neq k_2$.

As the implementation is complete, there exists a unique operation sequence *ocs* implemented by $\Theta(ocs)$. Since the class has proper imports and the implementation is complete, *ocs* must also end with a primitive type, so that $u_1.ocs = k$ for some value *k* of the primitive type. Because $u_1 \sim_{obs} u_2$, according to the definition of observational equivalence $\sim_{obs}$, we have $u_2.ocs = u_1.ocs = k$. As $k_1 \neq k_2$, we must have $k \neq k_1$ or $k \neq k_2$. Without loss of generality, suppose $k \neq k_1$. In such case, we have $u_1.ocs = k \neq k_1$ but $\Theta(u_1).\Theta(ocs) = k_1$. Let $\Theta(u_1.ocs)$ denote the execution result of the method sequence that implements the specified operation sequence $u_1.ocs$. Since the implementation is complete, we have $\Theta(u_1.ocs) = \Theta(u_1).\Theta(ocs)$. Let $\Theta(k_1)$ denote the execution result of k_1. Because k_1 is the value of a primitive type, we have $\Theta(k_1) = k_1$. Based on these relations, we obtain $u_1.ocs \neq k_1$ but $\Theta(u_1.ocs) = \Theta(k_1)$. In other words, $\neg(u_1.ocs \sim_{obs} k_1) \wedge (\Theta(u_1.ocs) \approx_{obs} \Theta(k_1))$. This contradicts the left-hand side of (4.10).

Similarly to the note at the end of the proof of Lemma 1, $\Theta(u_1.ocs) \approx_{obs} \Theta(k_1)$ and $\neg(\Theta(u_1) \approx_{obs} \Theta(u_2))$ are due to the same fault. □

Putting Lemmas 1 and 2 together, we arrive immediately at the following theorem.

> **Theorem 1.** Given a canonical specification of a class with proper imports and a complete implementation, the following two criteria
>
> a. Equivalence Criterion:
> $(\forall u_1)(\forall u_2)((u_1 \sim_{obs} u_2) \rightarrow (\Theta(u_1) \approx_{obs} \Theta(u_2)))$
> b. Nonequivalence Criterion:
> $(\forall v_1)(\forall v_2)(\neg(v_1 \sim_{obs} v_2) \rightarrow \neg(\Theta(v_1) \approx_{obs} \Theta(v_2)))$
>
> imply each other. Hence, the implementation satisfies the specification if and only if we can show that *either* a *or* b is satisfied**.**

We can express Theorem 1 in the form

$$(\forall u_1)(\forall u_2)((u_1 \sim_{obs} u_2) \rightarrow (\Theta(u_1) \approx_{obs} \Theta(u_2))) \Leftrightarrow (\forall v_1)(\forall v_2)(\neg(v_1 \sim_{obs} v_2) \rightarrow \neg(\Theta(v_1) \approx_{obs} \Theta(v_2))). \quad (4.11)$$

Let *OE* denote the set of all pairs of observationally equivalent ground terms of *Sp*, and *OE′* denote the set of all pairs of observational nonequivalent ground terms. It follows from (4.11) that

$$(\forall\{u_1, u_2\} \in OE)(\Theta(u_1) \approx_{obs} \Theta(u_2)) \Leftrightarrow (\forall\{v_1, v_2\} \in OE')(\neg(\Theta(v_1) \approx_{obs} \Theta(v_2))). \quad (4.12)$$

However, this may be too formal for the average software tester. For the sake of brevity, we will define a concept of "*P* satisfies *X*" in the paper as follows:

> **Definition 1 (*P* Satisfies *X*).**
>
> a. Given any set *AE* of pairs of attributively equivalent terms, "*P* satisfies *AE*" means that for any pair of terms u_1 and u_2 in *AE*, their corresponding implementations $\Theta(u_1)$ and $\Theta(u_2)$ are attributively equivalent. That is,
> $(\forall\{u_1, u_2\} \in AE)(\Theta(u_1) \approx_{att} \Theta(u_2))$.
> b. Given any set *X* of pairs of equivalent terms of a specified type other than attributive equivalence, "*P* satisfies *X*" means that for any pair of terms u_1 and u_2 in *X*, their corresponding implementations $\Theta(u_1)$ and $\Theta(u_2)$ are observationally equivalent. That is,
> $(\forall\{u_1, u_2\} \in X)(\Theta(u_1) \approx_{obs} \Theta(u_2))$.
> c. Given any set *AE′* of pairs of attributively nonequivalent terms, "*P* satisfies *AE′*" means that for any pair of terms v_1 and v_2 in *AE′*, their corresponding implementations $\Theta(v_1)$ and $\Theta(v_2)$ are attributively nonequivalent. That is,
> $(\forall\{v_1, v_2\} \in AE')(\neg(\Theta(v_1) \approx_{att} \Theta(v_2)))$.
> d. Given any set *X′* of pairs of nonequivalent terms of a specified type other than attributive nonequivalence, "*P* satisfies *X′*" means that for any pair of terms v_1 and v_2 in *X*, their corresponding implementations $\Theta(v_1)$ and $\Theta(v_2)$ are observationally nonequivalent. That is,
> $(\forall v_1, v_2 \in X')(\neg(\Theta(v_1) \approx_{obs} \Theta(v_2)))$.

Attributive equivalence is treated separately in Definition 1a because it would be too strong to require that u_1 and u_2 are attributively equivalent in the specification but $\Theta(u_1)$ and $\Theta(u_2)$ are observationally equivalent in the implementation. Readers may refer to [10, Theorems 2 and 3] and [10, note (f), p. 78] for more reasoning behind this point. For the sake of uniformity of style, we also treat attributive nonequivalence separately in Definition 1c.

Based on the notation in Definition 1, we can simply write (4.12) as

$$(P \text{ satisfies } OE) \Leftrightarrow (P \text{ satisfies } OE'). \quad (4.13)$$

[4] If O_1 and O_2 are values of a primitive type, then ocs_{impl} is empty.

This is an important result because OE and OE' are intuitively not subsets of each other. In fact, $OE \cap OE' = \emptyset$.

In Section 6, we will further extend (4.13) with other forms of "P satisfies X" proposed by previous related work, such as "P satisfies FP," "P satisfies CI," "P satisfies GI," "P satisfies RP," "P satisfies NE," "P satisfies AE," and "P satisfies AE'."

5 Mutual Replaceability of Equivalence and Nonequivalence Criteria

In this section, we discuss the implications of the new results in the last section to the class-level testing of object-oriented software.

Taking the negations of criteria a and b in Theorem 1, we obtain:

> **Corollary 1.** Given a canonical specification of a class with proper imports and a complete implementation, the following two criteria
>
> a′. Equivalence Failure Criterion: $(\exists u_1)(\exists u_2)((u_1 \sim_{obs} u_2) \wedge \neg(\Theta(u_1) \approx_{obs} \Theta(u_2)))$
>
> b′. Nonequivalence Failure Criterion: $(\exists v_1)(\exists v_2)(\neg(v_1 \sim_{obs} v_2) \wedge (\Theta(v_1) \approx_{obs} \Theta(v_2)))$
>
> imply each other.

> **Corollary 2.** Given any canonical specification of a class with proper imports, suppose its implementation is complete. If a pair of nonequivalent terms $\neg(v_1 \sim_{obs} v_2)$ reveals a failure due to a fault f, then there exists a pair of equivalent terms $(u_1 \sim_{obs} u_2)$ that will also reveal a failure due to the same fault f, and vice versa. In other words, the testing of observationally equivalent ground terms and the testing of observationally nonequivalent ground terms cover each other.

Proof. If a pair of nonequivalent terms $\neg(v_1 \sim_{obs} v_2)$ reveals a failure, according to the nonequivalent failure criterion, we have $\Theta(v_1) \approx_{obs} \Theta(v_2)$. Similarly to the proof of Lemma 1, we have $(v_1.ocs \sim_{obs} k_1)$ and $\neg(\Theta(v_1.ocs) \approx_{obs} \Theta(k_1))$, where, $\neg(\Theta(v_1.ocs) \approx_{obs} \Theta(k_1))$ and $\Theta(u_1) \approx_{obs} \Theta(u_2)$ are due to the same fault f. Thus, the pair of equivalent terms $(v_1.ocs \sim_{obs} k_1)$ also reveals a failure due to the same fault f. The proof of the converse is similar. □

> **Corollary 3.** Given any canonical specification of a class with proper imports, suppose its implementation is complete. Let OE be the set of all observationally equivalent ground pairs and OE' be the set of all observationally nonequivalent ground pairs. Given any finite test suit $OE_s' \subset OE'$, there exists a finite test suit $OE_s \subset OE$ such that for any failure (due to a fault) revealed by a test case in OE_s', a failure (due the same fault) can also be revealed by a test case in OE_s. Conversely, given any finite test suite $OE_t \subset OE$, there exists a finite test suite $OE_t' \subset OE'$ such that for any failure (due to a certain fault) revealed by a test case in OE_t, a failure (due to the same fault) can also be revealed by a test case in OE_t'. Thus, we can replace a finite test suit of observationally nonequivalent ground pairs by a finite test suit of observationally equivalent ground pairs *while revealing failures due to the same faults*, and vice versa.

Proof. For any failure (due to a fault, say, f) revealed by a nonequivalent pair oe' in OE_s', by Corollary 2, there exists an equivalent pair oe that will also reveal a failure due to the same fault f. Let OE_s be the set of all such oe. As OE_s' is finite and every test case in OE_s' may reveal at most one failure, OE_s is also finite and satisfies the post-condition. The proof of the converse is similar. □

In addition to a formal proof, readers may also be interested in how OE_s can actually be constructed. This can be achieved by the following procedure:

Procedure 1. Given any canonical specification of a class with proper imports, suppose its implementation is complete. Suppose further that there is a set F of failures that can be revealed by a finite test suite $OE_s' (\subset OE')$ of observationally nonequivalent ground pairs. The following procedure shows that there exists another finite test suite $OE_s (\subset OE)$ of observationally equivalent ground pairs such that, for every failure in F (due to a certain fault f), there is a test case in OE_s that can also reveal a failure due to the same fault f.

```
Procedure {
Read OE_s';
OE_s = ∅;
For each oe' ∈ OE_s' do {
    /* Each oe' should be of the form ¬(u1 ~obs u2).
       Each oe' reveals a failure in F. */
    /* Under such oe', Θ(u1) ≈obs Θ(u2). */
    There exists an oc sequence ocs on the
       specification such that the output class of the
       observer at the end of the ocs is a primitive
       type and u1.ocs ≠ u2.ocs;
    Suppose u1.ocs = k1 and u2.ocs = k2 such that
       k1 ≠ k2;
    There exists a unique method sequence Θ(ocs)
       that implements ocs;
    If Θ(u1).Θ(ocs) = k, as Θ(u1) ≈obs Θ(u2),
       we should have
       Θ(u2).Θ(ocs) = Θ(u1).Θ(ocs) = k;
    As k1 ≠ k2, we must have k ≠ k1 or k ≠ k2;
    Without loss of generality, suppose k ≠ k1;
    Thus, we have u1.ocs = k1
       but Θ(u1).Θ(ocs) = k ≠ k1;
    Let Θ(u1.ocs) denote the execution result of
       implemented method sequence
       corresponding to u1.ocs;
    We have Θ(u1.ocs) = Θ(u1).Θ(ocs) = k ≠ k1;
    Thus, we obtain u1.ocs = k1
       but Θ(u1.ocs) ≠ Θ(k1) = k1,
       that is, (u1.ocs ~obs k1) ∧ ¬(Θ(u1.ocs) ≈obs Θ(k1));
    Denote the pair of equivalent terms
        (u1.ocs ~obs k1) by oe;
    Set OE_s = OE_s ∪ {oe};
    }
Write OE_s;
}
```

In summary, given a canonical specification of a class with proper imports and a complete implementation, as candidate sets for test case selection, the infinite set of all observationally nonequivalent ground pairs can be replaced by the infinite set of all observationally equivalent ground pairs *while revealing failures due to the same faults*, and vice versa. Furthermore, as test suites, a given finite set of observationally nonequivalent ground pairs can be replaced by a finite set of observationally equivalent ground pairs *while revealing failures due to the same faults*, and vice versa. Of course, we should *not* expect that the infinite set of all observationally nonequivalent (or equivalent, respectively) ground pairs can be replaced by a finite set of observationally equivalent (or nonequivalent, respectively) ground pairs while revealing failures due to the same faults.

6 Theoretical Implications in Relation to Previous Work on Class-Level Testing Based on Algebraic Specifications

In this section, we present the theoretical impacts and contributions of our present work in relation to previous research on object-oriented software testing based on algebraic specifications. We will present the practical implications in the next section.

(a) In previous work such as Aiguier et al. [1], Bernot et al. [3], [4], Le Gall and Arnould [26], Machado [27], [28], and Machado and Sannella [29], the authors defined that a program P is correct with respect to a specification Sp if and only if P satisfies the set of all ground instances of every axiom in Sp. For the sake of brevity, we will use GI to denote this set of all such ground instances. Although the above authors referred to GI as an "exhaustive test set," Gaudel [18] pointed out that "it does not correspond exactly to initial semantics of algebraic specifications since inequalities are not tested: it rather corresponds to loose semantics."

(b) Aiguier et al. [1] proved that, given a canonical[5] specification Sp, if P satisfies the set of all ground instances of every axiom in Sp that contains only creators or constructors (but not transformers), then P satisfies GI. We will use CI to denote this set of all ground instances of every axiom that contains only creators or constructors.[6] In other words, Aiguier et al. proved that

$$(P \text{ satisfies } CI) \Rightarrow (P \text{ satisfies } GI).$$

On the other hand, we note that the ground instances of any axiom may or may not contain only creators or constructors in general. Hence,

$$CI \subset GI, \tag{6.1}$$

where "$\subset$" denotes "is a proper subset of." In other words, $CI \neq GI$. It follows from (6.1) and Definition 1b that

$$(P \text{ satisfies } GI) \Rightarrow (P \text{ satisfies } CI). \tag{6.2}$$

(c) In 1994, Doong and Frankl [16] defined that P is correct with respect to Sp if and only if P satisfies the set of all "equivalent" ground pairs in Sp and the set of all "nonequivalent" ground pairs in Sp, where two ground terms are said to be "equivalent" if and only if the first can be rewritten into the second. Intuitively, the new definition requires that P should not only satisfy every axiom individually, but also satisfy the results from multiple usages of axioms as left-to-right rewrite rules. The former does not necessarily imply the latter, as pointed out by Weyuker's anticomposition and antidecomposition axioms [30].

We will use RP to denote the former set of all such "equivalent" ground pairs[7] and use RP' to denote the latter set of all "nonequivalent" ground pairs. According to Gaudel [18], this "bigger exhaustive test set [$RP \cup RP'$] ... includes for every ground term the inequalities with other normal forms, following the definition of initial semantics." Gaudel and Le Gall [19] indicate that this is "an example of a case where the [bigger] exhaustive test set is not built from instantiations of the axioms, but more generally from an adequate set of semantic consequences of the specification." Zhu [32] further recognizes that "One of [Doong and Frankl's] most important contributions ... is the extension of test cases to include negative test cases, which consists of two terms that are supposed to generate non-equivalent results."

A pair of ground terms u_1 and u_2 is said to be "equivalent" in Doong and Frankl [16] if and only if u_1 can be rewritten to u_2 using one or more axioms as left-to-right rewrite rules. In particular, if only one axiom is used, u_1 and u_2 will be a ground instance of an axiom. Hence, we have

$$GI \subset RP, \tag{6.3}$$

where the symbol "$\subset$" again denotes "is a proper subset of." It follows from (6.3) and Definition 1b that

$$(P \text{ satisfies } RP) \Rightarrow (P \text{ satisfies } GI). \tag{6.4}$$

We will discuss the converse after consolidating all the logic relationships in the proof of (6.16).

(d) In 1998, Chen et al. [9] found that the "equivalence" criterion in Doong and Frankl [16] corresponding to "P satisfies RP" is problematic. To solve the problem, they defined the concepts of normal equivalence and fundamental pairs. Any pair of ground terms u_1 and u_2 are normally equivalent if and only if both of them can be rewritten to the same normal form. Each fundamental pair is formed by replacing all the variables on both sides of an axiom by normal forms. We will use NE to denote the set of normally equivalent pairs, and FP to denote the set of all fundamental pairs. We note the following properties:

[5] The original wording in Aiguier et al. [1] was "under the form of a reductive and confluent rewrite system". Readers in theoretical computer science may recall that a specification in a rewrite system is canonical if and only if it is reductive and confluent.

[6] Other authors such as Gaudel and Le Gall [19] refer to CI as the set of ground instances of every axiom that contains only constructors, because they regard creators as constructors also.

[7] RP is an abbreviation for "*R*ewriting *P*air".

(i) Chen et al. prove in [9, Theorem 2] that, given a canonical specification of a class with proper imports and a complete implementation P,

$$(P \text{ satisfies } NE) \Leftrightarrow (P \text{ satisfies } FP) \quad (6.5)$$

even though FP is only a proper subset of NE.

(ii) Each pair of terms in CI is formed by replacing all the variables on both sides of an axiom by ground terms containing *only creators or constructors*. Since every normal form contains only creators or constructors, we have $FP \subseteq CI$. Conversely, it is proven in [8, Proposition 4] that every ground term containing only creators or constructors may not necessarily be in normal form, and hence we have $FP \neq CI$, and hence

$$FP \subset CI. \quad (6.6)$$

It follows from (6.6) and Definition 1b that

$$(P \text{ satisfies } CI) \Rightarrow (P \text{ satisfies } FP). \quad (6.7)$$

We will also discuss the converse after consolidating all the logic relationships in the proof of (6.16).

(e) In 2001, Chen et al. [10] showed that the "equivalence and not-equivalence" criteria in Doong and Frankl [16] corresponding to "P satisfies RP and P satisfies RP'" are problematic. Given any pair of ground terms u_1 and u_2, if u_1 can be rewritten to u_2, then they are said to be "equivalent" in the sense of RP. However, since u_2 cannot be rewritten to u_1 in many circumstances, they are not "equivalent" in the sense of RP, thus giving contradictory verdicts. More seriously, the terms *new*.*push*(1).*push*(3).*pop* and *new*.*push*(5).*pop*.*push*(1) under the specification in Example 1 produce observationally equivalent objects when the implementation is correct. However, they are *not* "equivalent" in the sense of RP in [16]. Thus, they wrongly report a failure.

As proven in [10, Theorem 1], "equivalence" in the sense of RP implies normal equivalence, but not vice versa. Hence, we have

$$RP \subset NE. \quad (6.8)$$

It follows from (6.8) and Definition 1b that

$$(P \text{ satisfies } NE) \Rightarrow (P \text{ satisfies } RP). \quad (6.9)$$

We will again discuss the converse after consolidating all the logic relationships in the proof of (6.16).

(f) Chen et al. [10] further defined that P is correct with respect to Sp if and only if P satisfies the set (OE) of all observationally equivalent ground pairs in Sp and the set (OE') of all the observationally nonequivalent ground pairs in Sp.

Let AE denote the set of all attributively equivalent ground pairs and AE' denote the set of all attributively nonequivalent ground pairs. Chen et al. proved in [10, Theorems 1, 3, and 4] that

$$NE \subset OE \subset AE \text{ and } AE' \subset OE', \quad (6.10)$$

$$(P \text{ satisfies } AE) \Leftrightarrow (P \text{ satisfies } OE) \Leftrightarrow (P \text{ satisfies } NE), \quad (6.11)$$

$$(P \text{ satisfies } OE') \Leftrightarrow (P \text{ satisfies } AE'). \quad (6.12)$$

(g) In Theorem 1 of this paper, we go one step further to connect (6.11) and (6.12) by proving mathematically that "P satisfies OE" if and only if "P satisfies OE'." Hence, either one of them is a necessary and sufficient condition for verifying the general program correctness criterion of "P satisfies $OE \cup OE'$." In other words, although

$$OE \subset OE \cup OE', \quad (6.13)$$

we have

$$(P \text{ satisfies } OE \cup OE') \Leftrightarrow (P \text{ satisfies } OE) \Leftrightarrow (P \text{ satisfies } OE'). \quad (6.14)$$

Taking the subset relationships (6.1), (6.3), (6.6), (6.8), (6.10), and (6.13) together, for a canonical specification of a class with proper imports and a complete implementation, we have

$$FP \subset CI \subset GI \subset RP \subset NE \subset OE \subset AE \text{ and } OE \subset OE \cup OE'. \quad (6.15)$$

Taking the logic relationships (6.2), (6.4), (6.5), (6.7), (6.9), (6.11), (6.12), and (6.14) together, for a canonical specification of a class with proper imports and a complete implementation, we have

$$\begin{aligned}&(P \text{ satisfies } OE \cup OE')\\ &\Leftrightarrow (P \text{ satisfies } OE) \Leftrightarrow (P \text{ satisfies } OE')\\ &\Leftrightarrow (P \text{ satisfies } AE) \Leftrightarrow (P \text{ satisfies } AE')\\ &\Leftrightarrow (P \text{ satisfies } NE) \Rightarrow (P \text{ satisfies } RP)\\ &\Rightarrow (P \text{ satisfies } GI) \Rightarrow (P \text{ satisfies } CI)\\ &\Rightarrow (P \text{ satisfies } FP) \Leftrightarrow (P \text{ satisfies } NE).\end{aligned}$$

Note that the relationships from "(P satisfies NE)" in line 4 to "(P satisfies NE)" in line 6 of the above statement form a cycle. We conclude that

$$\begin{aligned}&(P \text{ satisfies } OE \cup OE')\\ &\Leftrightarrow (P \text{ satisfies } OE) \Leftrightarrow (P \text{ satisfies } OE')\\ &\Leftrightarrow (P \text{ satisfies } AE) \Leftrightarrow (P \text{ satisfies } AE')\\ &\Leftrightarrow (P \text{ satisfies } NE) \Leftrightarrow (P \text{ satisfies } RP)\\ &\Leftrightarrow (P \text{ satisfies } GI) \Leftrightarrow (P \text{ satisfies } CI)\\ &\Leftrightarrow (P \text{ satisfies } FP). \quad (6.16)\end{aligned}$$

In summary, earlier work defined program correctness via GI, which included only a subset of test cases for verifying equivalent ground terms and did not include any test case for verifying nonequivalent ground terms. Doong and Frankl enhanced the semantics by expanding GI to $RP \cup RP'$, which included subsets of test cases for verifying equivalent and nonequivalent ground terms. Chen et al. further improved the semantics by replacing $RP \cup RP'$ by $OE \cup OE'$. They also proved that, given a canonical specification of a class with proper imports and a complete implementation, the equivalence criterion "P satisfies OE" and the nonequivalence criterion "P satisfies OE'" can be expressed in terms of attributive equivalence and nonequivalence, which can be verified more easily in real-world practice. In this paper, we further prove that the equivalence criterion can be substituted by the nonequivalence criterion and vice versa while revealing failures due to the same faults. In other words, either "P satisfies OE" or "P satisfies OE'" will be necessary and sufficient to confirm that "P satisfies $OE \cup OE'$."

Please refer also to Table 1 for a visual summary. From the table, we see that researchers have gone a long way since the first proposal for program correctness according to "P satisfies GI."

Table 1. Summary of Research on Program Correctness with Respect to Algebraic Specifications

Bernot [3] in 1991	✓							
Bernot et al. [4] in 1991	✓							
Gaudel [18] in 1995	✓							
Le Gall and Arnould [26] in 1996	✓							
Machado [27] in 1998	✓							
Machado [28] in 2000	✓							
Machado and Sannella [29] in 2002	✓							
Aiguier et al. [1] in 2006	✓			✓				
Gaudel and Le Gall [19] in 2008	✓							
Doong and Frankl [16] in 1994		✓						
Chen et al. [9] in 1998					✓			
Chen et al. [10] in 2001			✓			✓		
Chen and Tse in the present paper								✓
Definitions of the Correctness of a Program *P* with Respect to a Specification *Sp*								
Define that *P* is correct with respect to *Sp* if and only if *P* satisfies the set (*GI*) of all ground instances of every axiom in *Sp*.								
Define a pair of "equivalent" ground terms as two ground terms such that one can be rewritten to the other using the axioms in *Sp* as left-to-right rewrite rules. Define that *P* is correct with respect to *Sp* if and only if *P* satisfies the set (*RP*) of all "equivalent" ground pairs in *Sp* and satisfies the set (*RP′*) of all "nonequivalent" ground pairs in *Sp*.								
Point out that the "equivalence" and "nonequivalence" criteria (namely, *P* satisfies *RP* and *P* satisfies *RP′*) in the previous definition are problematic. Define pairs of observationally equivalent ground terms*. Define that *P* is correct with respect to *Sp* if and only if *P* satisfies the set (*OE*) of all observationally equivalent ground pairs in *Sp* and satisfies the set (*OE′*) of all the observationally nonequivalent ground pairs in *Sp*.								
Preconditions for the Correctness of a Program *P* with Respect to a Specification *Sp*								
Prove that, given a canonical specification *Sp*, if *P* satisfies the set (*CI*) of all ground instances of every axiom in *Sp* that contains creators or constructors only, then *P* satisfies *GI*.								
Define two ground terms as (normally) equivalent if and only if both can be rewritten to the same normal form using the axioms in *Sp* as left-to-right rewrite rules. Further define a fundamental pair as two (normally) equivalent ground terms formed by replacing all the variables on both sides of an axiom by normal forms. Prove that, given a canonical specification *Sp* with proper imports and a complete implementation *P*, *P* satisfies the set (*NE*) of all normally equivalent ground pairs in *Sp* if and only if *P* satisfies the set (*FP*) of all fundamentals pairs in *Sp*, even though *FP* is only a proper subset of *NE*.								
Prove that, given a canonical specification *Sp* with proper imports and a complete implementation *P*, *P* satisfies *AE* if and only if *P* satisfies *OE*, *P* satisfies *OE* if and only if *P* satisfies *NE*, and *P* satisfies *AE′* if and only if *P* satisfies *OE′*.								
Prove that, given a canonical specification *Sp* with proper imports and a complete implementation *P*, *P* satisfies *OE* if and only if *P* satisfies *OE′*. Highlight that $$FP \subset CI \subset GI \subset RP \subset NE \subset OE \subset AE, \text{ and } OE \subset OE \cup OE'$$ where the relation "$X \subset Y$" denotes that X is a proper subset of Y, which infers that $X \neq Y$. Prove that, given a canonical specification *Sp* with proper imports and a complete implementation *P*, (*P* satisfies $OE \cup OE'$) $\Leftrightarrow$ (*P* satisfies *OE*) $\Leftrightarrow$ (*P* satisfies *OE′*) $\Leftrightarrow$ (*P* satisfies *AE*) $\Leftrightarrow$ (*P* satisfies *AE′*) $\Leftrightarrow$ (*P* satisfies *NE*) $\Leftrightarrow$ (*P* satisfies *RP*) $\Leftrightarrow$ (*P* satisfies *GI*) $\Leftrightarrow$ (*P* satisfies *CI*) $\Leftrightarrow$ (*P* satisfies *FP*). Based on these relations, emphasize that if *P* satisfies *AE′*, then *P* satisfies all the sets *AE*, *FP*, *CI*, *GI*, *RP*, *NE*, *OE*, *OE′*, and $OE \cup OE'$ in previous work. Then, conclude that it will be most effective to select test cases from the subset *AE′*, which is most easily conducted in real world practice. (See Section 7.2 for the details of this final point.)								

* Previous work defined observational equivalence of objects but did not define observational equivalence of terms.

In general, a program P is correct if and only if it satisfies the requirements of its specification. In the case of an algebraic specification Sp, the requirements include two aspects:

1. the program P must satisfy every axiom in Sp, and
2. the program P must satisfy all the consequences derived from the axioms in Sp.

$OE \cup OE'$ is the set of all such consequences. Hence, the definition that program P is correct if and only if P satisfies GI, defined by early authors, considers aspect 1 only. However, the definition that program P is correct if and only if P satisfies $OE \cup OE'$, defined in our previous work, takes into account not only aspect 1 but also aspect 2. As GI is only a proper subset of $OE \cup OE'$, we have (P satisfies $OE \cup OE'$) $\Rightarrow$ (P satisfies GI) but the converse is not necessarily true. Under what conditions will the converse be valid? Other authors did not investigate this problem, while (6.16) and its proof in our current paper show that, given a canonical specification of a class with proper imports and a complete implementation, the converse will also hold. Under this specific condition, the previous proposal for program correctness according to "P satisfies GI " can theoretically be retained. Similar arguments apply also to FP, CI, RP, NE, AE, and AE'.

Given (6.16), does it mean that any of these criteria require the same amount of testing effort from software testers in real-world practice? The answer is not so simple. We will discuss the practical implications of (6.16) in the next section.

7 Practical Implications in Class-Level Testing Based on Algebraic Specifications

7.1 Practical Implication in Test Case Selection

This section discusses the practical implications due to the new relationship between the testing of observationally equivalent ground pairs and the testing of observationally nonequivalent ground pairs. For the sake of brevity, we will refer to the former as the testing of OE and the latter as the testing of OE'. In general, the relationship between the testing of OE and the testing of OE' may fall into either of the following possibilities:

Possibility 1: The testing of OE and the testing of OE′ does not imply each other. This has been the assumption of previous work such as Doong and Frankl [16], Gaudel [18], Gaudel and Le Gall [19], Chen et al. [10], and Zhu [32]. Under this view, a fault that causes a failure detectable by the testing of OE may or may not cause a failure detectable by the testing of OE', and vice versa. Hence, given any implementation fault, it may

a. cause a failure detectable by the testing of OE but does not cause a failure detectable by the testing of OE', or
b. cause a failure detectable by the testing of OE' but does not cause a failure detectable by the testing of OE, or
c. cause a failure detectable by the testing of OE as well as cause another failure detectable by the testing of OE'.

Thus, test cases have to be selected for verifying a, b, and c. This is similar to the partition testing approach in traditional program testing. Techniques such as proportional sampling strategy [12] may be applied, where the selected numbers of test cases for testing a, b, and c are proportional to their relative input domain sizes. If Possibility 1 is indeed true, a difficulty we need to face is that the relative input domain sizes for a, b, and c are not easy to estimate in order to apply the proportional sampling technique. Fortunately, the main theorem in this paper proves that, given a canonical specification of a class with proper imports and a complete implementation, Possibility 1 is not true.

Possibility 2: The testing of OE and the testing of OE′ imply each other. In this case, a fault that causes a failure detectable by the testing of OE must cause a failure detectable by the testing of OE', and vice versa. As the main contribution of the present paper, Theorem 1 proves that, given a canonical specification of a class with proper imports and a complete implementation, only Possibility 2 is the true scenario. In this scenario, if we select test cases for verifying both OE and OE', it will likely result in redundancy, which lowers the effectiveness and efficiency of testing. Hence, we should select test cases for either OE or OE' but not both.

7.2 Practical Implication in Verifying Observational Equivalence and Nonequivalence of Objects

As we have seen in Section 6, various researchers have proposed different criteria to test the correctness of a program P with respect to a specification Sp, including whether P satisfies $OE \cup OE'$, whether P satisfies OE', whether P satisfies OE, whether P satisfies NE, whether P satisfies RP, whether P satisfies GI, whether P satisfies CI, whether P satisfies FP, whether P satisfies AE, and whether P satisfies AE'. Our analysis in Section 7.1, based on Theorem 1, eliminates the need to verify whether P satisfies $OE \cup OE'$. Our proof of (6.16) in Section 6 also shows that, given a canonical specification of a class with proper imports and a complete implementation, all the above criteria are theoretically equivalent. However, does it mean that any of these criteria require the same amount of testing effort from software testers in real-world practice? This section further analyzes the issue and proposes a practically feasible choice.

Based on Definition 1 in Section 4, the above criteria can be classified into three distinct categories:

1. For criteria of the form "whether P satisfies X," where X is OE, NE, RP, GI, CI, and FP, we need to test whether, for any pair of terms u_1 and u_2 in X, their corresponding implementations $\Theta(u_1)$ and $\Theta(u_2)$ are observationally equivalent. It is practically impossible to verify observational equivalence in real-world software testing because each test case involves an infinite set of potential observable contexts. These criteria are not ideal choices in practice. Although we proposed in [9], [10] that we test whether P satisfies FP, which is the smallest set

among those in (6.15), we needed a heuristic white-box technique ROCS [11] to select a relevant finite subset of the set of observable contexts so as to determine the observational equivalence of objects.

2. For the criterion "whether P satisfies OE'," we need to test whether, for any pair of terms u_1 and u_2 in OE', their corresponding implementations $\Theta(u_1)$ and $\Theta(u_2)$ are observationally nonequivalent. It is practically difficult to find observational nonequivalence in real-world testing because we need to go through possibly an infinite set of observable contexts for each test case. Thus, this criterion is not a practical choice either.
3. For the criteria "whether P satisfies AE" and "whether P satisfies AE'," we need to test whether, for any pair of terms u_1 and u_2 in AE, their corresponding implementations $\Theta(u_1)$ and $\Theta(u_2)$ are attributively equivalent and nonequivalent, respectively. It is simple to verify attributive equivalence and nonequivalence of objects in real-world testing because the set of attributes in any class is finite and usually small. Thus, these two criteria are potential practical choices. Readers may refer to [10, paragraphs after Corollary 2 in Section 4.3] for more analysis.

We have conducted an analysis on the need to test "whether P satisfies AE" or "whether P satisfies AE'" or both, similarly to that in Section 7.1 for OE and OE'. We arrive at a similar conclusion that we should select test cases for AE or AE' but not both. In particular, we recommend testing the criterion "whether P satisfies AE'" because a technique has already been developed for selecting a finite number of test cases from AE'. Our Generating Attributively Nonequivalent terms (GAN) approach in [10, Section 4.4] handles this process using techniques in State-Transition Diagrams (STDs), and turns it into a terminating process via interactive input from users for the maximum numbers of iterations for cyclic paths in STDs. The implementations and experimentation of the GAN approach are also discussed in [10, Section 4]. A limitation of the GAN approach is that it assumes the regularity hypothesis [3], [4], namely, that if a statement has been tested for the positive integers 1, 2, ..., k for some constant k, then it is assumed that the statement will hold for all positive integers. Thus, having tested the cyclic paths for the maximum number of iterations specified by the users, it is assumed that the implementation is correct with any number of iterations.

In summary, based on the practical considerations in Sections 7.1 and 7.2, given a canonical specification of a class with proper imports and a complete implementation in real-world software testing, we recommend selecting test cases from the set AE' and verifying whether the objects that result from executing the corresponding implemented method sequences are attributively nonequivalent, rather than selecting test cases from OE', OE, NE, RP, GI, CI, FP, or AE.

As future work, we will also study the application of ARTOO [13] to select test cases from AE' by defining the object distance of nonequivalent terms with a view to spreading the test cases evenly in the AE'. This will alleviate the users from having to assume the regularity hypothesis and make decisions on the maximum numbers of iterations for cyclic paths.

8 Conclusion

It is generally believed that class-level testing of object-oriented software based on algebraic specification involves two independent aspects: the testing of equivalent ground terms and the testing of nonequivalent ground terms. Previous researchers have cited intuitive examples to illustrate that the latter cannot be replaced by the former, and is therefore of equal importance.

We have proven formally in this paper, however, that given a canonical specification of a class with proper imports and a complete implementation, the equivalence criterion for testing observationally equivalent terms (denoted by "P satisfies OE") and the nonequivalence criterion for testing observationally nonequivalent terms (denoted by "P satisfies OE'") imply each other. Based on this result, we have shown that the testing of observationally equivalent ground terms and the testing of observationally nonequivalent ground terms cover each other. In other words, if a failure due to a certain fault in the implementation that can be revealed by testing observationally nonequivalent ground terms, another failure due to the same fault can also be revealed by testing observationally equivalent ground terms, and vice versa.

We have discussed the theoretical implications of our new findings to related work on class-level testing based on algebraic specifications. We have proven that, given a canonical specification with proper imports and a complete implementation, all the correctness criteria proposed by previous researchers are theoretically equivalent to "P satisfies OE" and to "P satisfies OE'." Under this specific condition, the criteria proposed by previous researchers for program correctness can theoretically be retained.

On the other hand, the need to test either "P satisfies OE" or "P satisfies OE'" is an impossible task in software testing because of the need to verify an infinite number of behavioral outcomes even for one single test case. We have discussed real-world implications, and recommend to conduct testing using a more practical criterion "P satisfies AE'," which is guaranteed by our theoretical results to reveal failures due to the same faults. Even so, we need to assume a regularity hypothesis. As future work, we propose to study the application of ARTOO as an alternative technique to alleviate this assumption.

Acknowledgments

This research was supported by a grant from the National Science Foundation of China (project no. 60173038), grants from the General Research Fund of the Research Grant

Council of Hong Kong (project nos. 717811 and 716612), and a linkage grant from the Australian Research Council (project no. LP100200208). Part of the research was conducted when Huo Yan Chen was in Canada supported by CuiTeck Inc. in Montreal. Part of the research was conducted when T.H. Tse was serving as a distinguished visiting scholar at the Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China.

REFERENCES

[1] M. Aiguier, A. Arnould, C. Boin, P. Le Gall, and B. Marre, "Testing from Algebraic Specifications: Test Data Set Selection by Unfolding Axioms," *Proceedings of the Fifth International Conference on Formal Approaches to Software Testing* (*FATES '05*), pp. 203–217, 2006.

[2] S. Antoy and D. Hamlet, "Automatically Checking an Implementation against Its Formal Specification," *IEEE Transactions on Software Engineering*, vol. 26, no. 1, pp. 55–69, 2000.

[3] G. Bernot, "Testing against Formal Specifications: A Theoretical View," *Proceedings of the Fourth International Joint CAAP/FASE Conference on Theory and Practice of Software Development* (*TAPSOFT '91*), *Part 2: Advances in Distributed Computing and Colloquium on Combining Paradigms for Software Development*, pp. 99–119, 1991.

[4] G. Bernot, M.-C. Gaudel, and B. Marre, "Software Testing Based on Formal Specifications: A Theory and a Tool," *Software Engineering Journal*, vol. 6, no. 6, pp. 387–405, 1991.

[5] M. Bidoit and P.D. Mosses, *CASL User Manual: Introduction to Using the Common Algebraic Specification Language CASL*. Springer, 2004.

[6] P. Borba and J.A. Goguen, "An Operational Semantics for FOOPS," *International Workshop on Information Systems: Correctness and Reusability* (*IS-CORE '94*), 1994.

[7] L. Bougé, N. Choquet, L. Fribourg, and M.-C. Gaudel, "Test Sets Generation from Algebraic Specifications Using Logic Programming," *Journal of Systems and Software*, vol. 6, pp. 343–360, 1986.

[8] H.Y. Chen and T.H. Tse, "Automatic Generation of Normal Forms for Testing Object-Oriented Software," *Proceedings of the Ninth International Conference on Quality Software* (*QSIC '09*), pp. 108–116, 2009.

[9] H.Y. Chen, T.H. Tse, F.T. Chan, and T.Y. Chen, "In Black and White: An Integrated Approach to Class-Level Testing of Object-Oriented Programs," *ACM Transactions on Software Engineering and Methodology*, vol. 7, no. 3, pp. 250–295, 1998.

[10] H.Y. Chen, T.H. Tse, and T.Y. Chen, "TACCLE: A Methodology for Object-Oriented Software Testing at the Class and Cluster Levels," *ACM Transactions on Software Engineering and Methodology*, vol. 10, no. 1, pp. 56–109, 2001.

[11] H.Y. Chen, T.H. Tse, and Y.T. Deng, "ROCS: An Object-Oriented Class-Level Testing System Based on the Relevant Observable Contexts Technique," *Information and Software Technology*, vol. 42, no. 10, pp. 677–686, 2000.

[12] T.Y. Chen, T.H. Tse, and Y.T. Yu, "Proportional Sampling Strategy: A Compendium and Some Insights," *Journal of Systems and Software*, vol. 58, no. 1, pp. 65–81, 2001.

[13] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, "ARTOO: Adaptive Random Testing for Object-Oriented Software," *Proceedings of the 30th International Conference on Software Engineering* (*ICSE '08*), pp. 71–80, 2008.

[14] P. Dauchy, M.-C. Gaudel, and B. Marre, "Using Algebraic Specifications in Software Testing: A Case Study on the Software of an Automatic Subway," *Journal of Systems and Software*, vol. 21, no. 3, pp. 229–244, 1993.

[15] R.-K. Doong, "An Approach to Testing Object-Oriented Programs," PhD thesis, Polytechnic University, Brooklyn, NY, 1993.

[16] R.-K. Doong and P.G. Frankl, "The ASTOOT Approach to Testing Object-Oriented Programs," *ACM Transactions on Software Engineering and Methodology*, vol. 3, no. 2, pp. 101–130, 1994.

[17] J.D. Gannon, P.R. McMullin, and R. Hamlet, "Data Abstraction, Implementation, Specification, and Testing," *ACM Transactions on Programming Languages and Systems*, vol. 3, no. 3, pp. 211–223, 1981.

[18] M.-C. Gaudel, "Testing Can Be Formal, Too," *Proceedings of the Sixth International Joint CAAP/FASE Conference on Theory and Practice of Software Development* (*TAPSOFT '95*), pp. 82–96, 1995.

[19] M.-C. Gaudel and P. Le Gall, "Testing Data Types Implementations from Algebraic Specifications," *Formal Methods and Testing*, R.M. Hierons, J.P. Bowen, and M. Harman, eds., pp. 209–239, Springer, 2008.

[20] J.A. Goguen, C. Kirchner, H. Kirchner, A. Megrelis, and J. Meseguer, "An Introduction to OBJ3," *Proceedings of the First Workshop on Conditional Term Rewriting Systems*, pp. 258–263, 1988.

[21] J.A. Goguen, J.W. Thatcher, and E.G. Wagner, "An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types," *Current Trends in Programming Methodology, Vol. IV: Data Structuring*, R.T. Yeh, ed., pp. 80–149, Prentice Hall, 1978.

[22] J.V. Guttag and J.J. Horning, "The Algebraic Specification of Abstract Data Types," *Acta Informatica*, vol. 10, no. 1, pp. 27–52, 1978.

[23] R.M. Hierons, K. Bogdanov, J.P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Luettgen, A.J.H. Simons, S. Vilkomir, M.R. Woodward, and H. Zedan, "Using Formal Specifications to Support Testing," *ACM Computing Surveys*, vol. 41, no. 2, article no. 9, 2009.

[24] P. Jalote, "Specification and Testing of Abstract Data Types," *Proceedings of the Seventh Annual International Computer Software and Applications Conference* (*COMPSAC '83*), pp. 508–511, 1983.

[25] L. Kong, H. Zhu, and B. Zhou, "Automated Testing EJB Components Based on Algebraic Specifications," *Proceedings of the 31st Annual International Computer Software and Applications Conference* (*COMPSAC '07*), pp. 717–722, 2007.

[26] P. Le Gall and A. Arnould, "Formal Specifications and Test: Correctness and Oracle," *Selected Papers from the 11th Workshop on Specification of Abstract Data Types Joint with the Eighth COMPASS Workshop on Recent Trends in Data Type Specification*, pp. 342–358, 1996.

[27] P.D.L. Machado, "On Oracles for Interpreting Test Results against Algebraic Specifications," *Proceedings of the Seventh International Conference on Algebraic Methodology and Software Technology* (*AMAST '98*), pp. 502–518, 1998.

[28] P.D.L. Machado, "Testing from Structured Algebraic Specifications," *Proceedings of the Ninth International Conference on Algebraic Methodology and Software Technology* (*AMAST '00*), pp. 529–544, 2000.

[29] P.D.L. Machado and D. Sannella, "Unit Testing for CASL Architectural Specifications," *Proceedings of the 27th Symposium on Mathematical Foundations of Computer Science*, pp. 506–518, Springer, 2002.

[30] E.J. Weyuker, "Axiomatizing Software Test Data Adequacy," *IEEE Transactions on Software Engineering*, vol. 12, no. 12, pp. 1128–1138, 1986.

[31] B. Yu, L. Kong, Y. Zhang, and H. Zhu, "Testing Java Components Based on Algebraic Specifications," *Proceedings of the First International Conference on Software Testing, Verification, and Validation* (*ICST '08*), pp. 190–199, 2008.

[32] H. Zhu, "A Note on Test Oracles and Semantics of Algebraic Specifications," *Proceedings of the Third International Conference on Quality Software* (*QSIC '03*), pp. 91–98, 2003.

[33] S.N. Zilles, "Algebraic Specification of Data Types," Project MAC Progress Report 11, MIT, 1974.

Huo Yan Chen received the BS degree in mathematics from Nankai University, China, in 1968 and the master's degree by research in computer science from Jinan University, China, in 1982. He has been a full professor of computer science at Jinan University since 1991. He worked in an IT company as a system analyst and software designer for 10 years. He was a visiting research scholar on computer science at the University of Illinois at Urbana-Champaign from 1986 to 1988. He has conducted research collaborations in software engineering many times at The University of Hong Kong. He was the president of the Guangzhou Intelligent Engineering Association from 2002 to 2006, and a vice-president of the Guangdong Computer Federation from 2004 to 2010. He has received three Science and Technology Awards in software engineering and knowledge engineering from the Ministry of Education and the Guangdong Province in China. He was also recognized by the State Council of China for his outstanding contributions in higher education and science.

T.H. Tse received the PhD degree from the London School of Economics in 1988 and was a visiting fellow at the University of Oxford in 1990 and 1992. He is a professor in computer science at The University of Hong Kong. His current research interest is in program testing, debugging, and analysis. He is the steering committee chair of QSIC and an editorial board member of the *Journal of Systems and Software*, *Software Testing, Verification and Reliability*, *Software: Practice and Experience*, and the *Journal of Universal Computer Science*. He also served on the search committee for the editor-in-chief of the *IEEE Transactions on Software Engineering* in 2013. He is a fellow of the British Computer Society, a fellow of the Institute for the Management of Information Systems, a fellow of the Institute of Mathematics and Its Applications, and a fellow of the Hong Kong Institution of Engineers. He was awarded an MBE by The Queen of the United Kingdom. He is a senior member of the IEEE.