HKU CSIS Tech Report TR-96-07

In Black and White: An Integrated Approach to Class Level Testing of Object-Oriented Programs

HUO YAN CHEN
Jinan University
T.H. TSE, F.T. CHAN
The University of Hong Kong
and
T.Y. CHEN
The University of Melbourne

Because of the growing importance of object-oriented programming, a number of testing strategies have been proposed. They are based either on pure black-box or white-box techniques. We propose in this paper a methodology to integrate the black- and white-box techniques. The black-box technique is used to select test cases. The white-box technique is mainly applied to determine whether two objects resulting from the program execution of a test case are observationally equivalent. It is also used to select test cases in some situations.

We define the concept of a fundamental pair as a pair of equivalent terms that 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 as good as that of all possible term rewritings, and hence we need only concentrate on the testing of fundamental pairs. Our strategy is based on mathematical theorems. According to the strategy, we propose an algorithm for selecting a finite set of fundamental pairs as test cases.

Given a pair of equivalent terms as a test case, we should then determine whether the objects that result from executing the implemented program are observationally equivalent. We prove, however, that the observational equivalence of objects cannot be determined using a finite set of observable contexts (which are operation sequences ending with an observer function) derived from any black-box technique. Hence, we supplement our approach with a "relevant observable context" technique, which is a heuristic white-box technique to select a relevant finite subset of the set of observable contexts for determining the observational equivalence. The relevant observable contexts are constructed from a Data member Relevance Graph, which is an abstraction of the given implementation for a given specification. A semi-automatic tool has been developed to support this technique.

Categories and Subject Descriptors: D.2.1 **[Software Engineering]**: Requirements/Specifications — *languages*; D.2.5 [**Software Engineering**]: Testing and Debugging — *test data generators*; D.3.2 [**Programming Languages**]: Language Classifications — *object-oriented languages*

General Terms: Algorithms, Languages, Reliability

Additional Key Words and Phrases: Abstract data types, algebraic specification, object-oriented programming, software testing methodologies, observational equivalence

This research is supported in part by a grant of the Hong Kong Research Grants Council, a grant of the Guangdong Province Science Foundation (#950618), and a grant of the Australian Research Council.

Authors' addresses: Huo Yan Chen, Department of Computer Science, Jinan University, Guangzhou 510632, China. Email: "tchy@maina.jnu.edu.cn" and "hychen@cs.hku.hk". T.H. Tse **(Contact Author)**, Department of Computer Science, The University of Hong Kong, Pokfulam Road, Hong Kong. Email: "thtse@cs.hku.hk". (Part of this research was done when the second author was an exchange visitor at the University of Melbourne.) F.T. Chan, School of Professional and Continuing Education, The University of Hong Kong, Pokfulam Road, Hong Kong. Email: "hrxecft@hkucc.hku.hk". T.Y. Chen, Department of Computer Science, The University of Melbourne, Parkville 3052, Australia. Email: "tyc@cs.mu.oz.au".

1. INTRODUCTION

The special characteristics and properties of an object-oriented approach render resulting software systems more reliable, maintainable, and reusable. However, an object-oriented approach also poses new challenges to software testing as a software system is now composed of classes of objects and has unique features not found in other programming paradigms. New testing problems arise from the following facts: (1) Programs in an object-oriented system are not necessarily executed in a predefined order; the sequence of invocation of methods in a class is not specified explicitly; and there are more variations in combining methods in the same class or across different classes [1] . (2) Special testing techniques are also required to deal with inheritance, polymorphism, overloading, message passing, association, aggregation, and state-dependent behavior [2, 3, 4, 5, 6, 7]. (3) Furthermore, it is mandatory to derive an algorithm for determining the observational equivalence of the output objects so as to judge the correctness of implementations. The concept of object observational equivalence reflects the encapsulation and information hiding features of the object-oriented paradigm. In this paper, we only consider the facts (1) and (3) in *class level testing*, which concerns only the interactions of methods and data within a given class. However, inheritance, polymorphism, overloading, message passing, association, and aggregation concern the relationships and interactions among different classes in a given **cluster**, which are considered in our other paper [8]

In recent years, a number of papers on class-level testing of object-oriented programs have been published. The techniques involved can be classified into two categories. The black-box technique refers to program testing based on software specifications [9, 10, 11]; whereas the white-box technique refers to that based on information from the source code of the developed systems [1, 12, 13, 14, 15, 16, 17]. Each technique has its advantages and disadvantages. For example, if part of the specification is missing in an implementation, there is no way of revealing the problem using a pure white-box technique. On the other hand, we shall formally prove that it is impossible to determine whether two objects are observationally equivalent using a pure black-box technique. We therefore propose to integrate black- and white-box techniques in our project. We do not consider program syntax errors and specification errors in this paper.

The organization of the paper is as follows. Section 2 states the problems of test case selection, including the reasons why we use equivalent sequences of operations rather than individual operations as test cases, and our new selection strategy. In Section 3, we present a white-box technique, namely a "relevant observable context" technique, to determine the observational equivalence of objects. Section 4 is devoted to comparing our approaches with related work by other researchers. In Section 5, we conclude our current findings, summarize their limitations, and make suggestions for future work.

2. SELECTION OF TEST CASES

2.1 Background: Equivalent Terms as Test Cases

Algebraic specifications are popular in the formal specification of object-oriented programs [18, 19, 20]. An algebraic specification for a class consists of a syntax declaration and a semantic specification. The syntax declaration lists the *operations*[1] involved, plus their domains and co-domains, corresponding to the input and output parameters of the operations. The semantic specification consists of equational

[1] In this paper, the word "operation" is used in a specification, while its counterpart in the implementation is called a "method".

axioms that describe the behavioral properties of the operations. The following is an example of an algebraic specification for the class of integer stacks.

Example 1

module *INTSTACK is*
classes *Int Bool IntStack*
inheriting *INT*
operations
new: → *IntStack*
_.empty: *IntStack* → *Bool*
_.push(*_*): *IntStack Int* → *IntStack*
_.pop: *IntStack* → *IntStack*
_.top: *IntStack* → *Int* ∪ {*NIL*}
variables
S: *IntStack*
N: *Int*
axioms
a_1: *new.empty = true*
a_2: *S.push*(*N*).*empty = false*
a_3: *new.pop = new*
a_4: *S.push*(*N*).*pop = S*
a_5: *S.top = NIL if S.empty*
a_6: *S.push*(*N*).*top = N* ■

Intuitively, a *term* is a series of operations in an algebraic specification. For example, *new.push*(1).*push*(2).*pop* is a term in the class of integer stacks above. A term is in *normal form* if and only if it cannot be further transformed by any axiom in the specification. For example, *new.push*(1).*push*(2) is in normal form but *new.push*(1).*push*(2).*pop* is not.

The concept of equivalent terms has been adopted in testing [9, 11, 21, 22, 23]. Two terms are said to be *equivalent* if and only if they can both be transformed to the same normal form. The terms *new.push*(1).*push*(2).*pop* and *new.push*(3).*pop.push*(1) are equivalent as they can both be transformed to the normal form *new.push*(1). A term without variables is called a *ground term*. In this paper, we only consider ground terms because in dynamic testing, actual test cases involve ground terms only.

Let u_1 and u_2 be two ground terms and s_1 and s_2 be their corresponding method sequences in a given implementation. The test case $\{u_1, u_2\}$ reveals an error of the implementation if u_1 is equivalent to u_2 but s_1 and s_2 produce observationally different objects.

The idea of using pairs of equivalent terms, rather than individual operations, as test cases in object-oriented black-box testing is justified by the following reasons:

(1) In object-oriented programming, a series of messages are often passed to an object, and the resulting object is then evaluated for correctness. The concept of observational equivalence is very important here. Consider, for example, a word processor that maintains the history of insertions and deletions in its document file for the purpose of undo's and redo's before it is finally saved. A user may enter a series of messages into the word processor, possibly with a number of wrong insertions followed

by a number of corrective deletions. Another user may make different mistakes followed by different corrections when creating the same document. In either case, as long as they produce the same printed version, the final document files produced by the two users should be regarded as observationally equivalent. The concept of "equivalent terms" models this phenomenon very naturally. A series of messages passed to the object is modeled by a sequence of operations in a term. The objects resulting from two different series of messages would be equivalent if their observable versions, modeled by normal forms, are identical.

(2) The conventional approach of testing the output B of an individual operation _.*op* using an input A is just a special case of the testing of equivalence. The equivalent terms in this case are A.*op* and B. However, testing pairs of equivalent terms includes the checking of interactions among operations in the terms, while testing individual operations separately does not.

Obviously, if an error occurs in a common subterm of a pair of equivalent terms, it cannot be revealed with this pair as a test case. We can, however, find another pair of equivalent terms to reveal this error. For example, if u_1 = *new*.*push*(1).*push*(2).*pop* and u_2 = *new*.*push*(1), then u_1 and u_2 are equivalent. The common subterm of u_1 and u_2 is *new*.*push*(1). If _.*push*(1) is erroneously implemented as _.*push*(11), then the error cannot be detected by the test case of equivalent terms u_1 and u_2, but can be revealed by another pair of equivalent terms *new*.*push*(1).*top* and 1.

2.2 Basic Concepts

The following are the formal definitions of the basic concepts used in this paper. Definitions 2.1, 2.2, 2.3, and 2.6 are about algebraic specification, Definitions 2.7 to 2.10 are about implementation, while Definitions 2.4 and 2.5 are related to both.

Definition 2.1 The sets T_C of *ground terms* in a term algebra T are defined recursively as follows:

(a) For any constant or *constant operation* f: $\rightarrow C$, f is a ground term in T_C. The *length* of f is defined to be 1.

(b) For any operation _.f(_, ..., _): $C_0\ C_1 \ldots C_n \rightarrow C$ (where $n \geq 0$), and for any ground terms u_i in T_{Ci}, $0 \leq i \leq n$, $u_0.f(u_1, ..., u_n)$ is a ground term in T_C. Each u_i is a *proper subterm* of $u_0.f(u_1, ..., u_n)$. Furthermore, if v is a proper subterm of u_i, then v is also a *proper subterm* of $u_0.f(u_1, ..., u_n)$. The *length* of $u_0.f(u_1, ..., u_n)$ is defined as

$$length(u_0.f(u_1, ..., u_n)) = 1 + length(u_0) + length(u_1) + ... + length(u_n)$$

In Example 1, for instance, *new*.*push*(1).*push*(2).*pop*.*top* is a ground term, with proper subterms "*new*.*push*(1).*push*(2).*pop*", "*new*.*push*(1).*push*(2)", "*new*.*push*(1)", "2", "*new*", and "1". Their lengths are shown in Table 1. By Definition 2.1, all ground terms are of finite lengths.

Table 1 Lengths of terms

Terms	Lengths
"1"	1
new	1
"2"	1
new.*push*(1)	3
new.*push*(1).*push*(2)	5
new.*push*(1).*push*(2).*pop*	6
new.*push*(1).*push*(2).*pop*.*top*	7

Definition 2.2 Suppose

(a) a_0: $u = u$' is an equational axiom such that each variable occurring in u' also appears in u.
(b) u_0 is a ground term containing a subterm that is a substitution instance of the left-hand side u of the axiom.
(c) if we replace that subterm in u_0 by the corresponding substitution instance of the right-hand side u', the result is a ground term u_1.

Then we say that the ground term u_0 can *be transformed* into the ground term u_1 using the axiom a_0 as a *left-to-right rewriting rule*. This is denoted by the notation:

$$u_0 \xrightarrow{a_0} u_1.$$

In Example 1, for instance, the ground term *new*.*push*(1).*push*(2).*pop*.*top* can be transformed into *new*.*push*(1).*top* using axiom a_4 as a left-to-right rewriting rule, and *new*.*push*(1).*top* can be transformed further into the ground term "1" using a_6. These transformations are expressed as:

$$new.push(1).push(2).pop.top \xrightarrow{a_4} new.push(1).top \xrightarrow{a_6} 1.$$

Definition 2.3 A ground term is said to be in *normal form* if and only if no further axiom is applicable to it as a left-to-right rewriting rule. A specification is said to be *canonical* if and only if every sequence of rewrites on the same ground term reaches a unique normal form in a finite number of steps.

Definition 2.4 Suppose A_1, ..., A_i, and E are classes different from class C, and D_1, ..., D_k are classes that may or may not be different from class C, where $i, k \geq 0$. An operation or method f: $A_1 \ldots A_i \rightarrow C$ is called a *creator* of class C. An operation or method _.g(_, ..., _): $C\ D_1 \ldots D_k \rightarrow C$ is called a *constructor* of class C if it can appear in a normal form. If an operation or method _.h(_, ..., _): $C\ D_1 \ldots D_k \rightarrow C$ cannot appear in any normal form, it is called a *transformer* of class C. An operation or method _.p(_, ..., _): $C\ D_1 \ldots D_k \rightarrow E$ is called an *observer* of class C.

In Example 1, for instance, the operation *new* is a creator, _.*push*(N) is a constructor, _.*pop* is a transformer, and _.*empty* and _.*top* are observers.

Definition 2.5 Suppose C is a class of a given specification. A valid sequence of operations or methods in C, starting from a constructor or transformer but ending in an observer, is called an *observable context* on C.

The general form of an observable context *oc* is as follows:

$$oc = \_.f_1(...).f_2(...)..f_k(...).obs(...)$$

where $\_.f_1(...)$, $\_.f_2(...)$, ..., $\_.f_k(...)$ are constructors or transformers of class *C* and $\_.obs(...)$ is an observer of class *C*. For any object *O* in *C*, *O.oc* denotes the result of applying *oc* to *O*. For example, $oc = \_.push(1).push(2).pop.push(4).top$ is an observable context on the class of integer stacks given by Example 1. The result of applying it to *new* is $new.push(1).push(2).pop.push(4).top = 4$.

Definition 2.6 For a given canonical specification, two ground 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 into the same normal form by some axioms as left-to-right rewriting rules.

The following definition is adapted from [11, 22]:

Definition 2.7 Given a canonical specification and its implementation in a class *C*, two objects O_1 and O_2 in *C* are said to be *observationally equivalent* (denoted by $O_1 \approx O_2$) if and only if the following conditions are satisfied:

(a) When *C* is a class provided by the implementation language, O_1 and O_2 are identical according to the built-in equality in the language.

(b) When *C* is a user-defined class, for any observable context *oc* on *C*, $O_1.oc$ is observationally equivalent to $O_2.oc$ in the output class.

Let $O_i.d_j$ represent the value of a data member d_j in an object O_i. Notice that even if $O_1.d_j \neq O_2.d_j$ for some data member d_j of the objects O_1 and O_2, it does not necessarily mean that O_1 and O_2 are observationally nonequivalent. This point reflects the encapsulation or hiding of implementation details in the object-oriented paradigm.

For a canonical system, the observational equivalence of objects is reflexive, symmetric, and transitive.

Definition 2.8 Given a canonical specification and its implementation, a series of methods corresponding to the operations in a ground term is called a *method sequence* corresponding to the ground term. Two such sequences s_1 and s_2 are said to be *equivalent* (denoted by $s_1 \approx s_2$) if and only if they produce observationally equivalent objects.

For a canonical system, the equivalence of method sequences is reflexive, symmetric, and transitive.

Definition 2.9 Suppose *P* is an implementation of a canonical specification *SP*. *P* is said to be *complete* if and only if, for every operation *f* in *SP*, there exists one and only one method m_f in *P* that implements *f*.

We can regard a complete implementation as a mapping Ψ from the specification to the implemented class, such that $\Psi(f) = m_f$. Let $u = f_1.f_2...f_k$ be a ground term in the class. We write the method sequence $\Psi(f_1).\Psi(f_2)...\Psi(f_k)$ as $\Psi(u)$.

Definition 2.10 A complete implementation Ψ is said to be *consistent with respect to the equivalent ground terms* u_1 and u_2 if and only if the corresponding method sequences $\Psi(u_1) \approx \Psi(u_2)$.

Obviously, given a canonical specification, if a complete implementation is not consistent with respect to some equivalent ground terms, then there is an error in this implementation. Hence, this forms the basis of using equivalent ground terms as test cases.

2.3 Our Strategy: Fundamental Pairs as Test Cases

Although we have seen the rationale behind the use of equivalent ground terms as test cases, the set of all such terms for a given specification is infinite in general. Exhaustive testing is of course impossible. How do we select a finite representative subset of all equivalent ground terms as test cases? In this and the next sections, we shall propose a mathematically based strategy for selecting representative equivalent ground terms as test cases. First, we define an important concept as follows.

Definition 2.11 For a given canonical specification, a pair of equivalent ground terms, formed by replacing all the variables on both sides of an axiom by normal forms, is called a *fundamental pair of equivalent terms* induced from the axiom. For the simplicity of expression, we shall refer to such a pair as a *fundamental pair* in this paper.

In Example 1, the pair of equivalent ground terms *new*.*push*(1).*push*(2).*push*(3).*pop* ~ *new*.*push*(1).*push*(2) can be formed by replacing the variables *S* and *N* in axiom a_4 by the normal forms *new*.*push*(1).*push*(2) and "3", respectively, and hence is a fundamental pair induced from axiom a_4. However, the pair of equivalent ground terms *new*.*push*(1).*push*(2).*pop*.*push*(3).*pop* ~ *new*.*push*(1) is not fundamental.

When we generate fundamental pairs from an axiom, if the right side of the axiom contains some conditions, the selected normal forms to replace variables have to satisfy these conditions.

Having defined the basic concepts, we would like to state our strategy for test case selection. We prove that, in order to test whether a complete implementation of a canonical specification is consistent with respect to all equivalent ground terms, we need only test fundamental pairs. That is, the testing coverage of all fundamental pairs remains the same as that of all equivalent ground terms. In other words, any error revealable by general equivalent ground terms can be revealed by some fundamental pairs.

Example 2 Given the specification as shown in Example 1, consider an erroneous implementation in which the second call of *pop* returns a wrong value because of a *flag*. In this implementation, a stack is represented by an *array* and has an internal Boolean *flag* that is set to *false* when a new stack is created. The operation *pop* is implemented as follows[2]:

```
S.pop:
    if S.flag then return NIL;     /* An error */
    else { ...     /* If S.flag is false, then pop behaves properly. The code for the correct
                      implementation will not be listed in full here. */
         S.flag = true;   }
```

[2] This example is due to [24].

This error can be revealed by the following pair of general equivalent ground terms:

$w_1 = new.push(1).push(2).pop.push(3).pop$,
$w_2 = new.push(1)$;

It must also be exposed by a fundamental pair, say,

$u_1 = new.push(1).push(2).push(3).pop$,
$u_2 = new.push(1).push(2)$.

Executing the corresponding implementation method sequences s_1 and s_2 of u_1 and u_2, respectively, we obtain the following objects as a result:

array flag /* Two data members in the implementation */
O_1 = ([1, 2], *true*)
O_2 = ([1, 2], *false*)

Since $O_1.pop.top = NIL$ but $O_2.pop.top = 1$, by Definition 2.7, O_1 and O_2 are not observationally equivalent. Thus, $\neg(s_1 \approx s_2)$, but $u_1 \sim u_2$. By Definition 2.10, the error is detected by $u_1 \sim u_2$. ■

In Example 1, for instance, we need only select test cases like the fundamental pair $new.push(1).push(2).push(3).pop \sim new.push(1).push(2)$, and need not consider general equivalent ground terms like $new.push(1).push(2).pop.push(3).pop \sim new.push(1)$. Obviously, the set of fundamental pairs is a proper subset of the set of equivalent ground terms.

To prove Theorem 2, we need the following lemmas and Theorem 1.

Lemma 1 If a ground term u_i is a proper subterm of ground term u, then $length(u) > length(u_i) > 0$.

Proof:

The proof of Lemma 1 follows immediately from Definition 2.1. ■

Lemma 2 A canonical specification cannot contain any axiom a in the following form:

a: $X = T$

where X is a variable, T is a term, and X and T belong to the same class.

Proof:

Suppose the given specification contains a: $X = T$. Let u_0 be a term of the class.

If T contains the variable X, we denote $T = T(X)$,

a: $X = T(X)$.

Thus, we have:

$$u_0 \xrightarrow{a} T(u_0) \xrightarrow{a} T(T(u_0)) \xrightarrow{a} \ldots$$

which is an infinite rewriting sequence, thus contradicting the termination property of canonical specifications.

If T does not contain the variable X, we have:

$$u_0 \xrightarrow{a} T \xrightarrow{a} T \xrightarrow{a} \ldots$$

which is also an infinite rewriting sequence, thus contradicting the termination property of canonical specifications. ■

For instance, if Example 1 contained an axiom a_{01}: $S = S.push(N).pop$, then we would obtain the infinite rewriting sequence:

$$new \xrightarrow{a_{01}} new.push(1).pop \xrightarrow{a_{01}} new.push(1).pop.push(2).pop \xrightarrow{a_{01}} \ldots$$

which would contradict the terminating property of canonical specifications. If it contained an axiom a_{02}: $S = new$, we would have an infinite rewriting sequence:

$$new \xrightarrow{a_{02}} new \xrightarrow{a_{02}} new \xrightarrow{a_{02}} \ldots$$

which would also contradict the terminating property of canonical specifications.

Lemma 3 Suppose u is a ground term in a canonical specification. If u is not in normal form, then there exists an axiom a which can be applied to u as a left-to-right rewriting rule such that the following *binding condition* is satisfied:

All the variables involved in a are bound to normal forms.

Proof:

The basic idea of the proof is shown in Figure 1. Since u is not in normal form, according to Definition 2.3, there exists some axiom a_1 that can be applied to u as a left-to-right rewriting rule. If a_1 satisfies the binding condition, then the conclusion holds.

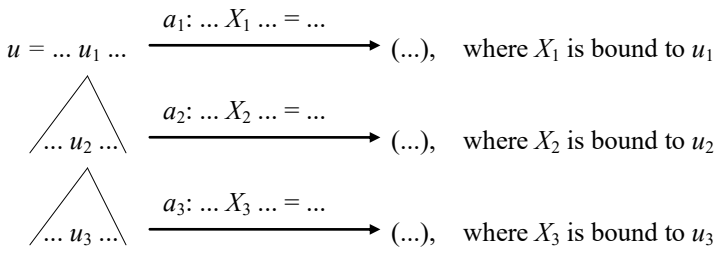


Figure 1. The basic idea of the proof of Lemma 3

Otherwise, a_1 includes a variable X_1 bound to a non-normal form u_1, where u_1 is a subterm of u. We know that u_1 must be a proper subterm of u because, if $u_1 = u$, a_1 would be of the form:

$$a_1: X_1 = T.$$

thus contradicting Lemma 2. According to Definition 2.3 again, there exists some other axiom a_2 that can be applied to u_1 as a left-to-right rewriting rule. Since u_1 is a proper subterm of u, a_2 can also be applied to u as a left-to-right rewriting rule. If a_2 satisfies the binding condition, then the conclusion holds.

Otherwise, a_2 includes a variable X_2 bound to a non-normal form u_2, where u_2 is a subterm of u_1. Similarly to the above, according to Lemma 2, u_2 must be a proper subterm of u_1. Furthermore, u_2 is also a proper subterm of u. According to Definition 2.3, there exists some other axiom a_3 that can be applied to u_2 as a left-to-right rewriting rule. Since u_2 is also a proper subterm of u, a_3 can also be applied to u as a left-to-right rewriting rule. If a_3 satisfies the binding condition, then the conclusion holds.

Otherwise, continue the process similar to the above. Since the length of u is finite, and according to Lemma 1, we have

$$length(u) > length(u_1) > length(u_2) > ... > 0,$$

the process must terminate in a finite number of steps. Thus, we obtain an axiom a_i that can be applied to u as a left-to-right rewriting rule and satisfies the binding condition. ■

Consider, for instance, a term u = *new*.*push*(1).*push*(2).*pop*.*push*(3).*top* for the specification in Example 1. We can apply axiom a_6 to u as a left-to-right rewriting rule. In this case, the variables S and N in a_6 are bound to *new*.*push*(1).*push*(2).*pop* and 3, respectively. However, *new*.*push*(1).*push*(2).*pop* is not in normal form. We can apply axiom a_4 to it, such that the variables S and N involved are bound to the normal forms "*new*.*push*(1)" and "2", respectively. Furthermore, since "*new*.*push*(1).*push*(2).*pop*" is a subterm of u, we can also apply a_4 directly to u. Thus, we have found an axiom a_4 which can be applied directly to u as a left-to-right rewriting rule such that the binding condition is satisfied.

Theorem 1 Suppose u is a ground term in a canonical specification. If u is not in normal form, then u can be transformed into a unique normal form u* via a series of axioms a_1, a_2, ..., a_k:

$$u \xrightarrow{a_1} u_2 \xrightarrow{a_2} \ldots \xrightarrow{a_{k-1}} u_k \xrightarrow{a_k} u_{k+1} = u^*,$$

such that each a_i satisfies the binding condition.

Proof:

Since the given specification is canonical, according to Lemma 3, there exists some axiom a_1 that can be applied to u as a left-to-right rewriting rule, transforming u into a ground term u_2:

$$u \xrightarrow{a_1} u_2,$$

and satisfies the binding condition. If u_2 is in normal form, then the theorem holds.

Otherwise, according to the same lemma, there exists another axiom a_2:

$$u_2 \xrightarrow{a_2} u_3,$$

which satisfies the binding condition. If u_3 is in normal form, then the theorem is satisfied.

Otherwise, continue the process similar to the above. Since the specification is canonical, according to Definition 2.3, the process must terminate at a unique normal form, thus yielding a finite series of axioms $a_1, a_2, \ldots, a_k$ that satisfies the theorem. ■

Theorem 2 Given a canonical specification, a complete implementation is consistent with respect to all equivalent ground terms if and only if it is consistent with respect to all fundamental pairs.[3]

Proof:

Obviously, if a complete implementation is consistent with respect to all equivalent ground terms, then it is consistent with respect to all fundamental pairs.

Suppose a complete implementation is consistent with respect to all fundamental pairs. Let $u_1 \sim u_2$ be any two equivalent ground terms. Since the implementation is complete, it can be regarded as a mapping Ψ. Let $\Psi(u_1) = s_1$ and $\Psi(u_2) = s_2$. We wish to prove that $s_1 \approx s_2$.

By Definition 2.6, u_1 and u_2 can be transformed into the same normal form u^*. Since the given specification is canonical, according to Theorem 1, we can find a series of k axioms that transform u_1 to u^*:

$$u_1 \xrightarrow{a_1} u_{12} \xrightarrow{a_2} u_{13} \xrightarrow{a_3} \ldots \xrightarrow{a_{k-1}} u_{1k} \xrightarrow{a_k} u^*.$$

[3] In spite of this theorem, we should note that an infinite set of observable contexts may be required to check the observational equivalence of objects resulting from the fundamental pairs. This problem cannot be solved by any black-box technique. We shall present in Section 3 a heuristic white-box technique that selects a relevant finite subset of the set of observable contexts.

where all the axioms a_1, ..., a_k satisfy the binding condition. Let $\Psi(u_{1j}) = s_{1j}, j = 2, ..., k$, and $\Psi(u^*) = s^*$.

u_1 must be of the form $f_0(v_0).f_1(v_1)..f_n(v_n)$. Each v_i, $i = 0, 1, ..., n$, is a list of parameters (possibly an empty list) of the form

$$v_i = (h_{10}(v_{10}).h_{11}(v_{11})...h_{1p}(v_{1p}), h_{20}(v_{20}).h_{21}(v_{21})...h_{2q}(v_{2q}), ..., h_{r0}(v_{r0}).h_{r1}(v_{r1})...h_{rs}(v_{rs}))_i ,$$

where each parameter contains only ground terms, each v_{10}, v_{11}, ..., v_{rs} may further be expressed in a form similar to v_i, and so on.

Since u_1 can be rewritten by applying a_1 as a rewriting rule, the left hand side of a_1 should match a subterm of u_1. Hence, a_1 must be of one of the following two forms:

$$X.f_j(X_j).f_{j+1}(X_{j+1})..f_k(X_k) = X.g_1(Y_1).g_2(Y_2)...g_m(Y_m), \qquad 1 \le j \le k \le n,$$

or

$$X.f_j(X_j).f_{j+1}(X_{j+1})\ldots f_k(X_k) = Y.g_1(Y_1).g_2(Y_2)\ldots g_m(Y_m), \qquad 1 \le j \le k \le n,$$

where X and Y ($\neq X$) are creators or object variables[4], X_j, X_{j+1}, ..., X_k, Y_1, Y_2,..., Y_m are lists of parameters (possibly empty lists) containing variables or ground terms, and $g_1(Y_1).g_2(Y_2)...g_m(Y_m)$ may be absent[5].

Thus, u_{12} will be of the corresponding form

$$f_0(v_0).f_1(v_1)..f_{j-1}(v_{j-1}).g_1(w_1).g_2(w_2)...g_m(w_m).f_{k+1}(v_{k+1})...f_n(v_n)$$

or

$$w.g_1(w_1).g_2(w_2)...g_m(w_m).f_{k+1}(v_{k+1})...f_n(v_n),$$

where w, w_1, w_2, ..., w_m are substitution instances of Y, Y_1, Y_2, ..., Y_m, respectively. Without loss of generality, we will only discuss the more complex case

$$u_{12} = f_0(v_0).f_1(v_1)..f_{j-1}(v_{j-1}).g_1(w_1).g_2(w_2)...g_m(w_m).f_{k+1}(v_{k+1})...f_n(v_n)$$

for the remaining part of this proof. Thus,

$$s_1 = \Psi(f_0(v_0)).\Psi(f_1(v_1))...\Psi(f_n(v_n)),$$

---

[4] Since a_1 satisfies the binding condition and X in a_1 is bound to $f_0(v_0).f_1(v_1)..f_{j \cdot 1}(v_{j \cdot 1})$, we need to show that any ground term u_1 contains at least a normal form at the beginning. In fact, any ground term consists of a creator at its beginning followed by constructors, transformers, or observers. According to Lemma 2, a canonical specification cannot contain any axiom of the form "$X = T$". Thus, any creator cannot be rewritten in a canonical specification, and hence is a normal form. Besides the creator, in the ground term u_1, any subterm beginning with this creator and followed only by some constructors is also a normal form. See footnote 6. We can therefore conclude that any ground term u_1 contains at least some normal form at its beginning.

[5] For example, the axiom *X.credit*(*M*).*balance* = *X.balance.add*(*M*) is of the first form such that $g_1(Y_1).g_2(Y_2)...g_m(Y_m)$ is present. The axiom *X.push*(*N*).*pop* = *X* is also of the first form such that $g_1(Y_1).g_2(Y_2)...g_m(Y_m)$ is absent. The axiom *X.push*(*N*).*top* = *N* is of the second form such that $Y = N$ and $g_1(Y_1).g_2(Y_2)...g_m(Y_m)$ is absent. The axiom *X.push*(*N*).*empty* = *false* is also of the second form such that Y is the creator *false* and $g_1(Y_1).g_2(Y_2)...g_m(Y_m)$ is absent.

$$s_{12} = \Psi(f_0(v_0)).\Psi(f_1(v_1))...\Psi(f_{j-1}(v_{j-1})).\Psi(g_1(w_1)).\Psi(g_2(w_2))...\Psi(g_m(w_m))$$
$$.\Psi(f_{k+1}(v_{k+1}))...\Psi(f_n(v_n)).$$

Since axiom a_1 satisfies the binding condition in the transformation from u_1 to u_{12},

$$f_0(v_0).f_1(v_1)\ldots f_{j-1}(v_{j-1}).f_j(v_j).f_{j+1}(v_{j+1})\ldots f_k(v_k) \sim f_0(v_0).f_1(v_1)\ldots f_{j-1}(v_{j-1}).g_1(w_1).g_2(w_2)\ldots g_m(w_m)$$

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

$$\Psi(f_0(v_0)).\Psi(f_1(v_1))\ldots\Psi(f_k(v_k))$$
$$\approx \Psi(f_0(v_0)).\Psi(f_1(v_1))\ldots\Psi(f_{j-1}(v_{j-1})).\Psi(g_1(w_1)).\Psi(g_2(w_2))...\Psi(g_m(w_m)) \quad \text{(a)}$$

If $f_n(v_n)$ is an observer, then

$$\Psi(f_{k+1}(v_{k+1}))...\Psi(f_n(v_n)) \quad \text{(b)}$$

is an observable context. According to Definitions 2.7 and 2.8, by applying (b) to both sides of (a), we have

$$\Psi(f_0(v_0)).\Psi(f_1(v_1))...\Psi(f_n(v_n))$$
$$\approx \Psi(f_0(v_0)).\Psi(f_1(v_1))...\Psi(f_{j-1}(v_{j-1})).\Psi(g_1(w_1)).\Psi(g_2(w_2))...\Psi(g_m(w_m))$$
$$.\Psi(f_{k+1}(v_{k+1}))...\Psi(f_n(v_n))$$

In other words, $s_1 \approx s_{12}$.

If $f_n(v_n)$ is not an observer, for any observable context oc of the given class,

$$\Psi(f_{k+1}(v_{k+1}))...\Psi(f_n(v_n)).oc \quad \text{(c)}$$

is still an observable context on the given class. According to Definitions 2.7 and 2.8, by applying (c) to both sides of (a), we have

$$\Psi(f_0(v_0)).\Psi(f_1(v_1))...\Psi(f_n(v_n)).oc$$
$$\approx \Psi(f_0(v_0)).\Psi(f_1(v_1))...\Psi(f_{j-1}(v_{j-1})).\Psi(g_1(w_1)).\Psi(g_2(w_2))...\Psi(g_m(w_m))$$
$$.\Psi(f_{k+1}(v_{k+1}))...\Psi(f_n(v_n)).oc$$

This means that the objects produced by s_1 and s_{12} are observationally equivalent. According to Definition 2.8, we have $s_1 \approx s_{12}$.

By the same argument, we have

$$s_{12} \approx s_{13} \approx ... \approx s_{1k} \approx s^*.$$

Therefore, $s_1 \approx s^*$. Similarly, we can prove that $s_2 \approx s^*$. Hence, $s_1 \approx s_2$. □

2.4 Algorithm GFT for Generating a Finite Number of Test Cases

In general, the axioms of an algebraic specification may contain branch conditions. An axiom may induce an infinite number of different fundamental pairs by assigning different normal forms to its variables. Exhaustive testing is impossible. How do we select a finite number of representative test cases from the infinite set of fundamental pairs? We present the following Algorithm GFT to deal with this problem. We give a related definition first.

Definition 2.12 Let m_f be an implemented method. We say we apply the *path-based domain partition technique* (PDP technique) to m_f if we:

(a) Partition the *input domain* of m_f into subdomains such that all the test points in each subdomain cause a particular path in the implementation of m_f to be executed. Here, the partition concept follows White and Cohen [25], but the *path generation algorithm* is the same as the one proposed by Jeng and Weyuker [26].

(b) Use the *simplified domain-testing strategy* presented by Jeng and Weyuker [26] to select some *test points* from each subdomain, if the assumptions required by the strategy are satisfied.

(c) Otherwise, randomly select a test point from each subdomain.

Since the PDP technique is path-oriented, it obviously inherits the problems associated with path testing, such as an infinite number of paths and the identification of infeasible paths. Recently, Jeng and Weyuker [26] proposed an innovative technique for detecting domain errors. Instead of the traditional approach of testing whether a border is correct, they test whether or not there is a displaced area. Their new perspective has greatly improved the practicality of White and Cohen's domain-testing strategy by removing most of the unrealistic constraints in its original model. Furthermore, although their new technique has a lower cost, the effectiveness is comparable. They also propose a path generation algorithm in which all the selected paths are executable, and hence infeasible paths are no longer an issue in the implementation. In view of all the above merits, we have adopted Jeng and Weyuker's method in our algorithm.

Algorithm GFT (**G**enerating a **F**inite number of **T**est cases) The algorithm asks the analyst to supply a canonical specification, and requests the designer to identify the mapping from the set of specified operations to the set of implemented methods. According to Theorem 2, we need not produce general equivalent ground terms as test cases. We need only construct fundamental pairs, which are produced from each axiom in the given canonical specification. Suppose the given specification contains n axioms $a_1, a_2, ..., a_n$. For each axiom a_i ($i = 1, 2, ..., n$), conduct the following steps:

(a) If a variable V of type T involved in a_i is not observable, use the syntax part of the given specification as a grammar [27] to construct all patterns of normal forms from the creators and constructors[6] of type T, such that their lengths do not exceed some positive integer k. Then replace each occurrence of V in a_i by these patterns to unfold a_i into several new equations, which are further unfolded until all the variables involved in the new equations a_{ij} are of observable types.

The above positive integer k may be determined by a white-box technique, such as by referring to the maximum sizes of arrays, or the boundary values of variables declared in the implemented code.

[6] We can infer from Definition 2.4 that a normal form contains only a creator and some constructors, but no transformer.

If the maximum sizes or the boundary values are too large for the generation of test sets of reasonable sizes, ask the user to specify an acceptable value of k.

(b) Suppose the right hand side of a new equation a_{ij} obtained in step (a) contains a defined operation f. Use the conditions of the set of axioms defining f to partition the input domain of f into subdomains.

(c) Randomly select an element from each subdomain obtained in step (b), and use these elements to replace all occurrences of the corresponding input variables in equation a_{ij} to obtain a group of fundamental pairs induced from axiom a_i.

(d) If the above group of fundamental pairs reveals an error, exit from the algorithm. Otherwise go to step (e).

(e) Suppose the defined operation f in step (b) is implemented by method m_f. Apply the PDP technique to m_f for selecting input data points to replace all occurrences of the corresponding variables in equation a_{ij}, and hence obtain another group of fundamental pairs for axiom a_i. ■

Example 3 below is used to illustrate Algorithm GFT.

Example 3 The specification is nearly the same as Example 1, except the following additional entries:

operations
...
_.ascending: *IntStack* → *Bool*
variables
...
*N*1 *N*2: *Int*
axioms
...
a_7: *new.ascending* = *true*
a_8: *new.push*(*N*).*ascending* = *true*
a_9: *S.push*(*N*1).*push*(*N*2).*ascending* = *N*1 ≤ *N*2 and *S.push*(*N*1).*ascending*

Suppose the following axioms in class *INT* define the operation "≤":

b_1: (*N*1 ≤ *N*2) = *true* *if N*1 == *N*2
b_2: (*N*1 ≤ *N*2) = *true* *if N*1 < *N*2
b_3: (*N*1 ≤ *N*2) = *false* *if N*2 < *N*1

Following Algorithm GFT, we should conduct steps (a), (b), and (c) for each of the axioms a_1 to a_9. For simplicity, however, we shall only illustrate the procedure for axiom a_9.

(a) a_9 includes a variable *S* that is not observable. Determine a positive integer k for *S*. For the sake of illustration, suppose k = 3. The patterns of normal forms of *S* of lengths ≤ 3 are as follows:

S = *new*,
S = *new.push*(*N*0).

By replacing S in a_9 with the above patterns, we unfold a_9 into the following new equations:

a_{91}: *new*.*push*($N1$).*push*($N2$).*ascending* = ($N1 \le N2$) *and new*.*push*($N1$).*ascending*
a_{92}: *new*.*push*($N0$).*push*($N1$).*push*($N2$).*ascending*
= ($N1 \le N2$) *and new*.*push*($N0$).*push*($N1$).*ascending*

(b) The right-hand side of a_{91} contains an operation $\le$ defined by axioms b_1, b_2, and b_3. Use the conditions of b_1, b_2, and b_3 to partition the input domain of the operation $\le$ into the following subdomains:

(1) $N1 = N2$
(2) $N1 < N2$
(3) $N2 < N1$

Furthermore, we partition the input domain into the following subdomains for axiom a_{92}:

(1) $N1 = N2$ *and* $N0 = N1$
(2) $N1 = N2$ *and* $N0 < N1$
(3) $N1 = N2$ *and* $N1 < N0$
(4) $N1 < N2$ *and* $N0 = N1$
(5) $N1 < N2$ *and* $N0 < N1$
(6) $N1 < N2$ *and* $N1 < N0$
(7) $N2 < N1$ *and* $N0 = N1$
(8) $N2 < N1$ *and* $N0 < N1$
(9) $N2 < N1$ *and* $N1 < N0$

(c) Replace the variables $N1$, $N2$, and $N0$ in the above axioms a_{91} and a_{92} by some integers randomly selected from the corresponding subdomains above, thus resulting in the following fundamental pairs induced from a_9 as a part of test cases:

new.*push*(1).*push*(1).*ascending* ~ (1 ≤ 1) *and new*.*push*(1).*ascending*
new.*push*(1).*push*(2).*ascending* ~ (1 ≤ 2) *and new*.*push*(1).*ascending*
new.*push*(2).*push*(1).*ascending* ~ (2 ≤ 1) *and new*.*push*(2).*ascending*
new.*push*(2).*push*(2).*push*(2).*ascending*
~ (2 ≤ 2) *and new*.*push*(2).*push*(2).*ascending*
new.*push*(1).*push*(2).*push*(2).*ascending*
~ (2 ≤ 2) *and new*.*push*(1).*push*(2).*ascending*
new.*push*(3).*push*(2).*push*(2).*ascending*
~ (2 ≤ 2) *and new*.*push*(3).*push*(2).*ascending*
new.*push*(3).*push*(3).*push*(4).*ascending*
~ (3 ≤ 4) *and new*.*push*(3).*push*(3).*ascending*
new.*push*(2).*push*(3).*push*(4).*ascending*
~ (3 ≤ 4) *and new*.*push*(2).*push*(3).*ascending*
new.*push*(5).*push*(3).*push*(4).*ascending*
~ (3 ≤ 4) *and new*.*push*(5).*push*(3).*ascending*
new.*push*(4).*push*(4).*push*(2).*ascending*
~ (4 ≤ 2) *and new*.*push*(4).*push*(4).*ascending*

new.*push*(3).*push*(4).*push*(2).*ascending*
~ (4 ≤ 2) *and new*.*push*(3).*push*(4).*ascending*
new.*push*(5).*push*(4).*push*(2).*ascending*
~ (4 ≤ 2) *and new*.*push*(5).*push*(4).*ascending* ■

It should be noted that if *k* in step (a) is not well chosen, some implementation errors may not be revealed, as illustrated in Example 4. Thus, the selection of *k* is important, but difficult as indicated in Section 2.5.2. This may warrant further investigation.

Example 4 Let us refer to the specification in Example 1 again. Suppose the implementation is as follows, where *array*[100] is the top of the stack and *array*[1] is the bottom.

```
#include <iostream.h>
#define SIZE 100
#define NIL 0

class intStack {
    int array[SIZE];        /* Only one data member */
public: ...
};
...

void intStack :: newStack( )
{
    for ( int j = 1; j <= 100; j++ )
        array[j] = NIL;
}

void intStack :: push(int i)
{
    for ( int j = 1; j <= 99; j++ )
        array[j] = array[j+1];
    array[100] = i;
}

void intStack :: pop( )
{
    for ( int j = 100; j > = 2; j-- )
        array[j] = array[j–1];
    array[1] = NIL;
}

int intStack::top( )
{
    return array[100];
}
...
```

Let $u_1 = new.push(1).push(2)...push(100).push(101).pop$
$u_2 = new.push(1).push(2)...push(100)$.
Obviously, $u_1 \sim u_2$ is a fundamental pair. The following objects O_1 and O_2 are produced by u_1 and u_2, respectively:

$O_1 = [NIL, 2, ..., 100]$
$O_2 = [1, 2, ..., 100]$.

Since O_1 and O_2 are not observationally equivalent, this implementation contains an error. In step (a) above, suppose for argument's sake we have chosen $k = 10$ for the variable S in axiom a_4. Then the error cannot be revealed. ■

2.5 Discussions on Algorithm GFT

In this section, we discuss a number of important issues on Algorithm GFT including assumptions, limitations, applicability, and complexity.

2.5.1 *Assumptions*

We have assumed in Theorem 2 and Algorithm GFT that a canonical specification and a complete implementation are given. How restrictive are these requirements?

Intuitively, the normal form of a ground term denotes the "abstract object value" [28] of this ground term. Two ground terms having the same normal form would have the same "abstract object value". Thus, according to Definition 2.3, every ground term under a canonical specification would have a unique "abstract object value", hence avoiding any ambiguity. In other words, if we relax the canonical requirement for a specification, the ground terms may be ambiguous.

If an implementation is not complete, there exists some operation f_0 such that (1) no method implements it or (2) two or more distinct methods implement it in the same class. Case (1) is obviously an error, since the implemented system will fail when f_0 is called. Case (2), on the other hand, is ambiguous, since the implemented system can have two distinct outcomes. In both cases, the problem can easily be detected by comparing a checklist of all the operations in the specification against the corresponding methods in the implementation.

To summarize, in order to avoid omissions and ambiguities, it is acceptable to require a specification to be canonical and an implementation to be complete.

2.5.2 *Limitations*

It is difficult to determine the positive integer k in step (a) of Algorithm GFT. It will be helpful to apply a white-box technique, such as referring to the maximum sizes of arrays, or the boundary values of variables declared in the implemented code. However, when the maximum sizes or the boundary values are large, the sizes of the test cases may be of the order $O(m^k)$ (where m is the number of constructors in the class under test) and hence unreasonable. Furthermore, even when the maximum sizes or the boundary values are not large, some of the faults on the capabilities for handling excess of the maximum sizes or the boundary values may not be identified. This is a natural limitation of step (a). Similarly, the PDP technique in step (e) of Algorithm GFT cannot be fully automated.

As an optional heuristic, we may supplement the algorithm by the "weak class graph" and "weak coverage criteria" approaches proposed by [15] for selecting the normal forms in step (a) of Algorithm GFT. (See Section 4.3 for more details.)

Alternatively, we may have to ask the user to choose k for the algorithm (similarly to [21]) when the white-box technique fails. Thus, Algorithm GFT can be implemented as a semi-automatic CASE tool that interacts with users when the above problems are encountered.

2.5.3 *Effectiveness and Applicability Issues*

In step (a) of Algorithm GFT, we replace every variable of nonobservable types by a finite number of patterns of normal forms with limited lengths. In fact, this is a common practice in testing, and has been formalized in [21] by means of a regularity hypothesis. The random selection of a value from each subdomain in step (c) of Algorithm GFT and PDP technique is also a common practice in testing. It has also been formalized in [21] by means of a uniformity hypothesis.

For example, consider a program "*if* $X > 0$ *then* $Y = f(X)$ *else* $Y = -X$". Suppose "$Y = f(X)$" is a computational error that should be corrected to "$Y = g(X)$". We partition the input domain of X into two subdomains $sb_1 = \{X \mid X > 0\}$ and $sb_2 = \{X \mid X \le 0\}$. Let $solutionSet_1 = \{X \mid X > 0 \text{ and } f(X) = g(X)\}$. Suppose t is some randomly selected test data from sb_1. If $t \in solutionSet_1$, the error cannot be revealed. However, if $t \in (sb_1 \setminus solutionSet_1)$, the error can be exposed. In many practical cases, we can expect the cardinal number of the set $(sb_1 \setminus solutionSet_1)$ to be much greater than that of the $solutionSet_1$. For instance, in many programs, $f(X)$ and $g(X)$ are arithmetic expressions. In this case, the $solutionSet_1$ is finite but the set $(sb_1 \setminus solutionSet_1)$ is infinite. This means that the probability of $t \in (sb_1 \setminus solutionSet_1)$ is much greater than that of $t \in solutionSet_1$. In other words, in such cases, the probability of revealing the error by the randomly selected t from the subdomain is much greater than that of not revealing it.

The "simplified domain-testing strategy" and its corresponding "path generation algorithm", adopted from [26] and used in Definition 2.12 and step (e) of Algorithm GFT of this paper, have been shown by the original authors to be effective and applicable.

2.5.4 *Complexity Issue*

Algorithm GFT is similar to that used in the tools described in Sections 5 and 6 of Bouge et al. [9], except the following differences: (1) We suggest in Algorithm GFT to use a white-box technique to determine the positive number k in step (a), whereas Bouge et al. regard k as a part of the regularity hypothesis. (2) In Algorithm GFT, we replace all the variables in the unfolded equations by normal forms, while Bouge et al. replace them by ground terms.

In practice, the complexity of Algorithm GFT depends heavily on the actual number of normal forms generated for a given positive integer k. According to Definition 2.4, a normal form contains only a creator and a number of constructors, but no transformer, whereas a ground term may contain all three types of operations. In most situations, a class contains more transformers than creators and constructors. Hence, our proposal in the algorithm to replace variables by normal forms, rather than ground terms in general, enhances the efficiency of testing.

3. DETERMINING THE OBSERVATIONAL EQUIVALENCE OF TWO OBJECTS

Suppose the fundamental pair $u_1 \sim u_2$ is selected as a test case for a given specification. To apply this test case to an implementation, we should map each operation in u_1 and u_2 to a method in the program. As a result, this mapping generates two method sequences s_1 and s_2 in the program corresponding to u_1 and u_2, respectively. For a complete implementation, this mapping exists and can be indicated manually by the implementation designer, or be derived automatically from a given interface specification. Let O_1 and O_2 be two objects resulting from the execution of s_1 and s_2, respectively. After executing s_1 and s_2 in the program and obtaining results O_1 and O_2, in order to judge whether the test case $\{u_1, u_2\}$ reveals an implementation error, we have to decide whether O_1 and O_2 are observationally equivalent (denoted by $O_1 \approx O_2$). In Section 3.1, we explain why this problem is undecidable using black-box techniques, and indicate that we have to use a heuristic white-box technique to select a relevant finite subset of the set of observable contexts.

3.1 Reason for Using a Heuristic White-Box Technique

According to Definition 2.7, we can use the observable contexts on class C to determine whether $O_1 \approx O_2$. Unfortunately, the set of all observable contexts in class C is infinite in general. How do we select a finite subset? For the stack example (O_1 and O_2 are stacks), it intuitively seems that

$$
\begin{aligned}
(O_1 \approx O_2) \Leftrightarrow\ & (O_1.height \approx O_2.height)\ \textit{and} \\
& (O_1.top \approx O_2.top)\ \textit{and} \\
& (O_1.pop.top \approx O_2.pop.top)\ \textit{and} \\
& \ldots\ \textit{and} \\
& (O_1.pop^{(O_1.height)}.top \approx O_2.pop^{(O_2.height)}.top)
\end{aligned}
\qquad \text{(Formula I)}
$$

Let $SS = \{height\} \cup \{pop^i.top \mid i = 0, 1, ..., height\}$. Although the subset SS of observable contexts is finite, Formula I is unfortunately still incorrect. A counterexample[7] is given in Bernot et al. [21]. The authors then added, "we get the depressing result that the only credible alternative is to consider the set of all observable contexts, which is infinite (and consequently impracticable)." They simply regard Formula I as a hypothesis, known as an "oracle hypothesis"[8], for the class of integer stacks.

In fact, we can formally prove that the observational equivalence of two objects cannot be decided by a black-box technique.

Theorem 3 For any given class, let *AllOCs* be the set of observable contexts, *CT* be the set of constructors and transformers, and *OBS* be the set of observers. If *CT* and *OBS* are non-empty, then *AllOCs* is infinite. Furthermore, if the class has at least one pair of equivalent ground terms $u_1 \sim u_2$ and a constructor or transformer $f(...) \in CT$ such that the numbers of appearances of $f(...)$ in u_1 and u_2 are

[7] Example 5 in Section 3.2 is another counter-example. It concludes that the objects O_1 = ([1, 2], 1, 0, 0) and O_2 = ([1, *NIL*], 1, 1, 0) are not observationally equivalent, but the finite subset *SS* of observable contexts shown above will report that they are observationally equivalent.

[8] The oracle hypothesis is an attempt by [21] to formalize the basic assumptions about the oracle problem in software testing. According to the authors, "the oracle problem [is related with] how to decide if a program execution returns a correct result. The solutions to this problem depend both on the kind of formal specification and program; a property required by the specification may not be observable using the program under test. Most of the formal specification methods provide a way to express observability. In this case, the program is assumed to satisfy the observability requirements (for instance, to decide correctly the equality of two integers); it is [known as] an oracle hypothesis."

different, then the observational equivalence of objects cannot be determined using a finite set of observable contexts selected independently of implementations. In other words, given any finite subset *SubOCs* of *AllOCs*, there exist some implementation and two objects O_1 and O_2 such that

$$(\forall oc \in SubOCs)\ (O_1.oc \approx O_2.oc) \wedge (\exists oc_0 \in AllOCs)\ (\neg(O_1.oc_0 \approx O_2.oc_0)) \qquad \text{(Formula II)}$$

That is, there exist objects that are not observationally equivalent, but appear to be so when only a finite subset of the observational contexts are applied.

Proof:

Since *CT* and *OBS* are non-empty, there exist a constructor or transformer *g*(...) in *CT* and an observer *obs*(...) in *OBS*. Given any positive integer *i*, we can construct an $oc_i = g(...)...g(...).obs(...)$ in *AllOCs* that contains *i* *g*(...)'s. Hence, *AllOCs* is infinite.

Suppose u_1 contains *m* *f*(...)'s and u_2 contains *n* *f*(...)'s, such that $0 \le n < m$. Consider any given finite subset *SubOCs* of *AllOCs*, selected independently of implementations. We can find an oc_1 from this *SubOCs* such that the number *k* of *f*(...)'s in oc_1 is maximal in *SubOCs* (where $k \ge 0$). We can then construct an implementation Ψ that contains an error in the $(m+k+1)$th call of *f*(...) but correct otherwise. Let O_1 and O_2 be the objects resulting from executing the method sequences corresponding to u_1 and u_2, respectively. For simplicity, we write $O_1 = \Psi(u_1)$ and $O_2 = \Psi(u_2)$. Since $O_1.oc_1 = \Psi(u_1).oc_1$ contains $m+k$ *f*(...)'s and $O_2.oc_1 = \Psi(u_2).oc_1$ contains $n+k$ *f*(...)'s such that $n < m$, for any $oc \in SubOCs$, there must be no more than $m+k$ *f*(...)'s in $O_1.oc$ or $O_2.oc$. Thus, according to the construction of implementation Ψ,

$$O_1.oc \approx O_2.oc$$

for any *oc* in *SubOCs*. By Definition 2.5, oc_1 must end with an observer. Let *u* be the result of removing the observer from oc_1. Consider $u^* = u.f(...)$. Obviously, since u^* contains $k+1$ *f*(...)'s,

$O_1.u^* = \Psi(u_1).u^*$ contains $m+k+1$ *f*(...)'s, and
$O_2.u^* = \Psi(u_2).u^*$ contains $n+k+1 \le m+k$ *f*(...)'s,

according to the construction of implementation Ψ, we have $\neg(O_1.u^* \approx O_2.u^*)$. By Definition 2.7, there exists an observable context oc_2 such that $\neg(O_1.u^*.oc_2 \approx O_2.u^*.oc_2)$. Let $oc_0 = u^*.oc_2$. It follows that

$$\neg(O_1.oc_0 \approx O_2.oc_0).$$

Since u^* contains $k+1$ *f*(...)'s, oc_0 must contain at least $k+1$ *f*(...)'s, and hence $oc_0 \notin SubOCs$ but $oc_0 \in AllOCs$. Thus we arrive at Formula II. ■

In the above proof, we realize that the number $m+k+1$ is closely related to this error. In order to reveal this error, we must catch the number $m+k+1$. Obviously, this can be done only by using a heuristic white-box technique, rather than a black-box technique. This is the reason why we propose to supplement our axiom-based black-box approach by the following heuristic white-box technique. (On the other hand, see also Section 3.3.2 that discusses why a white-box technique cannot be a substitute for the black-box technique.)

3.2 Relevant Observable Context Technique

The basic idea behind our heuristic technique is as follows. Suppose we want to determine whether $O_1 \approx O_2$. Suppose further that O_1 and O_2 have different values for the same data member d_i of the implemented class. Such different values may or may not have an effect on the observable attributes of O_1 and O_2. If no observable attribute is affected, d_i need not be considered. If some observable attribute is affected, d_i must have affected the attribute through some series of methods in the implemented class. Such a series of methods is called a *relevant observable context*. We need only use the relevant observable contexts to determine whether $O_1 \approx O_2$. We can ignore any other observable contexts for this decision. We shall give a formal definition for the concept of relevant observable context, and how it can be produced by means of a Data member Relevance Graph constructed from the implemented class *C*.

If a relevant observable context itself contains an implementation error, by applying it to determine the observational equivalence of objects, we may increase the chance of having the error revealed. If, after applying the relevant observable context to objects O_1 and O_2, we find inconsistencies in some observable attribute, we can conclude an implementation error in s_1 or s_2, or in the relevant observable context itself, or both. The worst case scenario happens when the error(s) in s_1 and s_2 offset the error(s) in the relevant observable context, so that neither can be revealed. We note, however, that the possible offsetting of errors cannot be avoided in testing. Even if we test a single method sequence, errors in two of the methods may happen to cancel each other. This is the well-known phenomenon of fault masking (see, for example, Morell [1990]).

In the relevant observable context technique, we assume a program model without pointers. This kind of program model is gaining popularity in the latest object-oriented programming languages such as JAVA.

Definition 3.1 If a data member d_1 is defined or revised by a data member d_2 in a method m under a condition $p(..., d_3, ...)$, we say that d_2 *directly affects* d_1 in m under $p(...)$, and that d_3 *directly affects* d_1 in m under $p(...)$.

In Example 5 below, for instance, we say that the data member *numPush* directly affects the data member *totalPush* in the method *incTop* under the condition *height* > 0.

Definition 3.2 Given an implemented class *C*, its ***D**ata member **R**elevance **G**raph* (DRG) is constructed as follows. Each data member of *C* is represented as a bold rectangle *node* in the DRG. The DRG also contains some thin rectangle *nodes*, which denote some constants coming from the given program. If the data member d_2 directly affects the data member d_1 in the method m_1 under a condition $p(...)$, then there is an *arc*, labeled by (p, m_1), from d_2 to d_1. (See Figure 2.) We call $[d_2, (p, m_1), d_1]$ a *segment* of the DRG, d_2 a *start node* of arc (p, m_1), d_1 an *end node* of arc (p, m_1), (p, m_1) an *output arc* of d_2, and (p, m_1) an *input arc* of d_1. If d_2 is identical to d_1, the segment is said to be a *cycle*. Otherwise it is said to be *acyclic*. Each DRG contains a special *node* called *observed*, which is the ending node of each arc with an observer as the second component of its label. An arc with *observed* as an ending node is call an *observer arc*. An example of a DRG is given in Figure 3.

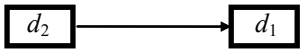
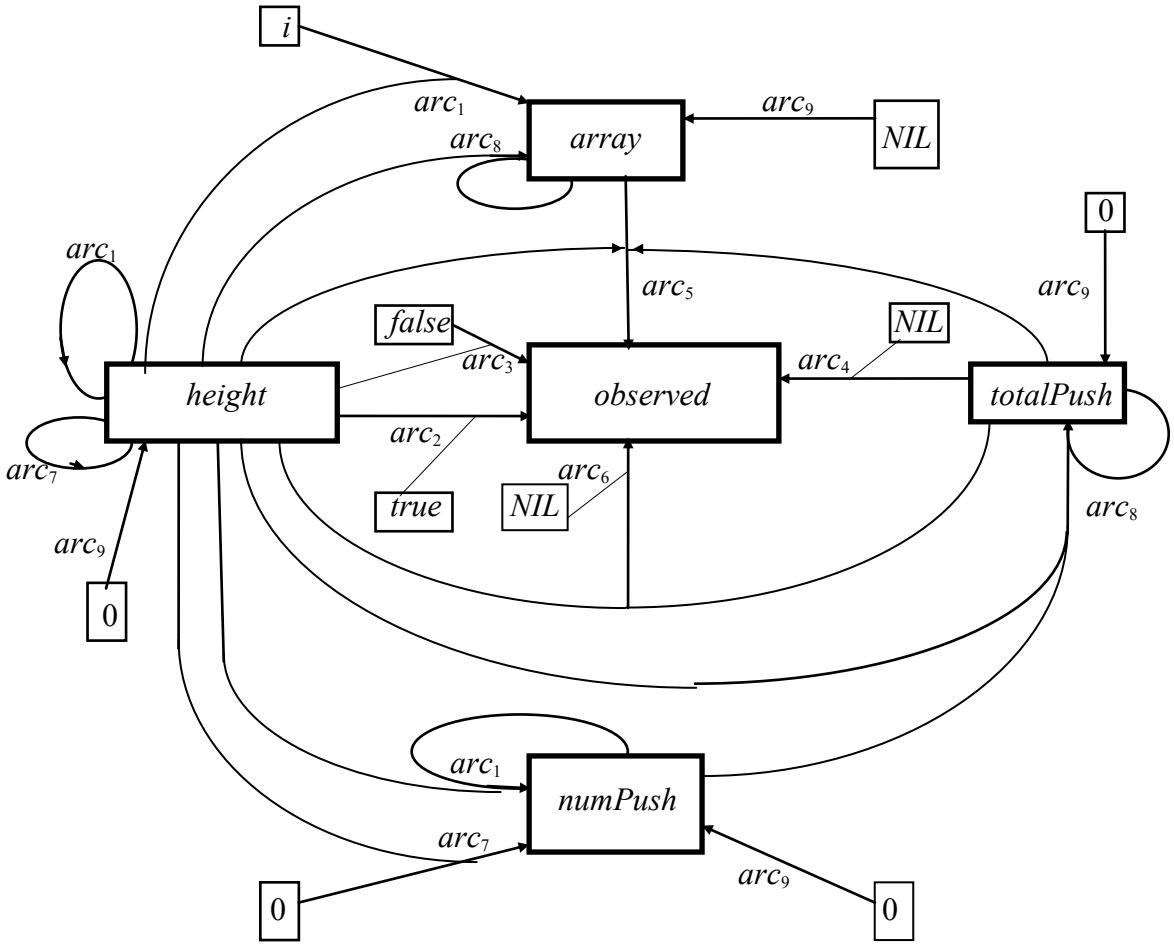


Figure 2. Nodes and arc in a DRG



arc_1: (*height* ≠ *size*, *push*(*i*));
arc_2: (*height* = 0, *empty*);
arc_3: (*height* ≠ 0, *empty*);
arc_4: (*totalPush* = 3, *top*);
arc_5: (*totalPush* ≠ 3 ∧ *height* > 0, *top*);
arc_6: (*totalPush* ≠ 3 ∧ *height* = 0, *top*);
arc_7: (*height* > 0, *pop*);
arc_8: (*height* > 0, *incTop*);
arc_9: (*true*, *newStack*);

Figure 3. DRG of integer stacks

Definition 3.3 Suppose d_1 is a data member of an implemented class C, O_1 and O_2 are two given objects of C. If

(1) $O_1.d_1 \neq O_2.d_1$,

(2) there is a path P from the node d_1 to the node *observed* in the DRG of class C, and

(3) the methods in the labels of the arcs in path P are op_1, op_2, ..., op_t, *obs* successively,

then we call $op_1.op_2...op_t.obs$ a *relevant observable context* induced from path P with respect to O_1 and O_2, and say that d_1 *affects* the observable attribute of O_1 and O_2. Notice that, the concept of *paths* in this paper is the same as that in directed graphs in general, except they end at a special node *observed*.

Definition 3.4 Let O_1 be an object of the implemented class C. Suppose the data members of class C are d_1, d_2, ..., d_n. In the DRG of class C, if all the conditions in the labels of the arcs in a given path P are satisfied by $O_1.d_i$ as initial data, then the path P is said to be *executable for* O_1. Otherwise P is said to be *unexecutable for* O_1.

Algorithm DOE (**D**etermining **O**bservational **E**quivalent) Suppose O_1 and O_2 are two objects of the same implemented class, resulting from the execution of the method sequences s_1 and s_2, respectively. The steps for deciding whether $O_1 \approx O_2$ by means of relevant observable contexts are as follows:

(a) If s_1 and s_2 end with an observer, then O_1 and O_2 are values of some import class. We can therefore directly decide whether $O_1 \approx O_2$ in the import class.

(b) Otherwise, suppose O_1 and O_2 belong to the implemented class C. Construct the Data member Relevance Graph of class C from the coding of class C.

(c) Suppose the data members of the implemented class C are d_1, d_2, ..., d_n. In general, the classes of d_i are imported, observable, and there are known methods to determine the equivalence of values of the classes. Suppose further that $O_t.d_i$ denotes the value of d_i of O_t for $i = 1, 2, ..., n$ and $t = 1, 2$.

Check whether the tuples ($O_1.d_1$, $O_1.d_2$, ..., $O_1.d_n$) and ($O_2.d_1$, $O_2.d_2$, ..., $O_2.d_n$) are equal. If yes, we have $O_1 \approx O_2$, and exit from Algorithm DOE. Otherwise proceed to step (d).

(d) Suppose O_1 and O_2 have different values with respect to the data members d_{x1}, d_{x2}, ..., d_{xk}, where $1 \leq x_1 < x_2 < ... < x_k \leq n$. In other words, suppose $O_1.d_{xj} \neq O_2.d_{xj}$ for $j = 1, 2, ..., k$.

For each d_{xj}, check whether there is a path from the node d_{xj} to the node *observed* in the DRG. If not, skip this d_{xj}. If yes, proceed as follows:

(1) If d_{xj} is a simple data type, traverse every acyclic executable path P once (using the original $O_1.d_i$ as **initial data** and **backtracking** if necessary) and obtain the relevant observable context *oc* induced from P. If there are uninstantiated input variables in *oc*, apply the PDP technique to select values for the input variables.

If a cycle l_i is encountered when traversing an executable path for O_1, the user should manually decide on a ceiling t_i for the number of iterations of l_i, or supply a global ceiling T allowed by the system.

Check whether at least one of these relevant observable contexts, say oc_0, fails, that is, $\neg(O_1.oc_0 \approx O_2.oc_0)$. If so, we have $\neg(O_1 \approx O_2)$, and exit from Algorithm DOE. Otherwise we say this d_{xj} has successfully passed the check, and proceed to step (3).

(2) If d_{xj} is a compound data type (such as an array or structure in C++), construct relevant observable contexts by the following process: For each value V of the component index or element variable of d_{xj} that satisfies $O_1.d_{xj}.V \neq O_2.d_{xj}.V$, select every method sequence *msEl* to change the current value of the component index or element variable to V, then traverse each executable path (not exceeding the iteration ceilings in the case of cycles, if any) for O_1 from the node d_{xj} to the node *observed* to obtain method sequence *msOb*, and create a relevant observable context *oc* = *msEl.msOb*. (If V is already the current value of the component index or the current element variable, then *msEl* is empty.) The process of traversing each executable path (not exceeding the iteration ceilings in the case of cycles, if any) for O_1 in this step is the same as that in step (1). If *oc* contains uninstantiated input variables, the PDP technique is applied to determine values for the input variables. If at least one of these relevant observable contexts, say oc_0, fails, then we have $\neg(O_1 \approx O_2)$, and exit from Algorithm DOE. Otherwise we say this d_{xj} has successfully passed the check, and proceed to step (3).

(3) If all the d_{xj} have successfully passed the checks, then we have $O_1 \approx O_2$, and exit from Algorithm DOE. Otherwise continue to check the next d_{xj} such that $O_1.d_{xj} \neq O_2.d_{xj}$. ■

Example 5 The specification is the same as Example 1, except for following additional operations and axioms:

operations

...

_.incTop: *IntStack* → *IntStack*

axioms

...

a_7: *S.incTop.top* = *if S.empty then S.top*
else S.top + 1

In the following implementation, the internal data member *numPush* is used to count the number of continuous calls to *_.push*, and *totalPush* is applied to record the total number of calls to *_.push*. For the sake of illustration, we have embedded some errors in the implementation.

```
#include <iostream.h>

#define SIZE 100
#define NIL 0

enum bool { false, true };
```

```
class intStack {
      /* intStack consists of 4 data members: */
      int array[SIZE];
      int height;
      int numPush;
      int totalPush;
public:
      void newStack( );
      bool empty( );
      void push(int i);
      void pop( );
      void incTop( );
      int top( );
};

void intStack :: newStack( )
{
      height = 0;
      numPush = 0;
      totalPush = 0;
      for ( int j = 1; j <= 100; j++ )
            array[j] = NIL;
}

bool intStack :: empty( )
{
      if (height == 0) return true;
      else return false;
}

void intStack :: push(int i)
{
      if (height == SIZE)
            cout << "Stack is full";
      else {
            height = height + 1;
            array[height] = i;
            numPush = numPush + 1;
      }
}

void intStack :: pop( )
{
      if (height > 0) {
            height = height – 1;
            numPush = 0;
      }
}
```

```
void intStack :: incTop( )
{
    if (height > 0) {
        array[height] = array[height] + 1;
        totalPush = totalPush + numPush; /*  Error 1: This statement should be in the
                                             method pop but has been placed here by
                                             mistake. */
    }
}

int intStack :: top( )
{
    if (totalPush == 3) return NIL;      /* Error 2: The condition should be totalPush == 0. */
    else {
        if (height > 0) return array[height];
        else return NIL;
    }
}
```

The fundamental pair u_1 = *new*.*push*(1).*push*(2).*pop* and u_2 = *new*.*push*(1) can be induced from axiom a_4. Let us denote their corresponding implemented method sequences as s_1 and s_2, respectively. Suppose the execution results of s_1 and s_2 are objects O_1 and O_2, respectively. We would like to illustrate how to use Algorithm DOE to determine whether $O_1 \approx O_2$.

(a) Since the sequences s_1 and s_2 do not end with an observer, proceed to step (b).

(b) Construct the Data member Relevance Graph of class *C* from the coding of class *C*. The DRG is shown in Figure 3.

(c) The execution results are:

	(***array***, ***height***, ***numPush***, ***totalPush***)
O_1 =	([1, 2], 1, 0, 0),
O_2 =	([1, *NIL*], 1, 1, 0).

Check whether the tuples ([1, 2], 1, 0, 0) and ([1, *NIL*], 1, 1, 0) are equal. The answer is no.

(d) O_1 and O_2 have different values on the data members *array* and *numPush*:

(1) For *array*, we follow step (d)(2) of Algorithm DOE. Here, the component index is *height*. The current values of both O_1.*height* and O_2.*height* are 1, whereas the value of *height*, which satisfies O_1.*array*[*height*] ≠ O_2.*array*[*height*], is 2. From the cycle *height*-arc_1-*height* in Figure 3, we see that the only method sequence $msEl_1$ which changes the value of *height* from 1 to 2 is *push*(*i*). On the other hand, by traversing the executable paths for O_1 from the node *array* to the node *observed* in Figure 3, we obtain the method sequences $msOb_1$ = *top* and $msOb_2$ = *incTop*.*top*, corresponding to the paths *array*-arc_5-*observed* and *array*-arc_8-*array*-arc_5-*observed*, respectively. Thus, by concatenating $msEl_1$ with $msOb_1$ and $msOb_2$, respectively, we

obtain the relevant observable contexts *push*(*i*).*top* and *push*(*i*).*incTop*.*top*. Then apply the PDP technique[9] to determine $i = 8$, and check whether

O_1.*push*(8).*top* = O_2.*push*(8).*top* and
O_1.*push*(8).*incTop*.*top* = O_2.*push*(8).*incTop*.*top*

They are both successful.

(2) For *numPush*, since it is a simple data type, we follow step (d)(1) of Algorithm DOE to traverse every executable path for O_1 by backtracking. Suppose the global ceiling T supplied by the user for the number of iterations of cycles is 2.

(i) In Figure 3, the node *numPush* has two output arcs arc_8 and arc_1 (where arc_1 is a cycle). All the conditions of arc_8 and arc_1 are satisfied by O_1.*height* = 1. Let us consider arc_8 first. Its label contains the method *incTop*. Execute the method and obtain O_1.*incTop* = ([2, 2], 1, 0, 0). The end node of arc_8 is *totalPush*, which has four output arcs arc_4, arc_5, arc_6, and arc_8 (where arc_8 is a cycle). The current state, O_1.*incTop* = ([2, 2], 1, 0, 0), satisfies the conditions of arc_5 and arc_8 but not those of arc_4 and arc_6. Hence, arc_5 and arc_8 are executable for the current state but arc_4 and arc_6 are unexecutable. Consider arc_5 first. The label of arc_5 contains the method *top*. Execute the method and obtain (O_1.*incTop*).*top* = 2. The end node of arc_5 is the node *observed*. Thus, we obtain an executable path p_1 = *numPush*-arc_8-*totalPush*-arc_5-*observed* for the given object O_1. The relevant observable context corresponding to path p_1 is oc_1 = *incTop*.*top*. Execute O_2.oc_1 and obtain O_2.oc_1 = 2. Hence, oc_1 succeeds because O_1.oc_1 = O_2.oc_1.

(ii) To obtain the other executable path for O_1, backtrack to the node *totalPush* and consider the other executable output arc arc_8 (which is a cycle). The label of arc_8 contains the method *incTop*. Execute the method and obtain (O_1.*incTop*).*incTop* = ([2, 2], 1, 0, 0).*incTop* = ([3, 2], 1, 0, 0). The end node of arc_8 is *totalPush*, which has four output arcs arc_4, arc_5, arc_6, and arc_8. The current state, O_1.*incTop*.*incTop* = ([3, 2], 1, 0, 0), satisfies the conditions of arc_5 and arc_8 but not those of arc_4 and arc_6. Hence, arc_5 and arc_8 are executable for the current state but arc_4 and arc_6 are unexecutable. Consider arc_5 first. The label of arc_5 contains the method *top*. Execute the method and obtain (O_1.*incTop*.*incTop*).*top* = 3. The end node of arc_5 is the node *observed*. Thus, we obtain an executable path p_2 = *numPush*-arc_8-*totalPush*-(arc_8-*totalPush*)^1-arc_5-*observed* for the given object O_1. The relevant observable context corresponding to path p_2 is oc_2 = *incTop*.*incTop*.*top*. Execute O_2.oc_2 and obtain O_2.oc_2 = 3. Hence, oc_2 also succeeds because O_1.oc_2 = O_2.oc_2.

(iii) Similarly to step (ii), in order to obtain the other executable path for O_1, backtrack to the node *totalPush*, and consider the other executable output arc arc_8 (which is a cycle). The label of arc_8 contains the method *incTop*. Execute the method and obtain (O_1.*incTop*.*incTop*).*incTop* = ([4, 2], 1, 0, 0). The end node of arc_8 is *totalPush*, which has four output arcs arc_4, arc_5, arc_6, and arc_8. The current state, O_1.*incTop* = ([4, 2], 1, 0, 0), satisfies the conditions of arc_5 and arc_8 but not those of arc_4 and arc_6. Hence, arc_5 and arc_8 are executable for the current state but arc_4 and arc_6 are unexecutable. Consider arc_5 first. The label of arc_5 contains the method *top*. Execute the method and obtain

[9] Here, as a special case, the partition only contains a unique subdomain.

(O_1.*incTop*.*incTop*.*incTop*).*top* = 4. The end node of arc_5 is the node *observed*. Thus, we obtain an executable path p_3 = *numPush*-arc_8-*totalPush*-(arc_8-*totalPush*)^2-arc_5-*observed* for the given object O_1. The relevant observable context corresponding to path p_3 is oc_3 = *incTop*.*incTop*.*incTop*.*top*. Execute $O_2.oc_3$ and obtain $O_2.oc_3$ = ([1, *NIL*], 1, 1, 0).*incTop*.*incTop*.*incTop*.*top* = *NIL*. However, as evaluated above, $O_1.oc_3$ = O_1.*incTop*.*incTop*.*incTop*.*top* = 4. Hence, oc_3 fails because $O_1.oc_3 \neq O_2.oc_3$. Report $\neg(O_1 \approx O_2)$. Then exit from Algorithm DOE. ■

3.3 Discussions on Algorithm DOE

We discuss in this section a number of important issues on Algorithm DOE including effectiveness, limitations, and complexity.

3.3.1 *Effectiveness* (1): *Skipping Irrelevant Observable Contexts*

By adopting Algorithms DOE, we can skip the testing of many irrelevant cases. Referring to step (d) in Example 5, none of the method sequences of the form *push*(i_1).*push*(i_2)^j.*top* or *push*(i)^j.*pop*^{k}.*incTop*.*top* (where j, k = 1, 2, ...) are relevant observable contexts with respect to O_1 and O_2. Hence, we need not consider them. In fact, none of them reveals the error.

3.3.2 *Effectiveness* (2): *Overcoming the* "*Missing Path*" *Problem*

A common drawback of white-box techniques is the failure to detect "missing paths", which are parts of the specification omitted from the implementation. However, even though Algorithm DOE is a white-box technique by itself, it can help to expose some of the missing paths when integrated with a black-box technique, such as our axiom-based approach to generate fundamental pairs as test cases. This is the main idea behind our proposal to integrate black- and white-box techniques in program testing. In Example 5, for instance, suppose the branch "*if* (*height*>0) *return array*[*height*]" is missing from the code of the method *top*( ). Then the path "*totalPush*-arc_5-*observed*" in Figure 3 will be missing. The originally selected fundamental pair u_1 = *new*.*push*(1).*push*(2).*pop* and u_2 = *new*.*push*(1) (see the paragraph before step (a) of Example 5) cannot reveal this error, since $O_1.oc_3$ = *NIL* = $O_2.oc_3$ (see step (iii) of Example 5). However, following Algorithm GFT in our axiom-based approach, this error will be exposed by another fundamental pair *new*.*push*(8).*top* ~ 8 induced from axiom a_6 in Example 1, since *new*.*push*(8).*top* = *NIL* ≠ 8.

3.3.3 *Limitation* (1): *Infinite Cycles*

If a DRG contains cycles, the set of relevant observable contexts is infinite. We can, however, only choose a finite subset as test cases. Thus some program faults may remain undetected. This is an inherent limitation of program testing. To select such a finite subset, step (d) of Algorithm DOE uses a positive integer t_i or T to control the number of iterations of the cycles. The determination of t_i or T remains a difficult problem. In the current phase, these integers are supplied manually by user. Alternatively, we may consider the feasibility of adding further heuristics to the algorithm. For instance, in step (d)(2) of Example 5, we may find that the required number t_i of iterations of the cycle l_i = -*totalPush*-(arc_8-*totalPush*)^{t_i}- is closely related with the number 3 in the branch condition of the method *top*, which can be identified in the labels of the output arcs of the node *totalPush*.

3.3.4 *Limitation* (2): *Fault Masking*

A new concern may be raised on our relevant observable context technique. If an observable context *oc* itself contains an error, can we determine whether $O_1 \approx O_2$? Let $u_1 \sim u_2$ be two equivalent ground terms and s_1 and s_2 be their corresponding method sequences in an implementation. There are four possible cases:

(a) There exists some error in s_1 or s_2 such that $\neg(O_1 \approx O_2)$:
   (1) The error in *oc* does not affect the decision whether $O_1 \approx O_2$. In this case, our procedure finds that $\neg(O_1 \approx O_2)$ and reports an error.
   (2) The error in *oc* causes an erroneous decision on the observational equivalence of O_1 and O_2. In this case, our procedure finds that $O_1 \approx O_2$ and does not report any error.

(b) There is no error in s_1 and s_2, and hence we should have $O_1 \approx O_2$:
   (1) The error in *oc* does not affect the decision whether $O_1 \approx O_2$. In this case, our procedure finds that $O_1 \approx O_2$ and does not report any error.
   (2) The error in *oc* causes an erroneous decision whether $O_1 \approx O_2$. In this case, our procedure finds that $\neg(O_1 \approx O_2)$ and reports an error. In spite of the erroneous decision, the error report is actually correct because there is an implementation error in *oc*.

It is well-known that program testing does not necessarily guarantee correctness [30, 31]. It is generally considered *acceptable* that a test may not reveal all the errors in an implementation. If a test reports an implementation error, we say that the test is *useful*. It would be *unacceptable*, however, if a test reports an error that does not exist in an implementation.

In the above, the cases (a)(1) and (b)(2) are useful, while the cases (a)(2) and (b)(1) are acceptable. Hence, our approach does not produce unacceptable cases.

3.3.5 *Size of DRG*

The size of a DRG can be represented by a tuple (N, S), where N is the number of nodes in the DRG, and S is the number of segments. If the corresponding implementation contains D data members and M methods, and P is the maximum number of conditions in each method, then $N = D+1$, and $S \leq D^2 \times M \times P$. In the worst case, "*directly affects*" is a universal relation, which corresponds to $S = D^2 \times M \times P$. In fact, this worst case very seldomly occurs, if ever. We expect the DRG to be rather simple in most practical situations, since the DRG models the class level, which is a relatively low level in an object-oriented system. The number of nodes in the DRG of a class, equivalent to the number of data members in a given concrete class, is usually small, and "*directly affects*" is generally far from a universal relation. For conventional programming, many authors have supplied statistical data to show that simple program structures are used more often than complex structures [25, 32, 33, 34, 35]. Since the class level is relatively low in an object-oriented system, the situation is very similar. For example, we have analyzed statistically the source code of one of our projects entitled **FOOD** (**F**unctional **O**bject-**O**riented **D**esign) [36]. We have reviewed 16 classes and found that the average numbers of the data members and methods in each class were 6 and 8, respectively. We have also examined 21 classes in another case study on bank accounts and found that the average numbers of data members and methods in each class were 4 and 7, respectively.

3.3.6 *Executability of a Given Path for a Given Object*

Note that the concept of executability of a given path **for a given object** defined in Definition 3.4 is very different from the concept of feasibility of a path in other flow graph techniques [25]. An *infeasible path* is normally defined as a path whose conditions cannot be satisfied by **any** input value, and is well-known to be undecidable. However, since executable and unexecutable paths defined in Definition 3.4 **are related to some object** O_1, they can be determined from the known values $O_1.d_i$, that is, the values of the data members of the given object O_1. Thus, unlike the concept of feasibility, the executability of a given path for a given object as defined in this paper is decidable.

3.3.7 *Complexity of Traversing Executable Paths*

Referring to Algorithm DOE and Example 5, let L be the maximum length of all acyclic paths from any node to the node *observed*. Let n be the maximum number of Boolean conditions in the output arcs of any node that are *true* for the current values of $O_1.d_i$ and $O_2.d_i$. Let T be the ceiling supplied by the user for the number of iterations of cycles. Since the maximum number of selective branches at any node in a path is $n(T+1)$, and the longest path contains L nodes, the maximum number of executable paths is $(n(T+1))^L$. We note that, for a given DRG of the class under test, n is a variable according to the different objects O_1 and O_2, but L and T are constants. Hence, the complexity of traversing executable paths is $O(n^L)$, in the worst case.

Furthermore, by the same reasoning as that of Section 3.3.5, we do not expect the constant L to be large in most practical situations.

3.4 An Implementation of Algorithm DOE

As indicated in Section 2.5.4, Algorithm GFT is analogous to that used in the tools described in [9]. Their implementations are also similar, so that we have not included it in the present phase of our prototyping study. Instead, we have focused our attention on the implementation of Algorithm DOE, and implemented a prototype on a Pentium/120. In summary, it is a reformed C++ interpreter, constructed by embedding Algorithm DOE into a C++ interpreter. The prototype consists of five modules: *parser.c*, *drg.c*, *pigeonC.h*, *subLib.c*, and *pigeonC.c*. The modules *pigeonC.h* and *subLib.c* contain the definitions of the main data structures and the interfaces to internal library functions, respectively. The module *parser.c* includes a lexical analyzer and a recursive descent parser. It also performs the initialization for *drg.c* and other modules. The module *drg.c* constructs the DRG, traverses executable paths by backtracking, and generates and executes the corresponding relevant observable contexts for two given objects. Finally, *pigeonC.c* serves as the main module of the prototype. It reads the *C++* program code for a given class under test, allocates memory for the program, prescans it, and calls and coordinates other modules to perform the task.

Note that Algorithm DOE as specified in this paper shows only an abstract summary. It is, in fact, refined into several sub-algorithms that call many other functions, as described in Chen et al. [37]. Readers may find the following additional notes useful:

3.4.1 *Pigeon C++*

The prototype has been implemented using Borland *C++*. It requires the program for a given class under test is written in a subset of *C++* language. We call this subset *Pigeon C++*, which is an extension of *Little C* [38]. In the present phase, *Pigeon C++* contains the following features of *C++* language:

parameterized or non-parameterized functions with local variables; recursion; *if* statements; *do-while*, *while*, and *for* statements; *return* statements; integer, character, and array variables; instance variables of classes; global variables; string constants; some standard library functions; member functions of classes; +, –, *, /, %, <, >, <=, >=, ==, ! = , unary –, unary +; and comments. As a limitation, *Pigeon C++* does not contain pointers in the present phase. The implementation of the relevant observable context technique with pointers is much more complex and needs further investigation.

3.4.2 *Construction and Traversal of a DRG*

Suppose an implemented class contains k methods $m_1, m_2, ..., m_k$. Let d_1 and d_2 be two data members in the implemented class. The following is a schematic summary of the tasks required for each method m_i in the construction of the DRG:

(1) Scan the code of m_i.

(2) Suppose c denotes a constant and p denotes a predicate. When a statement of the form "$d_1 = c$" or "$d_1 = f(..., d_2, ...)$" is found, put the *arc_label* ($true$, m_i) into the table of arc labels, and put the segment [c or index of d_2, index of *arc_label* ($true$, m_i), index of d_1] into the list of output arcs. When a statement such as "if (p) {...; $d_1 = c$ or $f(..., d_2, ...)$; ...}" is found, put the *arc_label* (p, m_i) into the table of arc labels, and put the segment [c or index of d_2, index of *arc_label* (p, m_i), index of d_1] into the list of output arcs. If $p = p(..., d_3, ...)$, d_3 is a data member different from d_2, we should also put the segment [index of d_3, index of *arc_label* (p, m_i), index of d_1] into the list of output arcs. If the statement also contain "*else* {...; $d_4 = c_0$ or $g(..., d_5, ...)$; ...}", we must also put the *arc_label* ($\neg p$, m_i) into the table of arc labels, and put the segment [c_0 or index of d_5, index of *arc_label* ($\neg p$, m_i), index of d_4] into the list of output arcs.

For every such statement, the time for performing this task is bounded. Assume that it is no more than T_1. If the method m_i contains s_{i1} such statements, the time for handling these statements is no more than $T_1 \bullet s_{i1}$.

(3) Skip the other statements in the method m_i. The time for skipping such a statement is also bounded. Assume that it is no more than T_2. Since the method m_i contains $(s_i - s_{i1})$ such statements, the time for handling these statements is no more than $T_2(s_i - s_{i1})$.

Thus, the time t for constructing the DRG of the class satisfies the following formula:

$$t \le \Sigma_{i=1, ..., k}(T_1 \bullet s_{i1} + T_2(s_i - s_{i1})) \le \Sigma_{i=1, ..., k}(T \bullet s_{i1} + T(s_i - s_{i1})) = \Sigma_{i=1, ..., k}(T \bullet s_i) = T \bullet s$$

where $T = \max\{T_1, T_2\}$ and $s = s_1 + s_2 + ... + s_k$. Hence, the time t for constructing the DRG is $O(s)$.

When traversing the executable paths, if backtracking is necessary, the algorithm traverses the observer arcs first, followed by the acyclic output arcs, and finally the output arcs for cycles.

Further implementation details, such as the internal representations and the actual procedures for the construction and traversal of the DRG, can be found in Chen et al. [37].

3.4.3 *Interactions with Users*

The prototype is a semi-automatic tool. In the present phase, it requires the users to supply the following information manually:

(1) Two equivalent method sequences corresponding to a selected fundamental pair of equivalent ground terms for the given class under test.

(2) A list of methods in the class that are observers.

(3) In step (d) of Algorithm DOE, when there are uninstantiated input variables in the *oc* just obtained, we should apply the PDP technique to select values for the input variables. This selection may be semi-automatic, but is only manual at present.

(4) In the traversal of an executable path in step (d), if a cycle l_i is encountered, the user should supply a ceiling t_i for the number of iterations of l_i, or determine a global ceiling T allowed by the system for the number of iterations of any cycle.

If Algorithm GFT is implemented, (1) can be semi-automatic. Even then, we shall reserve the manual interface as a supplement.

3.4.4 *Empirical Results*

We have experimented with Examples 4 and 5 on the prototype. The experimental result of Example 5 (on *Pentium*/120), as shown in the following table, is the same as predicted.

Global ceiling supplied by the user for the number of iterations of any cycle	Number of observable contexts generated by the prototype	Number of error-revealing observable contexts	Total run time for all observable contexts	Run time for the first observable context that reports the error
0	2	0	0.093407 s	–
1	6	0	0.164835 s	–
2	12	3	0.283516 s	0.108791 s
3	20	5	0.437363 s	0.107692 s
4	30	5	0.634066 s	0.129670 s

Here, the run time includes the time for generating the two objects under test, traversing the executable paths for the objects in the DRG, constructing observable contexts from the executable paths, executing the observable contexts, and reporting the detected error, if any. However, it does not include the time for constructing the DRG. In Example 5, the time for constructing the DRG is 0.043297 seconds.

Suppose that the user indicates a global ceiling of 3 iterations. From the above table, we know there are 20 observable contexts to be generated in total. The run time for executing all of them would be 0.437363 seconds. In fact, Algorithm DOE does not test all the 20 observable contexts. When the first observable context reporting an error is encountered, the algorithm will exit, ignoring other observable contexts. Hence, the actual run time is found to be 0.107692 seconds.

Some trouble was encountered in the experiment on Example 4, since "*#define SIZE* 100" was too large for generating test sets of reasonable sizes. The allowable maximum size of array in the experiment on this example is 50. After changing the *SIZE* to 50, the experiment succeeds in reporting the error, but the run time on *Pentium*/120 is 13.571429 seconds.

We also wrote a correct *C++* program for the specification in Example 1, embedded common errors into the program, such as writing *height* > 0 as *height* < 0 or *height* > 2, and then experimented with them on the prototype. The experimental results showed that all such common errors could be exposed by the prototype. As an illustration, the empirical results for the erroneous implementation with *height* > 2 are listed as follows:

Global ceiling supplied by the user for the number of iterations of any cycle	Number of observable contexts generated by the prototype	Number of error-revealing observable contexts	Total run time for all observable contexts	Run time for the first observable context that reports the error
0	5	3	0.092308 s	0.041758 s
1	14	6	0.226374 s	0.041758 s
2	24	7	0.404396 s	0.041758 s
3	34	7	0.602198 s	0.041758 s
4	44	7	0.836264 s	0.041758 s

The time for constructing the DRG of this example is 0.024835 seconds.

4. RELATED WORK

There are two ways to use algebraic specifications in software testing. One was originally presented by Jalote [39], and extended by Frankl and Doong [11, 22, 23]. The other was initially proposed by Gannon et al. [40], and extended by Gaudel and others [9, 10, 21].

The former considers the axioms as rewriting rules, suggests to choose test cases from all legal combinations of operations (or terms), and derives their equivalent terms by means of the rewriting rules. The latter selects test cases from "the set of ground instances of the axioms obtained by replacing each variable by all ground terms of the right sort" under well-defined hypotheses [10]. Our Theorem 2 reveals an essential relationship between these two approaches.

Our approach is motivated by the ASTOOT black-box approach of Frankl and Doong and the testing theory of Gaudel and others. For completeness, we shall also compare our approach with the white-box dataflow-based approach of Parrish and others.

4.1 The Work of Frankl and Doong

In general, there are a number of advantages in Frankl and Doong's functional approach [11, 22, 23] to test object-oriented programs. Using algebraic specifications, it helps to solve the oracle problem. By taking sequences of operations as test cases, instead of individual operations, this approach does not depend on a predefined calling method but on a suite of messages passing among objects. This concept is especially suitable for object-oriented programming. It can support an integrated set of semi-automatic tools covering test case generation, test driver generation, test execution, and test checking.

Our approach hopefully inherits the above advantages. However, there are a few substantial distinctions between Doong and Frankl's approach and ours:

(1) Frankl and Doong define two terms u_1 and u_2 to be equivalent "if we can use the axioms as rewrite rules to transform u_1 to u_2" [11]. There is a problem in this definition. Consider, for example, two terms *new*.*push*(1).*push*(2).*pop* and *new*.*push*(3).*pop*.*push*(1) for the specification of the class of integer stacks in Example 1. They produce observationally equivalent results when implemented correctly. However, they cannot be transformed from one to the other by left-to-right rewriting rules, and hence are not equivalent according to this definition. As a result, Frankl and Doong's approach cannot derive this kind of test cases. Our Definition 2.6 avoids this problem. However, as a supplement, Doong and Frankl's approach allows the user to add manually generated test cases that may remedy such kinds of situations.

(2) Doong and Frankl's approach requires the user to select a finite number of original (operation) sequences from the set of terms, which is infinite in general. They offer the following tentative guidelines to guide the selection and generation of test cases: "Use (at least some) long original sequences, with a variety of relative frequencies of different constructors and transformers" and "If the specification has conditional axioms (with comparison operators), choose a variety of test cases for each original sequence, with various parameters chosen over a large range. Equivalently, choose a variety of different paths through the ADT tree arising from each original sequence." [11] These guidelines are supported by two empirical case studies.

The selection domain of our "fundamental pair" strategy is much smaller than that of the set of equivalent ground terms, but the coverage of testing fundamental pairs remains the same. Using our strategy, two of Doong and Frankl's tools, the compiler and simplifier, can be replaced by a generator that induces fundamental pairs as test cases directly from the two sides of each axiom. Our strategy is based on mathematical theorems, thus more precise.

(3) Frankl and Doong give an "approximate check" [11] for object observational equivalence, known as the EQN method, consisting of two techniques. One produces a recursive version of *eqn* from specification. The other is at the "implementation level". The former requires the analyst to supply a special axiom *eqn* to the specification of each class to define equivalence of two objects in the class. Different *eqn* axioms are attached to different classes. The approach also requires the designer and programmer to implement a special recursive method *eqn* for the respective *eqn* axiom in each class. If one of the attached axioms for *eqn*, or its implementation, is problematic, then the test report may show an error even if the original class is implemented correctly. Having said that, if we consider the *eqn* function to be a part of the class under test, the above situation is acceptable. The technique at the "implementation level" suggests to use white-box information about how the data type is represented and manipulated in the implementation. "For example, knowing that a FIFO queue is represented as a linked list, one can traverse the two lists comparing the elements". If the

corresponding elements of the two lists are equal, we can indeed conclude that the two queues are observationally equivalent. If some corresponding elements of the two lists are not equal, however, we cannot immediately conclude that the two queues are observationally not equivalent, since the object-oriented paradigm allows encapsulation and the hiding of internal information. As discussed by Frankl and Doong, there are some advantages and disadvantages of the two techniques. As an option, our relevant observable context technique checks observational equivalence of objects using a different idea, which concentrates on checking relevant observable contexts only, skipping irrelevant observable contexts.

4.2 The Work of Gaudel, Bernot, Bouge, Choquet, Dauchy, Fribourg, and Marre

Bernot, Gaudel, and Marre [21] have proposed a general theory for software testing based on algebraic specifications. This theory includes several hypotheses such as a regularity hypothesis and a uniformity hypothesis for selecting test cases, and some oracle hypotheses to constrain the oracle problem. These hypotheses are important from a theoretical point of view, because they formalize common test practices and express the gap between testing and correctness proving. In our approach, we combine our strategy with the regularity hypothesis and the decomposition technique of uniform subdomains to select a finite set of fundamental pairs as test cases.

Furthermore, the oracle hypothesis and the related counterexample in [21] have inspired us to propose the relevant observable context technique.

An important distinction between our approach and the work of Gaudel and others is that the latter replaces all the variables in the axioms by ground terms according to the regularity hypothesis [10], whereas our approach replaces them by normal forms according to Theorem 2. The benefit of replacing variables by normal forms, rather than by general ground terms, has been described in Section 2.5.4.

4.3 The Work of Parrish, Borie, and Cordes

Parrish, Borie, and Cordes [15] proposed a white-box dataflow-based approach to testing classes. Their approach uses a *class graph*, which is a collection of <*N*, *E*, *D*, *U*, *I*>, where *N* is the set of nodes, *E* is the set of edges. A node represents an operation. An edge from a node *A* to a node *B* means that it is permissible to invoke the operation *B* after the operation *A*. *D* denotes the set of *definitions of data*. *U* denotes the set of *uses of data*. *I* refers to the set of infeasible subpaths. *N*, *E*, *D*, and *U* are obtained from the class interface in the implementation. The purpose of introducing the concept of *I* is to allow us to eliminate sequences that are inappropriate or impossible to test. For this purpose, the authors set up a *weak class graph* and the corresponding *weak coverage criteria*, and added two further restrictions. They then proved that a minimum-length sequence of operations which satisfies *weak node coverage criteria*, *weak branch coverage criteria*, *weak definition coverage criteria*, or *weak use coverage criteria* could be found in polynomial time. Hence, the approach can be automated efficiently. However, as admitted by the authors, these weak criteria and the two additional restrictions substantially weaken the degree of testing demanded, and reduce the significance of their results.

We can compare Parrish's class graph approach and our DRG approach as follows:

(a) Both of these two approaches are white-box techniques.

(b) The class graph approach only deals with syntax problems. However, the DRG approach is used to determine whether two given objects are observational equivalent, which is a semantics problem.

(c) In a class graph, a node represents an operation and an edge (op_1, op_2) denotes that the concatenation of two operations, $op_1.op_2$, is legal in syntax. On the other hand, in a DRG, a node represents a data member and an arc (d_1, d_2) denotes that the data member d_1 directly affects data member d_2.

(d) In DRGs, the counterpart to the weak branch coverage criteria in the class graph approach ensures each cycle is traversed only once. In general, this is very insufficient for the purpose of deciding whether two given objects are observationally equivalent. See Example 5 and the table in Section 3.4.4.

(e) As indicated by [15], the class graph approach can also be based on the syntax section of algebraic specifications. Hence, this approach can be considered as an optional technique to select normal forms in step (a) of Algorithm GFT without a choice on the positive integer k.

5. CONCLUSION

In this paper, we define a fundamental pair as a pair of equivalent ground terms formed by replacing all the variables on both sides of an axiom by normal forms. We prove that a complete implementation of a canonical specification is consistent with respect to all equivalent ground terms if and only if it is consistent with respect to all fundamental pairs. In other words, the testing coverage of fundamental pairs as good as that of all equivalent ground terms, and hence we need only concentrate on the testing of fundamental pairs. Our strategy is based on mathematical theorems. Combining this strategy with the regularity hypothesis and the decomposition technique of uniform subdomains, we construct an algorithm for selecting a finite set of fundamental pairs as test cases.

On the other hand, we prove that the observational equivalence of objects cannot be determined using a finite set of observable contexts derived from any black-box technique. Instead, we propose a relevant observable context approach, which is a heuristic white-box technique, and have implemented a prototype for it.

Many authors have indicated that program testing cannot thoroughly expose all the errors in the program under consideration [30, 31]. In this sense, testing in general is an incomplete and undecidable problem. Our approach cannot avoid this inherent limitation of testing. We decompose the testing problem into several sub-tasks, separate the decidable sub-tasks from the undecidable or difficult ones, and put them into a unified methodological framework via two algorithms. The undecidable or difficult sub-tasks are analyzed separately.

As future work, we shall investigate into the selection of nonequivalent terms as test cases, and the testing of interactions among groups of cooperating classes at the *cluster level*. We shall also consider the following problems: Is it feasible to abstract heuristic information from program code to alleviate the problems of deciding on the length of normal forms in step (a) of Algorithm GFT, and determining the number of iterations of cycles in step (d) of Algorithm DOE? How do we extend *Pigeon C++* and its implementation with the relevant observable context technique to include aliasing and pointers? How do

we use compiler techniques instead of interpreter techniques to improve the efficiency of the prototype? How can we make the prototype more practical and user-friendly?

ACKNOWLEDGMENTS

We are grateful to Professor P.G. Frankl for her helpful discussions via electronic mail. We also greatly appreciate the anonymous referees for their invaluable comments, questions, and suggestions, which lead our paper to a greater depth. Special thanks are due to Mr. Yue Tang Deng, the first author's M.Eng. student in Jinan University, for his invaluable contributions to the implementation and experimentation of the prototype.

REFERENCES

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

2. Fujiwara, S., von Bochmann, G., Khendek, F., Amalou, M., and Ghedamsi, A., Test selection based on finite state models, *IEEE Transactions on Software Engineering 17*, 6 (1991), 591–603.

3. Harrold, M.J., McGregor, J.D., and Fitzpatrick, K.J., Incremental testing of object-oriented class structures, In *Proceedings of 14th IEEE International Conference on Software Engineering* (*ICSE '92*), IEEE Computer Society, Los Alamitos, CA (1992), 68–80.

4. Kung, D., Gao, J., Hsia, P., Toyoshima, Y., and Chen, C., A test strategy for object-oriented programs, In *Proceedings of IEEE 19th Computer Software and Applications Conference* (*COMPSAC '95*), IEEE Computer Society, Los Alamitos, CA (1995), 239–244.

5. Kung, D., Gao, J., Hsia, P., Lin, J., and Toyoshima, Y., Design recovery for software testing of object-oriented programs, In *Proceedings of IEEE Working Conference on Reverse Engineering*, IEEE Computer Society, Los Alamitos, CA (1993), 202–211.

6. Perry, D.E. and Kaiser, G.E., Adequate testing and object-oriented programming, *Journal of Object-Oriented Programming 3*, 5 (1990), 13–19.

7. Turner, C.D. and Robson, D.J., A state-based approach to the testing of class-based programs, *Software: Concepts and Tools 16*, 3 (1995), 106–112.

8. Chen, H.Y. and Tse, T.H., Towards a new methodology for object-oriented software testing at the class and cluster levels, Technical Report TR-97-07, Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong (1997).

9. Bouge, L., Choquet, N., Fribourg, L., and Gaudel, M.-C., Test sets generation from algebraic specifications using logic programming, *Journal of Systems and Software 6* (1986), 343–360.

10. Dauchy, P., Gaudel, M.-C., and Marre, B., Using algebraic specification in software testing: a case study on the software of an automatic subway, *Journal of Systems and Software 21*, 3 (1993), 229–244.

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

12. Chen, T.Y. and Low, C.K, Dynamic data flow analysis for C++, In *Proceedings of 1995 Asia-Pacific Software Engineering Conference* (*APSEC' 95*), IEEE Computer Society, Los Alamitos, CA (1995), 22–28.

13. Chen, T.Y. and Low, C.K, Error detection in C++ through dynamic data flow analysis, *Software: Concepts and Tools 18*, 1 (1997), 1–13.

14. Fiedler, S.P., Object-oriented unit testing, *Hewlett-Packard Journal 40*, 4 (1989), 69–74.

15. Parrish, A.S., Borie, R.B., and Cordes, D.W., Automated flow graph-based testing of object-oriented software modules, *Journal of Systems and Software 23*, 2 (1993), 95–109.

16. 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).

17. 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).

18. Goguen, J.A. and Diaconescu, R., Towards an algebraic semantics for the object paradigm, In *Recent Trends in Data Type Specification: Proceedings of 9th International Workshop on Specification of Abstract Data Types*, H. Ehrig and F. Orejas, Eds. *Lecture Notes in Computer Science, Vol. 785*. Springer-Verlag, Berlin, Germany (1994), 1–29.

19. Goguen, J.A. and Meseguer, J., Unifying functional, object-oriented, and relational programming with logical semantics, In *Research Directions in Object-Oriented Programming*, B. Shriver and P. Wegner, Eds. MIT Press, Cambridge, MA (1987), 417–477.

20. Wolfram, D.A. and Goguen, J.A., A sheaf semantics for FOOPS expressions, In *Object-Based Concurrent Programming: Proceedings of ECOOP '91 Workshop*, M. Tokoro, O.M. Nierstrasz, and P. Wegner, Eds. *Lecture Notes in Computer Science, Vol. 612*, Springer-Verlag, Berlin, Germany (1992), 81–98.

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

22. Doong, R.-K. and Frankl, P.G., Case studies on testing object-oriented programs, In *Proceedings of 4th ACM Annual Symposium on Testing, Analysis, and Verification* (*TAV 4*), ACM, New York, NY (1991), 165–177.

23. Frankl, P.G. and Doong, R.-K., Tools for testing object-oriented programs, In *Proceedings of 8th Pacific Northwest Conference on Software Quality* (1990), 309–324.

24. Frankl, P.G., Private communication (1994).

25. White, L.J. and Cohen, E.I., A domain strategy for computer program testing, *IEEE Transactions on Software Engineering SE-6*, 3 (1980), 247–257.

26. Jeng, B. and Weyuker, E.J., *A simplified domain-testing strategy*, *ACM Transactions on Software Engineering and Methodology 3*, 3 (1994), 254–270.

27. 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, Brooklin, New York, NY (1991).

28. Breu, R. and Breu, M., Abstract and concrete objects: an algebraic design method for object-based systems, In *Algebraic Methodology and Software Technology: Proceedings of 3rd International Conference* (*AMAST '93*), M. Nivat, C. Rattray, T. Rus, and G. Scollo, Eds. Workshops in Computing, Springer-Verlag, Berlin, Germany (1993), 343–348.

29. Morell, L.J., A theory of fault-based testing, *IEEE Transactions on Software Engineering 16, 8* (1990), 844–857.

30. Clarke, L.A., A system to generate test data and symbolically execute programs., *IEEE Transactions on Software Engineering SE-2*, 3 (1976), 215–222.

31. Miller, E.F., Notes on the philosophy of testing, In *Proceedings of 1st Annual International Computer Software and Applications Conference (COMPSAC '77)*, IEEE Computer Society, New York, NY (1977).

32. Knuth, D.E., An empirical study of FORTRAN programs, *Software: Practice and Experience 1* (1971), 105–133.

33. Elshoff, J.L., A numerical profile of commercial PL/I programs, Report GMR-1927, Computer Science Department, General Motors Research Laboratory, Warren, MI (1975).

34. Elshoff, J.L., An analysis of some commercial PL/I programs, *IEEE Transactions on Software Engineering SE-2* (1976), 208–215.

35. Cohen, E.I., A Finite Domain-Testing Strategy for Computer Program Testing, PhD Dissertation, Ohio State University, Columbus, OH (1978).

36. Tse, T.H. and Goguen, J.A., Functional object-oriented design (FOOD), In *Foundations of Information Systems Specification and Design*, Dagstuhl Seminar Report No. 35, H.-D. Ehrich, A. Sernadas, and J.A. Goguen, Eds. International Conference and Research Center for Computer Science, Wadern, Germany (1992).

37. Chen, H.Y., Deng, Y.T., and Tse, T.H., ROCS: an object-oriented software testing system at the class level based on the relevant observable context technique, Technical Report TR-97-08, Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong (1997).

38. Schildt, H., *The Craft of C: Take-Charge Programming*, Osborne McGraw-Hill, Berkeley, CA (1992).

39. Jalote, P., Specification and testing of abstract data types, In *Proceedings of 7th Annual International Computer Software and Applications Conference* (*COMPSAC '83*), IEEE Computer Society, New York, NY (1983), 508–511.

40. Gannon, J.D., McMullin, P.R., and Hamlet, R., Data-abstraction implementation, specification, and testing, *ACM Transactions on Programming Languages and Systems 3*, 3 (1981), 211–223.

AUTHORS' BIOGRAPHIES

Huo Yan Chen is a full Professor at Jinan University in China. He has also performed research in software engineering and expert systems at the University of Hong Kong for more than six years. He had conducted research in logic programming and knowledge engineering at the University of Illinois at Urbana-Champaign for two years. He had also served in an electronic company in Guangzhou as a computer systems analyst and software designer for ten years. His current research interests are in software engineering and knowledge engineering, including formal methods, object-oriented methodology, software testing, logic programming, expert systems, and discrete mathematics.

Professor Chen obtained his BS degree in Mathematical Science from Nankai University of China in 1968 and his M.Eng. degree in Computer Science from the National University of Defence Science and Technology in 1982. He has received the "Governmental Special Allowance Monthly for Outstanding Contributions to the Nation" from the State Council of China since 1992. He has also been awarded three Prizes for Science and Technology Achievements by Guangdong Provincial Government.

T.H. Tse is a Professor in Computer Science at the University of Hong Kong. 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 had been a Council Member of the Vocational Training Council for eight years.

Dr. Tse received his PhD from the University of London. He was a Visiting Fellow of the University of Oxford in 1990 and 1992, and an Academic Exchange Visitor of the University of Melbourne in 1996. He has been selected for a Ten Outstanding Young Persons' Award, a Key of Success Award, and a Twentieth Century Award for Achievement. He has been decorated with an MBE by the Queen.

F.T. Chan received the BSc, MPhil, and MBA degrees from the University of Hong Kong. He is an Associate Professor of the School of Professional and Continuing Education, the University of Hong Kong. He is responsible for developing and organizing continuing and professional courses in computer science and information science for adult learners. He had worked in the Computer Centre of the same university for eleven years before being an academic.

He is a member of the British Computer Society, the Hong Kong Computer Society, and the Hong Kong Institution of Engineers. He is a Chartered Engineer of the Engineering Council in the UK. His research interests are in software engineering, expert systems, and adult continuing education.

T.Y. Chen received his BSc and MPhil from the University of Hong Kong, his M.Sc. from the Imperial College of Science and Technology, and his PhD from the University of Melbourne. He taught at the University of Hong Kong and is currently with the University of Melbourne. His main research interests include software testing and software engineering.