

# TACCLE: A Methodology for Object-Oriented Software Testing at the Class and Cluster Levels

HUO YAN CHEN

Jinan University

T. H. TSE

The University of Hong Kong

and

T. Y. CHEN

Swinburne University of Technology

---

Object-oriented programming consists of several different levels of abstraction, namely, the algorithmic level, class level, cluster level, and system level. The testing of object-oriented software at the algorithmic and system levels is similar to conventional program testing. Testing at the class and cluster levels poses new challenges. Since methods and objects may interact with one another with unforeseen combinations and invocations, they are much more complex to simulate and test than the hierarchy of functional calls in conventional programs. In this paper, we propose a methodology for object-oriented software testing at the class and cluster levels. In class-level testing, it is essential to determine whether objects produced from the execution of implemented systems would preserve the properties defined by the specification, such as behavioral equivalence and nonequivalence. Our class-level testing methodology addresses both of these aspects. For the testing of behavioral equivalence, we propose to select fundamental pairs of equivalent ground terms as test cases using a black-box technique based on algebraic specifications, and then determine by means of a white-box technique whether the objects resulting from executing such test cases are observationally equivalent. To address the testing of behavioral nonequivalence, we have identified and analyzed several nontrivial problems in the current literature. We propose to classify term equivalence into four types,

---

Huo Yan Chen is supported in part by the National Natural Science Foundation of China under Grant No. 69873020 and the Guangdong Province Science Foundation under Grants #980690 and #950618. T. H. Tse is supported in part by the Hong Kong Research Grants Council and the University Research Committee of the University of Hong Kong. T. Y. Chen is supported in part by the Hong Kong Research Grants Council.

Authors' addresses: H. Y. Chen, Department of Computer Science, Jinan University, Guangzhou 510632, China; email: tchy@jnu.edu.cn; Corresponding author: T. H. Tse, Department of Computer Science and Information Systems, The University of Hong Kong, Pokfulam Road, Hong Kong; email: tse@csis.hku.hk; T. Y. Chen, School of Information Technology, Swinburne University of Technology, Hawthorn, Victoria 3122, Australia; email: tychen@it.swin.edu.au.

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.

© 2001 ACM 1049-331X/01/0100-0056 \$5.00

thereby setting up new concepts and deriving important properties. Based on these results, we propose an approach to deal with the problems in the generation of nonequivalent ground terms as test cases. Relatively little research has contributed to cluster-level testing. In this paper, we also discuss black-box testing at the cluster level. We illustrate the feasibility of using Contract, a formal specification language for the behavioral dependencies and interactions among cooperating objects of different classes in a given cluster. We propose an approach to test the interactions among different classes using every individual message-passing rule in the given Contract specification. We also present an approach to examine the interactions among composite message-passing sequences. We have developed four testing tools to support our methodology.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications—*Languages*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools* (e.g., data generators, coverage testing); D.3.2 [**Programming Languages**]: Language Classifications—*Object-oriented languages*

General Terms: Languages, Reliability

Additional Key Words and Phrases: Algebraic specifications, Contract specifications, object-oriented programming, software testing, message passing

## 1. INTRODUCTION

Object-oriented systems contain four different levels of abstraction. They are the algorithmic level, class level, cluster level, and system level. The algorithmic level considers the code for each operation in a class. The class level is composed of the interactions of methods and data that are encapsulated within a given class. The cluster level consists of the interactions among cooperating classes, which are grouped to accomplish some tasks. The system level is composed of all the clusters [Smith and Robson 1992].

Testing at the algorithmic and system levels is similar to conventional program testing. Most research workers concentrate themselves on class-level testing [Doong and Frankl 1991; 1994; Fiedler 1989; Frankl and Doong 1990; Kung et al. 1994; Smith and Robson 1992; Turner and Robson 1993a; 1993b; 1995]. Relatively little study has been made on cluster-level testing or its relationship with class-level testing. In this paper, we present a unified methodology TACCLE, for object-oriented software **Testing At the Class and Custer LEvels**. This methodology is based on type signature specifications, including algebraic specifications for classes and Contract specifications for clusters. The complete methodology consists of three components: using fundamental pairs of equivalent ground terms as class-level test cases and a relevant observable context technique to determine the observational equivalence of objects; using nonequivalent ground terms as further class-level test cases; and using sequences of message-passing expressions and postconditions as cluster-level test suites. These three components are closely related and supplement one another. For example, the relevant observable context technique for determining the observational equivalence of objects in the first component will be invoked by the second and third components.

We have improved on the ASTOOT approach of Doong and Frankl [1994] by using equivalent ground terms in algebraic specifications as class-level test cases. We deploy fundamental pairs of equivalent ground terms as class-level test cases. This has been reported in detail in our companion paper [Chen et al. 1998] and is summarized as the first part of our comprehensive methodology in this paper.

Besides the proposal to consider equivalent ground terms as test cases, another important contribution of Doong and Frankl [1994] is the identification of a need to use “non-equivalent” ground terms as test cases. They assert that if two ground terms are nonequivalent, but their corresponding implemented method sequences produce observationally equivalent objects, then there is an error in the implementation. Furthermore, they present an approach to generate nonequivalent test cases from equivalent test cases by “exchang[ing] the path conditions.” In this paper, we illustrate that there are nontrivial problems in Doong and Frankl’s assertion and approach on nonequivalent ground terms as test cases. In order to solve these problems, we classify the relations among terms into four different types, namely, rewriting relations, normal equivalence, observational equivalence, and attributive equivalence. We investigate the relationships among them. Based on these results, we propose a new approach to generate nonequivalent ground terms as test cases using state-transition diagrams.

At the cluster level, some recent research has been devoted to the test orders among different classes [Jorgensen and Erickson 1994; Kung et al. 1995]. Our concern in this paper is to trace the relationships and interactions among different classes in a cluster. Relationships among different classes in a cluster can be divided into two types: vertical inheritance and horizontal interactions. Testing problems on inheritance has been investigated by Harrold et al. [1992]. We will concentrate on the testing problems related to horizontal interactions among classes in a cluster. Consider, for example, a banking system containing two different classes *SavingAccount* and *CheckAccount*. An operation *transferTo* transfers money from a *SavingAccount* to a *CheckAccount*. This is a horizontal interaction between the two classes.

We illustrate that neither algebraic specifications nor interface specifications are sufficient for specifying message passing and other interactions among cooperating classes for the purpose of cluster-level testing. It is therefore necessary to use another formal specification technique. We find that Contract specifications proposed by Helm et al. [1990] are suitable for this purpose. Our scheme for cluster-level testing consists of two parts. The first part tests the interactions among different classes in the cluster according to every individual message-passing rule in the given Contract specification. The other part reviews the interactions according to composite message-passing sequences.

Four testing tools have been developed to support our methodology TACCLE. An interactive tool DOE supports the determination of object observational equivalence. An automatic tool GCS generates composite

message-passing sequences from Contract specifications. The extraction and composition of message-passing sequences from a program implementing the cluster are supported by automatic tools ESI and GCS, respectively. An interactive tool GAN supports the generation of attributively non-equivalent terms as test cases.

The organization of this paper is as follows: Section 2 gives the basic concepts on algebraic specifications used in class-level testing. In Section 3, we outline our integrated approach to use fundamental pairs of equivalent ground terms as class-level test cases and to use the relevant observable context technique to determine the observational equivalence of the resulting objects. Section 4 addresses the topic of generating nonequivalent ground terms as class-level test cases. Section 5 gives the basic concepts on Contract specifications used in cluster-level testing. Sections 6 and 7 show how to solve the problems on cluster-level testing using Contract specifications. In Section 8, we discuss briefly the open issues and future work. Section 9 concludes the paper.

## 2. ALGEBRAIC SPECIFICATIONS

As indicated by Clarke and Wing [1996], “One current trend is to integrate different specification languages, each able to handle a different aspect of a system.” In order to facilitate the generation of test cases in a black-box approach, we propose to use formal specifications, including algebraic specifications [Breu 1991; Goguen and Meseguer 1987] for classes, and Contract specifications [Helm et al. 1990] for clusters. Both algebraic specifications and Contract specifications are based on type signatures. Hence, they have a common theoretical basis. In our methodology, we select fundamental pairs of equivalent ground terms and pairs of nonequivalent ground terms as class-level test cases according to algebraic specifications, and select cluster-level test suites according to Contract specifications. In this section, we present some basic concepts on algebraic specification. The concepts of Contract specifications will be given in Section 5.

An *algebraic specification* for a class is composed of a syntax declaration and a semantic specification. The syntax declaration lists the *operations* involved, as well as their domains and codomains, corresponding to the input parameters and output of the operations. The semantic specification consists of equational *axioms* that describe the behavioral properties of the operations.

*Example 1 (Algebraic Specification of the Class of Integer Stacks).*

```

module INTEGER-STACK
include INTEGER
class Stack
imported classes Integer Boolean
operations
  new :  $\rightarrow$  Stack
  .isEmpty : Stack  $\rightarrow$  Boolean
  .push(_) : Stack Integer  $\rightarrow$  Stack
  .pop : Stack  $\rightarrow$  Stack
  .top : Stack  $\rightarrow$  Integer  $\cup$  {nil}

```

**variables** $S : Stack$  $N : Integer$ **axioms** $a_1: new.isEmpty = true$  $a_2: S.push(N).isEmpty = false$  $a_3: new.pop = new$  $a_4: S.push(N).pop = S$  $a_5: S.top = nil \text{ if } S.isEmpty$  $a_6: S.push(N).top = N$ *End of Example 1*

Intuitively, a *term* is a sequence of operations in an algebraic specification. For example,

$$new.push(10).push(20).pop$$

is a term in the class of integer stacks above. A term without variables is called a *ground term*. In this paper, we only consider ground terms because, during dynamic testing, test cases involve actual data rather than structural or symbolic manipulation.

If a subterm within a ground term is unified against the left-hand side of an equational axiom and substituted by the right-hand side of the axiom, we say that the ground term is *transformed* into another using the axiom as progressive *left-to-right rewriting rules*. A ground term is in *normal form* if and only if it cannot be further transformed by any axiom in the specification. For example,  $new.push(10).push(20)$  is in normal form, but  $new.push(10).push(20).pop$  is not, since the latter can be transformed by axiom  $a_4$  into  $new.push(10)$ .

An algebraic 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. We will limit ourselves only to canonical specifications in this paper. Please refer to Section 4.6 for a discussion on our basic assumptions.

In a given class  $C$ , operations or methods that return the values of the attributes of the objects in  $C$  are called *observers* of  $C$ . Operations or methods that return initial objects of  $C$  are called *creators* of  $C$ . Operations or methods that transform the states of objects in  $C$  are called *constructors* or *transformers* of  $C$ . The current *state* of an object is the combination of current values of all attributes of this object. When a constructor or transformer acts on an object, it changes the value of at least one attribute of the object. The difference between a constructor and a transformer is that a transformer may be eliminated from a term by applying rewriting rules, but a constructor may not. In Example 1, for instance, the operation  $new$  is a creator,  $_.push(N)$  is a constructor,  $_.pop$  is a transformer, and  $_.isEmpty$  and  $_.top$  are observers.

An *observable context* on a class  $C$  is a sequence of constructors or transformers of  $C$  (possibly an empty sequence) followed by an observer of  $C$ .

For example,  $push(100).push(200).pop.top$  is an observable context on the class *Stack*. The observer *top* is also regarded as an observable context on *Stack*.

A *primitive type* in the specification of a class *C* is a type imported into *C* at the lowest level of the hierarchy of imports. Examples are *Integer* or *Boolean*. Typically, they do not need to be defined specifically, do not import further classes or types, have no observers, and can be mapped directly to the built-in types of most implementation languages.

An implementation of a given canonical specification is said to be *complete* if and only if every operation in the specification is implemented by one and only one method in the program; every imported class in the specification is implemented by one and only one imported class in the program; and every primitive type in the specification is implemented either by a type built in the implementation language, or by a type that has been fully tested and deemed to be correct. Without loss of generality, we will assume in this paper that both an operation in the specification and the corresponding method in the implementation bear the same name.

*Definition 1 (Observational Equivalence of Object).* Given a canonical specification and an implementation of a class *C*, two objects  $O_1$  and  $O_2$  are said to be **observationally equivalent** (denoted by " $O_1 \approx_{obs} O_2$ ") if and only if the following condition is satisfied:

If no observable context *oc* on *C* is applicable to  $O_1$  and  $O_2$ , then  $O_1$  and  $O_2$  are identical objects. Otherwise, for any such *oc* on *C*,  $O_1.oc$  and  $O_2.oc$  are observationally equivalent objects.

### 3. CLASS-LEVEL TESTING USING FUNDAMENTAL EQUIVALENT PAIRS

The first phase of our TACCLE methodology covers the use of fundamental pairs of equivalent ground terms as class-level test cases and the use of a "relevant observable context" technique to determine the observational equivalence of the resulting objects. We will present only a summary of this phase in the current section because the full details have been published in our companion paper [Chen et al. 1998].

In this section, for a given canonical specification of a class, two ground terms are said to be *equivalent* if and only if they can be transformed into the same normal form by some axioms as left-to-right rewriting rules. An implementation is said to be *consistent with respect to* two equivalent ground terms if and only if the method sequences corresponding to these two ground terms produce observationally equivalent objects. Obviously, if an implementation is not consistent with respect to two equivalent ground terms, then there is some error in this implementation. This assertion is the basis of selecting equivalent ground terms as class-level test cases. Doong and Frankl [1994] proposed the ASTOOT approach to test object-oriented programs. They recommended heuristic guidelines on the use of equivalent ground terms as class-level test cases.

We define the concept of 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. Obviously, the set of fundamental pairs is a proper subset of the set of equivalent ground terms. We prove that a complete implementation of a canonical specification 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 use of fundamental pairs as test cases covers the use of equivalent ground terms for the same purpose, and hence we need only concentrate on the testing of fundamental pairs. Our strategy is based on mathematical theorems. Based on the strategy, we propose a GFT algorithm for **Generating a Finite set of fundamental pairs as Test cases**.

Given a pair of equivalent ground terms as a test case, we should then determine whether the objects that result from executing the implemented program are observationally equivalent. We have proved, however, that the observational equivalence of objects cannot be determined using a finite set of observable contexts derived from any black-box technique [Chen et al. 1998]. Hence, we supplement our approach with a “relevant observable context” technique, which is a white-box technique, to determine observational equivalence. This task is performed by a DOE algorithm for **Determining the Observational Equivalence of objects**.

Like any other testing method, the DOE algorithm cannot guarantee that all implementation errors will be revealed by a finite set of test cases. The effectiveness and limitations of the algorithm are discussed in Section 3.3 of Chen et al. [1998] and will not be repeated in this paper.

We have implemented a prototype of the interactive tool DOE to support the construction of a Data member Relevance Graph (DRG), traversing executable paths in the DRG, generating and executing relevant observable contexts, determining object observational equivalence, and reporting detected errors, if any. Some experimental results on the prototype are given in Chen et al. [1998].

#### 4. CLASS-LEVEL TESTING USING NONEQUIVALENT TERMS

The second phase of our TACCLE methodology consists of class-level testing using nonequivalent ground terms as test cases. As indicated by Doong and Frankl [1994], testing on nonequivalent ground terms is significant. Even if an implementation is consistent with respect to all equivalent ground terms, it may contain an error that results in a pair of nonequivalent ground terms being erroneously implemented as equivalent. In Section 4.1, we outline the related work of Doong and Frankl, and analyze some nontrivial problems in it. In Section 4.2, we classify the concept of term equivalence into different types and highlight their subsumption relationships. Section 4.3 discusses some fundamental properties on the use of nonequivalent terms as test cases. Based on these properties, we present in Section 4.4 an approach to generate nonequivalent ground terms as test cases using state-transition diagrams. Section 4.5 discusses how to determine whether a test case of nonequivalent ground terms reveals an error.

#### 4.1 Related Work and Analysis of Problems

The concept of equivalent terms has been applied to testing [Bernot et al. 1991; Bouge et al. 1986; Chen et al. 1998; Doong and Frankl 1991; 1994; Frankl and Doong 1990]. In particular, Doong and Frankl [1991; 1994] and Frankl and Doong [1990] defined the concept of equivalent terms as follows:

*Definition 2.* Two terms  $u_1$  and  $u_2$  in a given specification are said to be equivalent if we can use the axioms in the specification as rewrite rules to transform  $u_1$  into  $u_2$ .

Based on Definition 2, they proposed a framework for testing as follows:

- Consider the set  $U$  consisting of all 3-tuples  $(S_1, S_2, tag)$ , where  $S_1$  and  $S_2$  are sequences of messages and  $tag$  is “equivalent” if  $S_1$  is equivalent to  $S_2$  according to the specification, and is “not-equivalent” otherwise.
- [Suppose  $O_1$  and  $O_2$  are identical or equivalent objects of a class  $C$ .] For each element of  $U$ , send message-passing sequences  $S_1$  and  $S_2$  to the objects  $O_1$  and  $O_2$ , respectively. Then check whether the returned object of  $O_1$  is observationally equivalent to the returned object of  $O_2$ .
- If all the observational equivalence checks agree with the tags, then the implementation is correct. Otherwise, it is incorrect.

The following assertions are implicit in the above framework:

*Assertion 1.* Let  $u_1$  and  $u_2$  be two ground terms in a given specification and  $s_1$  and  $s_2$  be their corresponding method sequences in an implementation of the specification. If  $u_1$  is equivalent to  $u_2$ , but  $s_1$  and  $s_2$  produce observationally nonequivalent objects, then the implementation is incorrect.

*Assertion 2.* If  $u_1$  is not equivalent to  $u_2$ , but  $s_1$  and  $s_2$  produce observationally equivalent objects, then the implementation is incorrect.

These two assertions formed the theoretical basis for generating equivalent and nonequivalent ground terms as class-level test cases. Doong and Frankl [1994] further indicated that the testing of nonequivalent ground terms has significant ramifications. Even the exhaustive testing of equivalent ground terms may fail to detect an error that results in two different states being confused as a single state. As an extreme example, consider a problematic implementation in which none of the operations changes the states of objects. In this case, any two equivalent ground terms will return the same observational result. Thus, the error will not be detected by only testing equivalent terms. The testing of nonequivalent ground terms is therefore necessary and cannot be ignored.

In general, the contributions of Doong and Frankl [1994] are valuable. There are, however, a couple of nontrivial problems.

4.1.1 *Problem 1.* Assertion 2 does not hold in the context of Definition 2. Consider the following example:

*Example 2.* Let  $u_1 = \text{new.push}(10).\text{push}(20).\text{pop}$  and  $u_2 = \text{new.push}(30).\text{pop.push}(10)$  for the specification of the class of integer stacks in Example 1. According to Definition 2, the terms  $u_1$  and  $u_2$  are nonequivalent, since they cannot be transformed from one into the other by the axioms in Example 1 as left-to-right rewriting rules. However, they produce observationally equivalent objects when the implementation is correct. This contradicts Assertion 2. *End of Example 2.*

We shall discuss how to deal with this problem in Sections 4.2 and 4.3.

4.1.2 *Problem 2.* Doong and Frankl [1994] also presented an approach to generate nonequivalent test cases from equivalent test cases by “exchang[ing] the path conditions.” They illustrated their approach by the following example:

*Example 3 (Algebraic Specification for the Class of Priority Queues of Integers).*

```

module PRIORITY-QUEUE
include INTEGER
class IntegerQueue
imported classes Integer Boolean
operations
  new :  $\rightarrow$  IntegerQueue
   $\_.$ isEmpty : IntegerQueue  $\rightarrow$  Boolean
   $\_.$ largest : IntegerQueue  $\rightarrow$  Integer  $\cup$   $\{-\infty\}$ 
   $\_.$ add( $\_$ ) : IntegerQueue Integer  $\rightarrow$  IntegerQueue
   $\_.$ delete : IntegerQueue  $\rightarrow$  IntegerQueue
  // Delete the largest element in the queue
variables
  Q : IntegerQueue
  N : Integer
axioms
   $a_1$  : new.isEmpty = true
   $a_2$  : Q.add(N).isEmpty = false
   $a_3$  : new.largest =  $-\infty$ 
   $a_4$  : Q.add(N).largest = N           if  $N > Q.largest$ ,
                                     Q.largest otherwise
   $a_5$  : new.delete = new
   $a_6$  : Q.add(N).delete = Q           if  $N > Q.largest$ ,
                                     Q.delete.add(N) otherwise

```

The test case  $(\text{new.add}(M).\text{add}(N).\text{delete}, \text{new.add}(M), \text{equivalent})$  with the path condition “ $N > M$ ” can be derived from the axioms above. By exchanging the path conditions, Doong and Frankl [1994] obtained the following test case:

$(\text{new.add}(M).\text{add}(N).\text{delete}, \text{new.add}(M), \text{non-equivalent})$   
 under the condition “ $N \leq M$ .”

*End of Example 3.*

In fact, this test case is erroneous because, according to axiom  $a_6$ , the two terms should be equivalent when  $N = M$ . This is exactly one of the problems that a tester should set out to test.

Furthermore, we have constructed the following example to show that this kind of error may even occur throughout the entire input domain, rather than only at some isolated boundary values.<sup>1</sup>

*Example 4 (Algebraic Specification for the Class of Priority Queues of Real Numbers).*

```

module REAL-QUEUE
include REAL
class RealQueue
imported classes Real Boolean
operations
  new :  $\rightarrow$  RealQueue
   $\_.$ isEmpty : RealQueue  $\rightarrow$  Boolean
   $\_.$ largest : RealQueue  $\rightarrow$  Real  $\cup$   $\{-\infty\}$ 
   $\_.$ smallest : RealQueue  $\rightarrow$  Real  $\cup$   $\{+\infty\}$ 
   $\_.$ add( $\_.$ ) : RealQueue Real  $\rightarrow$  RealQueue
   $\_.$ deleteLargest : RealQueue  $\rightarrow$  RealQueue
   $\_.$ deleteSmallest : RealQueue  $\rightarrow$  RealQueue
variables
  Q : RealQueue
  X : Real
axioms
   $a_1$  : new.isEmpty = true
   $a_2$  : Q.add(X).isEmpty = false
   $a_3$  : new.largest =  $-\infty$ 
   $a_4$  : new.smallest =  $+\infty$ 
   $a_5$  : Q.add(N).largest = X           if X > Q.largest,
                                     Q.largest otherwise
   $a_6$  : Q.add(X).smallest = X        if X < Q.smallest,
                                     Q.smallest otherwise
   $a_7$  : new.deleteLargest = new
   $a_8$  : new.deleteSmallest = new
   $a_9$  : Q.add(X).deleteLargest =
                                     Q           if X > Q.largest,
                                     Q.deleteLargest.add(N) otherwise
   $a_{10}$  : Q.add(X).deleteSmallest =
                                     Q           if X < Q.smallest,
                                     Q.deleteSmallest.add(X) otherwise

```

Using the axioms above, we can select a test case

$(new.add(X).add(Y).add((X + Y)/2).deleteLargest.deleteSmallest,$   
 $new.add((X + Y)/2), equivalent)$  under the path condition “ $Y > X$ .”

By exchanging the path conditions, we obtain a second test case

<sup>1</sup>In order to appreciate the main idea behind this example, readers are suggested to note that  $\min\{X, Y\} \leq (X + Y)/2 \leq \max\{X, Y\}$  regardless of whether “ $Y > X$ ” or “ $Y \leq X$ .” Hence, any exchange of the path conditions will not turn a pair of equivalent terms into nonequivalent terms.

$(new.add(X).add(Y).add((X + Y)/2).deleteLargest.deleteSmallest,$   
 $new.add((X + Y)/2), equivalent)$  under the condition “ $Y \leq X$ .”

The second test case is erroneous because, using the axioms above as left-to-right rewriting rules, we can actually prove that these two terms are equivalent whenever  $Y \leq X$ ! *End of Example 4.*

Hence, it is erroneous to generate nonequivalence from equivalence by “exchang[ing] the path conditions” [Doong and Frankl 1994]. We shall present a better approach to generate nonequivalent terms as test cases in Section 4.4.

## 4.2 A Classification of Equivalence

To solve Problem 1 in Section 4.1.1, we must review and revise the definition of equivalent terms.

We note that the relation among terms defined in Definition 4.1 is not symmetric, and hence it is not really an equivalence relation. We shall call it a *rewriting relation* instead. Thus, Definition 4.1 will be replaced by the following:

*Definition 3 (Rewriting Relation of Terms).* Two terms  $u_1$  and  $u_2$  in a given specification are said to satisfy a **rewriting relation** (denoted by “ $u_1 \rightarrow^* u_2$ ”) if and only if  $u_1$  can be transformed into  $u_2$  using the axioms in the specification as rewrite rules.

Let us consider the following attempt to improve the definition of equivalence:

*Definition 4 (Normal Equivalence of Terms).* Given a canonical specification of a class, two ground terms  $u_1$  and  $u_2$  are said to be **normally equivalent** (denoted by “ $u_1 \sim_{nor} u_2$ ”) if and only if both of them can be transformed into the same normal form by some axioms as left-to-right rewriting rules.

Definition 4 is obviously weaker than Definition 3. We indicated in Section 4.1.1 that Example 2 contravenes Assertion 2 in the context of Definition 2 (and hence Definition 3). Does this example contravene Assertion 2 in the context of the relaxed Definition 4?

According to Definition 4, the terms

$$u_1 = new.push(10).push(20).pop$$

and

$$u_2 = new.push(30).pop.push(10)$$

in Example 2 are equivalent, since they can be transformed into the same normal form  $new.push(10)$  by the axioms in Example 1 as left-to-right rewriting rules. Hence, this example does not contravene Assertion 2 in the context of Definition 4.

Unfortunately, Assertion 2 still does not hold in the context of Definition 4.2. This can be illustrated by the following example:

*Example 5 (Algebraic Specification of the Class of Bank Accounts).*

```

module ACCOUNT
include MONEY
class Account
imported classes Money String
operations
  overdrawn :  $\rightarrow$  money
  new( $\_$ ) : String  $\rightarrow$  Account
   $\_.$ name : Account  $\rightarrow$  String
   $\_.$ addr : Account  $\rightarrow$  String // addr means address
   $\_.$ bal : Account  $\rightarrow$  Money // bal means balance
   $\_.$ setAddr( $\_$ ) : Account String  $\rightarrow$  Account
  // setAddr means setting the value of the address
   $\_.$ credit( $\_$ ) : Account Money  $\rightarrow$  Account
   $\_.$ debit( $\_$ ) : Account Money  $\rightarrow$  Account
variables
  S : String
  A : Account
  M : Money
axioms
  a1 : new(S).name = S
  a2 : new(S).addr = nil
  a3 : new(S).bal = 0
  a4 : A.credit(M).bal = A.bal + M
  a5 : A.debit(M).bal = A.bal - M if A.bal  $\geq$  M
  a6 : A.debit(M).bal = overdrawn if A.bal < M
  a7 : A.setAddr(S).bal = A.bal
  a8 : A.credit(M).addr = A.addr
  a9 : A.debit(M).addr = A.addr
  a10 : A.setAddr(S).addr = S
  a11 : A.credit(M).name = A.name
  a12 : A.debit(M).name = A.name
  a13 : A.setAddr(S).name = A.name

```

Consider the terms

$$u_1 = \text{new}('John').\text{setAddr}('2 \text{ University Drive}').\text{credit}(1000).\text{debit}(200)$$

and

$$u_2 = \text{new}('John').\text{setAddr}('2 \text{ University Drive}').\text{credit}(800).$$

According to Definition 4,  $u_1$  and  $u_2$  are nonequivalent, since they cannot be transformed into the same normal form by the above axioms as left-to-right rewriting rules. However, they produce observationally equivalent objects when the implementation is correct. This also contradicts Assertion 2. *End of Example 5.*

Examples 2 and 5 illustrate that a more fundamental understanding of term equivalence is vital before Problem 1 in Section 4.1.1 can be solved. We would like to investigate carefully different degrees of term equivalence and the relationships among them.

We shall define other degrees of equivalence using the recursive definitions 9 and 10 below. In order to do so, we must explain some related concepts first.

*Definition 5 (Input and Output Classes).* Given an operation

$$_ .f( _ . _ \cdot \cdot \cdot _ ) : C C_1 C_2 \cdot \cdot \cdot C_n \rightarrow D,$$

$C$  is called the input class of  $f$ , and  $D$  is called the output class of  $f$ .

*Definition 6 (Applicability).* Given an algebraic specification of a class  $C$ , let  $u = f_0.f_1.\cdot\cdot\cdot.f_i$  and  $v = g_0.g_1.\cdot\cdot\cdot.g_j$  be sequences of operation(s). We say that  $v$  is **applicable to**  $u$  if and only if the output class of  $f_i$  is the same as the input class of  $g_0$ .<sup>2</sup>

We would like to add that a ground term in a given class  $C$  might contain operations in its imported classes. Consider, for instance, the class *Account* in Example 5. If we take  $A = \text{JohnAccount}$  and  $M = 8000$ , Axiom  $a_4$  produces a ground term  $\text{JohnAccount.bal} + 8000$ , which contains an operation “+” in the imported class *Money* of the class *Account*. Furthermore, let  $C'$  be an imported class of  $C$ . For consistency and the ease of description, any observer of  $C'$  will also be regarded as an observer of  $C$ , and any observable context on  $C'$  will also be regarded as an observable context on  $C$ .

The concepts of operations, observers, and observable contexts due to imported classes can be further illustrated in the example below.

*Example 6 (Algebraic Specification for the Class of Stacks of Integer-Bags).*

```

module INTEGER-BAG
include INTEGER
class IntegerBag
imported classes Integer Boolean
operations
  newBag :  $\rightarrow$  IntegerBag
  .null : IntegerBag  $\rightarrow$  Boolean
  .largest : IntegerBag  $\rightarrow$  Integer  $\cup$   $\{-\infty\}$ 
  .add( ) : IntegerBag Integer  $\rightarrow$  IntegerBag
  .delete : IntegerBag  $\rightarrow$  IntegerBag
  // Delete the largest element in the Bag
variables
  B : IntegerBag
  N : Integer
axioms
  a1 : newBag.null = true
  a2 : B.add(N).null = false
  a3 : newBag.largest =  $-\infty$ 

```

<sup>2</sup>The concept of applicability is a special case of the concept of appropriateness defined by Goguen and Malcolm [2000].

$$a_4 : B.add(N).largest = N \quad \text{if } N > B.largest,$$

$$B.largest \quad \text{otherwise}$$

$$a_5 : newBag.delete = newBag$$

$$a_6 : B.add(N).delete = B \quad \text{if } N > B.largest,$$

$$B.delete.add(N) \quad \text{otherwise}$$

**module** STACK-OF-INTEGGER-BAGS

**include** INTEGER-BAG

**class** Stack

**imported classes** Boolean IntegerBag

**operations**

*newStack* :  $\rightarrow$  Stack  
*.isEmpty* : Stack  $\rightarrow$  Boolean  
*.top* : Stack  $\rightarrow$  IntegerBag  $\cup$  {nil}  
*.push(\_)* : Stack IntegerBag  $\rightarrow$  Stack  
*.pop* : Stack  $\rightarrow$  Stack

**variables**

*S* : Stack  
*NB* : IntegerBag

**axioms**

$a_1 : newStack.isEmpty = true$   
 $a_2 : S.push(NB).isEmpty = false$   
 $a_3 : newStack.pop = newStack$   
 $a_4 : S.push(NB).pop = S$   
 $a_5 : S.top = nil$  if  $S.isEmpty$   
 $a_6 : S.push(NB).top = NB$

- (a) The following are some ground terms in the class *Stack*:

*newStack.pop.push(newBag.add(5).add(4).delete)*  
*newStack.pop.push(newBag.add(5).add(4).delete).top*  
*newStack.pop.push(newBag.add(5).add(4).delete).top*  
*.add(3).add(2).delete*  
*newStack.pop.push(newBag.add(5).add(4).delete).top*  
*.add(3).add(2).delete.largest*  
*newStack.pop.push(newBag.add(5).add(4).delete).top*  
*.add(3).add(2).delete.largest + 6*

- (b) Not only are *isEmpty* and *top* observers of the class *Stack*. We observe from (a) that imported operators such as *largest* are also observers of *Stack*.

- (c) Similarly, not only are operation sequences like

$$oc_1 = push(newBag.add(5).add(4).delete).top$$

observable contexts on the class *Stack*. Imported operation sequences such as

$$oc_2 = add(3).add(2).delete.largest$$

are also observable contexts on *Stack*.

(d) Consider a ground term

$$u_1 = \text{newStack.pop.push}(\text{newBag.add}(8)).$$

The input class of the observable context  $oc_1$  is *Stack*, while that of  $oc_2$  is *IntegerBag*. Thus, the observable context  $oc_1$  is applicable to  $u_1$ , but  $oc_2$  is not.

(e) Consider another ground term

$$u_2 = \text{newStack.pop.push}(\text{newBag.add}(5).\text{add}(4).\text{delete}).\text{top}.$$

The input class of the observer *top* is *Stack*, while its output class is *IntegerBag*. The input class of the observer *largest* is *IntegerBag*, while its output class is *Integer*. Thus, the observer *largest* is applicable to  $u_2$ , but *top* is not.

*End of Example 6.*

In general, an algebraic specification of a class  $C$  may import other classes  $C_1, C_2, \dots, C_n$  as the output classes of its observers. An imported class  $C_i$  may be a primitive type, or may recursively import other classes  $C_{i_1}, C_{i_2}, \dots, C_{i_m}$  as output classes of its observers, and so on, until all the final imported classes are primitive types.

*Definition 7 (Observable Context Sequence).* Given a specification of a class  $C$ , an operation sequence of the form  $oc_1.oc_2.\dots.oc_n$  is called an **observable context sequence** or an **oc sequence** on  $C$  if and only if every  $oc_i$  ( $i = 1, 2, \dots, n$ ) is an observable context on  $C$  and every  $oc_j$  ( $j = 2, 3, \dots, n$ ) is applicable to  $oc_{j-1}$ . The **length** of this oc sequence is said to be  $n$ . If the output class of  $oc_n$  is a primitive type, then  $oc_1.oc_2.\dots.oc_n$  is called a **primitive oc sequence** on  $C$ .

In observation (c) of Example 6, for instance,  $oc_1.oc_2$  is an oc sequence, but  $oc_2.oc_1$  is not. Furthermore,  $oc_1.oc_2$  is a primitive oc sequence.

*Definition 8 (Proper Imports).* An algebraic specification of a class  $C$  is said to have **proper imports** if every oc sequence on  $C$  is of finite length and can be extended to a primitive oc sequence in a finite number of steps.<sup>3</sup> Otherwise, it is said to have **improper imports**.

Consider, for instance, the specification in Example 6. The class *Stack* imports *Boolean* and *IntegerBag* as output classes of its observers *isEmpty* and *top*, respectively. *Boolean* is a primitive type. *IntegerBag*

<sup>3</sup>More formally, given a specification  $SP$ , let  $SC$  be the finite set of classes in  $SP$ . We define a **class-import relation** of  $SP$  as the binary relation  $R = \{(C, C') \in SC \times SC \mid C' \text{ is an imported class of } C\}$ . Let  $R^*$  be the transitive closure of  $R$ . A specification  $SP$  has **proper imports** if (a)  $R^*$  is nonreflexive and (b) for any nonprimitive class  $C \in SC$ , there exists some primitive type  $C' \in SC$  such that  $(C, C') \in R^*$ .

imports the primitive types *Boolean* and *Integer* as output classes of its observers *null* and *largest*, respectively. This specification does not contain any infinite oc sequence and hence has proper imports. In this paper, we shall only consider specifications with proper imports.

We are now ready to define other degrees of term equivalence.

*Definition 9 (Observational Equivalence of Terms).* Given a canonical specification of a class  $C$  with proper imports, two ground terms  $u_1$  and  $u_2$  are said to be **observationally equivalent** (denoted by “ $u_1 \sim_{obs} u_2$ ”) if and only if the following condition is satisfied:

If no observable context  $oc$  on  $C$  is applicable to  $u_1$  and  $u_2$ , then the normal forms of  $u_1$  and  $u_2$  are identical. Otherwise, for any such  $oc$  on  $C$ ,  $u_1.oc$  and  $u_2.oc$  are observationally equivalent.

*Definition 10 (Attributive Equivalence of Terms).* Given a canonical specification of a class  $C$  with proper imports, two ground terms  $u_1$  and  $u_2$  in  $C$  are said to be **attributively equivalent** (denoted by “ $u_1 \sim_{att} u_2$ ”) if and only if the following condition is satisfied:

If no observer  $ob$  of  $C$  is applicable to  $u_1$  and  $u_2$ , then the normal forms of  $u_1$  and  $u_2$  are identical. Otherwise, for any such  $ob$  in  $C$ ,  $u_1.ob$  and  $u_2.ob$  are observationally equivalent.

Corresponding to Definition 9 for the observational equivalence of terms, we have Definition 2 for objects as shown in Section 2. Corresponding to Definition 10 for the attributive equivalence of terms, we have the following definition for objects.

*Definition 11 (Attributive Equivalence of Objects).* Given an implementation of a canonical specification of a class  $C$  with proper imports, two objects  $O_1$  and  $O_2$  in  $C$  are said to be **attributively equivalent** (denoted by “ $O_1 \approx_{att} O_2$ ”) if and only if the following condition is satisfied:

If no observer  $ob$  of  $C$  is applicable to  $O_1$  and  $O_2$ , then  $O_1$  and  $O_2$  are identical objects. Otherwise, for any such  $ob$  in  $C$ ,  $O_1.ob$  and  $O_2.ob$  are observationally equivalent objects.

The base cases of the recursions in Definitions 1, 9, 10, and 11 are the identity of primitive types, because there is no observer or observational context in such types. Since the given specification has proper imports and the implementation is complete, the recursions always terminate with finite numbers of steps. For example, the following lemmas can be derived directly from Definitions 9, 10, and 11, respectively. They will be useful in later theorems.

**LEMMA 1.** *Given a canonical specification of a class  $C$  with proper imports, two ground terms  $u_1$  and  $u_2$  are observationally equivalent if and only if the following condition is satisfied:*

If no observable context  $oc$  on  $C$  is applicable to  $u_1$  and  $u_2$ , then the normal forms of  $u_1$  and  $u_2$  are identical. Otherwise, for any primitive  $oc$  sequence  $oc_1.oc_2.\dots.oc_k$  on  $C$  applicable to  $u_1$  and  $u_2$ , the normal forms of  $u_1.oc_1.oc_2.\dots.oc_k$  and  $u_2.oc_1.oc_2.\dots.oc_k$  are identical.

LEMMA 2. Given a canonical specification of a class  $C$  with proper imports, two ground terms  $u_1$  and  $u_2$  are attributively equivalent if and only if the following condition is satisfied:

If no observer  $ob$  of  $C$  is applicable to  $u_1$  and  $u_2$ , then the normal forms of  $u_1$  and  $u_2$  are identical. Otherwise, for any primitive  $oc$  sequence  $ob.oc_1.oc_2.\dots.oc_k$  in  $C$  applicable to  $u_1$  and  $u_2$ , where  $ob$  is some observer of  $C$ , the normal forms of  $u_1.ob.oc_1.oc_2.\dots.oc_k$  and  $u_2.ob.oc_1.oc_2.\dots.oc_k$  are identical.

LEMMA 3. Given an implementation of a canonical specification of a class  $C$  with proper imports, two objects  $O_1$  and  $O_2$  are attributively equivalent if and only if the following condition is satisfied:

If no observer  $ob$  of  $C$  is applicable to  $O_1$  and  $O_2$ , then  $O_1$  and  $O_2$  are identical objects. Otherwise, for any primitive  $oc$  sequence  $ob.oc_1.oc_2.\dots.oc_k$  in  $C$  applicable to  $O_1$  and  $O_2$ , where  $ob$  is some observer of  $C$ , the executions of  $O_1.ob.oc_1.oc_2.\dots.oc_k$  and  $O_2.ob.oc_1.oc_2.\dots.oc_k$  result in identical objects.

LEMMA 4. Given a canonical specification of a class  $C$  with proper imports, let  $u$  be any ground term. No observable context  $oc$  on  $C$  is applicable to  $u$  if and only if no observer  $ob$  of  $C$  is applicable to  $u$ .

PROOF. An observer of  $C$  applicable to  $u$  is a special case of an observable context on  $C$  applicable to  $u$ . Conversely, suppose an observable context  $oc$  is applicable to  $C$ . It must be of the form  $op_0.op_1.\dots.op_n.ob$ , where each  $op_i$  ( $i = 0, 1, \dots, n$ ) is a constructor or transformer, and where  $ob$  is an observer. Now, the input and output classes of a constructor or transformer must be the same, and the input class of  $op_i$  must be the same as the output class of  $op_{i-1}$ . Hence, the input class of  $ob$  must be the same as the input class of  $op_0$ . Thus, the observer  $ob$  must be applicable to  $u$ .  $\square$

The relations among terms or objects defined in Definitions 1, 4, 9, 10, and 11 are reflexive, symmetric, and transitive. Hence, they are equivalence relations. The following theorems show the subsumption relationships among them:

**THEOREM 1 (SUBSUMPTION RELATIONSHIPS FOR TERM EQUIVALENCE).** Given a canonical specification of a class with proper imports,

- (a) *If two ground terms  $u_1$  and  $u_2$  satisfy the rewriting relation, then  $u_1$  and  $u_2$  must be normally equivalent, but the converse is not true.*
- (b) *If two ground terms  $u_1$  and  $u_2$  are normally equivalent, then  $u_1$  and  $u_2$  must be observationally equivalent, but the converse is not true.*
- (c) *If two ground terms  $u_1$  and  $u_2$  are observationally equivalent, then  $u_1$  and  $u_2$  must be attributively equivalent, but the converse is not true.*

PROOF.

- (a) Since  $u_1$  and  $u_2$  satisfy the rewriting relation,  $u_1$  can be transformed into  $u_2$  using the axioms in the specification as rewrite rules. Since the specification is canonical,  $u_2$  should have a unique normal form  $u^*$ . By the property of canonical specifications,  $u^*$  should also be the unique normal form of  $u_1$ . Thus,  $u_1$  and  $u_2$  have the same normal form  $u^*$ , and hence are normally equivalent.

Conversely, consider

$$u_1 = \text{new.push}(10).\text{push}(20).\text{pop}$$

and

$$u_2 = \text{new.push}(30).\text{push}(10)$$

in Example 1. They are normally equivalent, but do not satisfy the rewriting relation.

- (b) If  $u_1$  and  $u_2$  are normally equivalent and if no observable context on the class is applicable to them, then by Definition 4.2,  $u_1$  and  $u_2$  must be observationally equivalent.

Suppose some observable context on the class is applicable to  $u_1$  and  $u_2$ . If  $u_1$  and  $u_2$  are normally equivalent, they can be rewritten into the same normal form  $u^*$ . Hence, for any primitive oc sequence  $oc_1.oc_2.\dots.oc_k$  on  $C$  applicable to  $u_1$  and  $u_2$ , both  $u_1.oc_1.oc_2.\dots.oc_k$  and  $u_2.oc_1.oc_2.\dots.oc_k$  can be rewritten into  $u^*.oc_1.oc_2.\dots.oc_k$ . Furthermore,  $u^*.oc_1.oc_2.\dots.oc_k$  can be rewritten into a unique normal form  $u^{**}$ . Since the specification is canonical,  $u^{**}$  is also the common unique normal form of  $u_1.oc_1.oc_2.\dots.oc_k$  and  $u_2.oc_1.oc_2.\dots.oc_k$ . In other words, the normal forms of  $u_1.oc_1.oc_2.\dots.oc_k$  and  $u_2.oc_1.oc_2.\dots.oc_k$  are identical. Thus, according to Lemma 1,  $u_1$  and  $u_2$  are observationally equivalent.

Conversely, take

$$u_1 = \text{new}('John').\text{setAddr}('2 University Drive').\text{credit}(1000).\text{debit}(200)$$

and

$$u_2 = \text{new}('John').\text{setAddr}('2 \text{ University Drive}').\text{credit}(800)$$

in Example 5. They are observationally equivalent but not normally equivalent.

- (c) Let  $u_1$  and  $u_2$  be two observationally equivalent ground terms. If no observer of the class is applicable to them, by Lemma 4, no observational context on the class is applicable to them either. By Definition 9, therefore, the normal forms of  $u_1$  and  $u_2$  are identical. Thus, according to Definition 10,  $u_1$  and  $u_2$  are attributively equivalent.

Suppose some observer of the class is applicable to  $u_1$  and  $u_2$ . Any such observer is a special case of observable contexts on the class. Hence, according to Definition 9,  $u_1.ob$  and  $u_2.ob$  are observationally equivalent. By Definition 10, therefore,  $u_1$  and  $u_2$  are attributively equivalent. Conversely, take

$$u_1 = \text{new}.\text{push}(10).\text{push}(20)$$

and

$$u_2 = \text{new}.\text{push}(30).\text{push}(20)$$

in Example 1. They are attributively equivalent but not observationally equivalent.  $\square$

**THEOREM 2 (SUBSUMPTION RELATIONSHIPS FOR OBJECT EQUIVALENCE).** *Given a canonical specification of a class with proper imports, suppose the implementation is complete. If two objects  $O_1$  and  $O_2$  are observationally equivalent, then  $O_1$  and  $O_2$  must be attributively equivalent, but the converse is not true.*

**PROOF.** The proof is similar to that of Theorem 1(c).  $\square$

#### 4.3 A Theoretical Framework for the Testing of Equivalence and Nonequivalence

In this and the next two sections, we apply the above classification of equivalence to investigate the problems on nonequivalent terms as test cases.

By Theorem 1(b), two ground terms that are not normally equivalent may be observationally equivalent. Hence, their corresponding results in a correct implementation may be observationally equivalent. This is the reason why Assertion 2 does not hold in the context of normal equivalence. Similarly, we can explain why Assertion 2 does not hold in the context of rewriting relations.

We note from Theorem 1(b), however, that normal equivalence implies observational equivalence. Hence, Assertion 1 holds for normal equivalence. By similar reasoning, Assertion 1 also holds for rewriting relations. Thus, Assertion 1 under the concepts of rewriting relations and normal equivalence can be used as the theoretical basis of generating equivalent

ground terms as test cases, as done in Doong and Frankl [1994] and Chen et al. [1998], respectively.

Examples 2 and 5 indicate that rewriting relations and normal equivalence are too strong for Assertion 2. They should be replaced by weaker degrees of equivalence. Intuitively, the counterpart of the observational equivalence of objects in implementations should be the observational equivalence of ground terms in specifications, rather than the normal equivalence of ground terms. The following Definition 12 reflects this intuition.

*Notation 1.* Given any ground term  $u$ , we will use  $\Theta(u)$  to denote the object produced by the method sequence corresponding to  $u$ .

*Definition 12 (Consistent Implementation).* Given a canonical specification of a class with proper imports, suppose its implementation is complete. An implementation is **consistent with the specification** if and only if both of the following criteria are satisfied:

**Equivalence Criterion**

For any observationally equivalent ground terms  $u_1$  and  $u_2$ ,  $\Theta(u_1)$  and  $\Theta(u_2)$  must be observationally equivalent.

**Nonequivalence Criterion**

For any observationally nonequivalent ground terms  $u_1$  and  $u_2$ ,  $\Theta(u_1)$  and  $\Theta(u_2)$  must be observationally nonequivalent.

Although this formal definition is consistent with the intuitive concept of most software testers, it may not be immediately useful for practical situations. The observational equivalence of terms in a specification, for example, is fairly difficult to establish. Hence, in the first phase of the TACCLE project [Chen et al. 1998], we resorted to using an alternative equivalence criterion; namely, that “given any normally equivalent ground terms  $u_1$  and  $u_2$ ,  $\Theta(u_1)$  and  $\Theta(u_2)$  must be observationally equivalent.” How is this related to the formal definition above? Would the alternative criterion be too weak, so that some errors might not be revealed? We would like to establish a formal framework on the relationships among similar testing criteria. Our result turns out to be surprisingly neat and useful.

In order to establish the main theorems in the framework, we need a simple lemma first.

**LEMMA 5.** *Given an implementation of a canonical specification of a class  $C$  with proper imports,*

- (a) *Suppose  $u_1$  and  $u_2$  are two ground terms and  $v$  is a sequence of operations applicable to  $u_1$  and  $u_2$ . If  $u_1$  and  $u_2$  are observationally equivalent, then  $u_1.v$  and  $u_2.v$  are also observationally equivalent.*
- (b) *Suppose  $O_1$  and  $O_2$  are two objects, and  $s$  is a sequence of methods applicable to  $O_1$  and  $O_2$ . If  $O_1$  and  $O_2$  are observationally equivalent, then  $O_1.s$  and  $O_2.s$  are also observationally equivalent.*

PROOF.

- (a) If no observable context  $oc$  on  $C$  is applicable to  $u_1.v$  and  $u_2.v$ , then  $v$  must end with an observer. Hence,  $v$  itself is an observable context applicable to  $u_1$  and  $u_2$ . Thus, according to Definition 9, if  $u_1$  and  $u_2$  are observationally equivalent, then  $u_1.v$  and  $u_2.v$  are also observationally equivalent.

Suppose some observable contexts on the class are applicable to  $u_1.v$  and  $u_2.v$ . Let  $oc$  be any of such observable contexts. Then  $v.oc$  is also an observable context applicable to  $u_1$  and  $u_2$ . If  $u_1$  and  $u_2$  are observationally equivalent, by Definition 9,  $u_1.v.oc$  and  $u_2.v.oc$  are also observationally equivalent. Hence, by the same definition,  $u_1.v$  and  $u_2.v$  are observationally equivalent.

- (b) The proof for the implementation counterpart is similar.  $\square$

**THEOREM 3 (EQUIVALENCE CRITERIA).** *Given a canonical specification of a class with proper imports, suppose its implementation is complete. The following statements are equivalent:*

- (a) *For any observationally equivalent ground terms  $u_1$  and  $u_2$ ,  $\Theta(u_1)$  and  $\Theta(u_2)$  are observationally equivalent.*
- (b) *For any normally equivalent ground terms  $u_1$  and  $u_2$ ,  $\Theta(u_1)$  and  $\Theta(u_2)$  are observationally equivalent.*
- (c) *For any normally equivalent ground terms  $u_1$  and  $u_2$ ,  $\Theta(u_1)$  and  $\Theta(u_2)$  are attributively equivalent.*
- (d) *For any attributively equivalent ground terms  $u_1$  and  $u_2$ ,  $\Theta(u_1)$  and  $\Theta(u_2)$  are attributively equivalent.*
- (e) *For any observationally equivalent ground terms  $u_1$  and  $u_2$ ,  $\Theta(u_1)$  and  $\Theta(u_2)$  are attributively equivalent.*

PROOF.

**(a) implies (b):**

For any normally equivalent ground terms  $u_1$  and  $u_2$ , by Theorem 1(b),  $u_1$  and  $u_2$  must be observationally equivalent. Hence, if statement (a) is true,  $\Theta(u_1)$  and  $\Theta(u_2)$  must be observationally equivalent.

**(b) implies (c):**

For any normally equivalent ground terms  $u_1$  and  $u_2$ , if statement (b) is true,  $\Theta(u_1)$  and  $\Theta(u_2)$  must be observationally equivalent. By Theorem 2, therefore,  $\Theta(u_1)$  and  $\Theta(u_2)$  must be attributively equivalent.

**(c) implies (d):**

For any attributively equivalent ground terms  $u_1$  and  $u_2$ ,

- (i) If no observer of the class is applicable to  $u_1$  and  $u_2$ , then by

Definition 10, the normal forms of  $u_1$  and  $u_2$  are identical. In other words,  $u_1$  and  $u_2$  are normally equivalent. Hence, if statement (c) is true,  $\Theta(u_1)$  and  $\Theta(u_2)$  must be attributively equivalent.

- (ii) Suppose some observer of the class is applicable to  $u_1$  and  $u_2$ . Since the specification has proper imports, consider any primitive oc sequence  $ob.oc_1.oc_2.\dots.oc_k$  applicable to  $u_1$  and  $u_2$ , where  $ob$  is any observer applicable to  $u_1$  and  $u_2$ . Since  $u_1$  and  $u_2$  are attributively equivalent, by Lemma 2, the normal forms of  $u_1.ob.oc_1.\dots.oc_k$  and  $u_2.ob.oc_1.\dots.oc_k$  are identical. In other words,  $u_1.ob.oc_1.\dots.oc_k$  and  $u_2.ob.oc_1.\dots.oc_k$  are normally equivalent. Thus, if statement (c) is true, we have

$$\Theta(u_1.ob.oc_1.\dots.oc_k) \approx_{att} \Theta(u_2.ob.oc_1.\dots.oc_k).$$

Since the implementation is complete,

$$\Theta(u_1).ob.oc_1.\dots.oc_k \approx_{att} \Theta(u_2).ob.oc_1.\dots.oc_k.$$

Now, the objects resulting from the executions of  $\Theta(u_1).ob.oc_1.\dots.oc_k$  and  $\Theta(u_2).ob.oc_1.\dots.oc_k$  must be identical because  $ob.oc_1.oc_2.\dots.oc_k$  is a primitive oc sequence. Hence, by Lemma 3,  $\Theta(u_1)$  and  $\Theta(u_2)$  are attributively equivalent.

**(d) implies (e):**

For any observationally equivalent ground terms  $u_1$  and  $u_2$ , by Theorem 1(c),  $u_1$  and  $u_2$  must be attributively equivalent. Hence, if statement (d) is true,  $\Theta(u_1)$  and  $\Theta(u_2)$  must be attributively equivalent.

**(e) implies (a):**

For any observationally equivalent ground terms  $u_1$  and  $u_2$ ,

- (i) If no observable context on the class is applicable to  $u_1$  and  $u_2$ , by Lemma 4, no observer of the class is applicable to them either. Since  $u_1$  and  $u_2$  are observationally equivalent, if statement (e) is true,  $\Theta(u_1)$  and  $\Theta(u_2)$  must be attributively equivalent. Since no observer of the class is applicable, by Definition 4.2,  $\Theta(u_1)$  and  $\Theta(u_2)$  must be identical. Hence, according to Definition 2,  $\Theta(u_1)$  and  $\Theta(u_2)$  must be observationally equivalent.
- (ii) Suppose some observable contexts on the class are applicable to  $u_1$  and  $u_2$ . Let  $oc$  be any of such observable contexts. We can write  $oc = v.ob$  for some sequence of operations  $v$  (possibly an empty sequence) and some observer  $ob$  of  $C$ . In order to prove that  $\Theta(u_1) \approx_{obs} \Theta(u_2)$ , we need only prove that  $\Theta(u_1).v.ob \approx_{obs} \Theta(u_2).v.ob$ . Since  $u_1 \sim_{obs} u_2$ , by Lemma 5(a),  $u_1.v \sim_{obs} u_2.v$ . Hence, if statement (e) is true, we have  $\Theta(u_1.v) \approx_{att} \Theta(u_2.v)$ . By Definition 11, therefore,  $\Theta(u_1.v).ob \approx_{obs} \Theta(u_2.v).ob$ . Since the implementation is complete,  $\Theta(u_1).v.ob \approx_{obs} \Theta(u_2).v.ob$ .  $\square$

The following corollary is a direct result of Definition 12 and Theorem 3.

*Corollary 1 (Error in Equivalence).* Given a canonical specification of a class with proper imports, suppose its implementation is complete. Any of the following statements indicates an error in the implementation. Furthermore, the statements are equivalent to one another.

- (a)  $\Theta(u_1)$  and  $\Theta(u_2)$  are observationally nonequivalent for some observationally equivalent ground terms  $u_1$  and  $u_2$ .
- (b)  $\Theta(u_1)$  and  $\Theta(u_2)$  are observationally nonequivalent for some normally equivalent ground terms  $u_1$  and  $u_2$ .
- (c)  $\Theta(u_1)$  and  $\Theta(u_2)$  are attributively nonequivalent for some normally equivalent ground terms  $u_1$  and  $u_2$ .
- (d)  $\Theta(u_1)$  and  $\Theta(u_2)$  are attributively nonequivalent for some attributively equivalent ground terms  $u_1$  and  $u_2$ .
- (e)  $\Theta(u_1)$  and  $\Theta(u_2)$  are attributively nonequivalent for some observationally equivalent ground terms  $u_1$  and  $u_2$ .

In the events of (a) and (e), we say that the test case  $(u_1 \sim_{obs} u_2)$  reveals an error. In the events of (b) and (c), we say that the test case  $(u_1 \sim_{nor} u_2)$  reveals an error. In the event of (d), we say that the test case  $(u_1 \sim_{att} u_2)$  reveals an error.

Note that the following do not necessarily entail an error in the implementation:

- (f)  $\Theta(u_1)$  and  $\Theta(u_2)$  are observationally nonequivalent for some attributively equivalent ground terms  $u_1$  and  $u_2$ .
- (g)  $\Theta(u_1)$  and  $\Theta(u_2)$  are not identical objects for some observationally equivalent ground terms  $u_1$  and  $u_2$ .
- (h)  $\Theta(u_1)$  and  $\Theta(u_2)$  are not identical objects for some normally equivalent ground terms  $u_1$  and  $u_2$ .

PROOF OF COROLLARY 1. Statements (a) to (e) in this corollary are, respectively, the negations of statements (a) to (e) of Theorem 3, which are mutually equivalent. Hence, statements (a) to (e) in this corollary are mutually equivalent.

Statement (a) in this corollary is the negation of the equivalence criterion in Definition 12. Hence, there will be an error in the implementation if statement (a) in this corollary is true. Furthermore, since statements (a) to (e) in this corollary are mutually equivalent, there will also be an error in the implementation if one of the statements (b) to (e) in this corollary is true.  $\square$

The observational equivalence of terms in a specification is intuitively the most straightforward yardstick for the observational equivalence of

objects in an implementation. As we have indicated earlier, however, this is not useful in testing practice because the observational equivalence of terms cannot be easily verified. In view of practical considerations, during the first phase of our TACCLE project [Chen et al. 1998], we chose to use the normal equivalence of terms as a test case selection criterion instead. Theorem 3 and Corollary 1 confirm that there is no compromise on the equivalence criterion.

**THEOREM 4 (NONEQUIVALENCE CRITERIA).** *Given a canonical specification of a class with proper imports, suppose its implementation is complete. The following statements are equivalent:*

- (a) *For any observationally nonequivalent ground terms  $u_1$  and  $u_2$ ,  $\Theta(u_1)$  and  $\Theta(u_2)$  are observationally nonequivalent.*
- (b) *For any attributively nonequivalent ground terms  $u_1$  and  $u_2$ ,  $\Theta(u_1)$  and  $\Theta(u_2)$  are attributively nonequivalent.*
- (c) *For any attributively nonequivalent ground terms  $u_1$  and  $u_2$ ,  $\Theta(u_1)$  and  $\Theta(u_2)$  are observationally nonequivalent.*

PROOF.

**(a) implies (b):**

For any attributively nonequivalent ground terms  $u_1$  and  $u_2$ ,

- (i) If no observer of the class is applicable to  $u_1$  and  $u_2$ , then by Lemma 4, no observable context on the class is applicable to them either. Since  $u_1$  and  $u_2$  are attributively nonequivalent, by Theorem 1(c), they are observationally nonequivalent. Hence, if statement (a) is true,  $\Theta(u_1)$  and  $\Theta(u_2)$  must be observationally nonequivalent. Since no observable context on the class is applicable to  $u_1$  and  $u_2$ , by Definition 1, the objects  $\Theta(u_1)$  and  $\Theta(u_2)$  cannot be identical. By Definition 4.2,  $\Theta(u_1)$  and  $\Theta(u_2)$  must be attributively nonequivalent.
- (ii) Suppose some observer of the class is applicable to  $u_1$  and  $u_2$ . By Definition 4.2, there exists some  $ob$  such that  $\neg(u_1.ob \sim_{obs} u_2.ob)$ . Hence, if statement (a) is true, we have  $\neg[\Theta(u_1.ob) \approx_{obs} \Theta(u_2.ob)]$ . Since the implementation is complete,  $\neg[\Theta(u_1).ob \approx_{obs} \Theta(u_2).ob]$ . Thus, by Definition 10,  $\Theta(u_1)$  and  $\Theta(u_2)$  must be attributively nonequivalent.

**(b) implies (c):**

For any attributively nonequivalent ground terms  $u_1$  and  $u_2$ , if statement (b) is true,  $\Theta(u_1)$  and  $\Theta(u_2)$  must be attributively nonequivalent. By Theorem 2, therefore,  $\Theta(u_1)$  and  $\Theta(u_2)$  must be observationally nonequivalent.

**(c) implies (a):**

For any observationally nonequivalent ground terms  $u_1$  and  $u_2$ ,

- (i) If no observable context on the class is applicable to  $u_1$  and  $u_2$ , the output class of  $u_1$  and  $u_2$  is a primitive type. By Definition 4.2, the normal forms of  $u_1$  and  $u_2$  cannot be identical. By Lemma 4, no observer of the class is applicable to  $u_1$  and  $u_2$ . Hence, according to Definition 4.2,  $u_1$  and  $u_2$  are attributively nonequivalent. Thus, if statement (c) is true,  $\Theta(u_1)$  and  $\Theta(u_2)$  are observationally nonequivalent.
- (ii) Suppose some observable contexts  $oc$  on the class are applicable to  $u_1$  and  $u_2$ . By Definition 4.2, for at least one such  $oc$ , we have  $\neg(u_1.oc \sim_{obs} u_2.oc)$ . Now, we can write  $oc = v.ob$  for some sequence of operations  $v$  (possibly an empty sequence) and some observer  $ob$  of  $C$ . In other words,  $\neg(u_1.v.ob \sim_{obs} u_2.v.ob)$ . Hence, by Definition 10,  $\neg(u_1.v \sim_{att} u_2.v)$ . If statement (c) is true, therefore,  $\neg[\Theta(u_1.v) \approx_{obs} \Theta(u_2.v)]$ . Thus, by Lemma 5(b)  $\Theta(u_1)$  and  $\Theta(u_2)$  are observationally nonequivalent.  $\square$

The next corollary follows immediately from Definition 12 and Theorem 4.

*Corollary 2 (Error in Nonequivalence).* Given a canonical specification of a class with proper imports, suppose its implementation is complete. Any of the following statements indicates an error in the implementation. Furthermore, the statements are equivalent to one another.

- (a)  $\Theta(u_1)$  and  $\Theta(u_2)$  are observationally equivalent for some observationally nonequivalent ground terms  $u_1$  and  $u_2$ .
- (b)  $\Theta(u_1)$  and  $\Theta(u_2)$  are attributively equivalent for some attributively nonequivalent ground terms  $u_1$  and  $u_2$ .
- (c)  $\Theta(u_1)$  and  $\Theta(u_2)$  are observationally equivalent for some attributively nonequivalent ground terms  $u_1$  and  $u_2$ .

In the event of (a), we say that the test case  $\neg(u_1 \sim_{obs} u_2)$  reveals an error. In the events of (b) and (c), we say that the test case  $\neg(u_1 \sim_{att} u_2)$  reveals an error.

Note that the following do not necessarily entail an error in the implementation:

- (d)  $\Theta(u_1)$  and  $\Theta(u_2)$  are observationally equivalent for some normally nonequivalent ground terms  $u_1$  and  $u_2$ .
- (e)  $\Theta(u_1)$  and  $\Theta(u_2)$  are attributively equivalent for some observationally nonequivalent ground terms  $u_1$  and  $u_2$ .
- (f)  $\Theta(u_1)$  and  $\Theta(u_2)$  are attributively equivalent for some normally nonequivalent ground terms  $u_1$  and  $u_2$ .

PROOF OF COROLLARY 2. The proof is similar to that of Corollary 1.  $\square$

Theorem 4 provides us with alternatives to the nonequivalence criterion in Definition 12. As a result, Corollary 2 provides us with alternatives for detecting errors in nonequivalence. Furthermore, statement (d) after Corollary 2 reinforces our earlier finding that Assertion 2 does not hold in the context of normal nonequivalence. On the other hand, statements (a) and (b) in Corollary 2 indicate that Assertion 2 *does* hold in the context of observational and attributive nonequivalence, respectively.

The theoretical result is simple and elegant. However, which of these alternatives is better from the practical view of a software tester? The most obvious choices are between statements (a) and (b) of Corollary 2. Intuitively, there appears to be a trade-off between the complexity of operation sequences and that of verifying the equivalence of the resulting objects. One may argue that, for the same error, error-exposing attributively nonequivalent terms are generally longer than error-exposing observationally nonequivalent terms. Unfortunately, the lengths of error-exposing observationally nonequivalent terms cannot be known before test case selection. Hence, we must select test cases from the set of all pairs of observationally nonequivalent terms, which is infinite in general. Thus, the task of selecting error-exposing observationally nonequivalent terms is more complex than this simple argument.

We can compare the task of testing based on observationally nonequivalent terms with that based on attributively nonequivalent terms by breaking up each of them into two subtasks:

- (a) Testing based on observationally nonequivalent terms includes:
  - (a1) Selecting test cases from the set  $S_{obs}$  of pairs of observationally nonequivalent terms, which is infinite in general.
  - (a2) Selecting observable contexts from the set  $S_{oc}$  of possible observable contexts.
- (b) Testing based on attributively nonequivalent terms includes:
  - (b1) Selecting test cases from the set  $S_{att}$  of pairs of attributively nonequivalent terms, which is infinite in general.
  - (b2) Selecting observers from the set  $S_{ob}$  of all observers.

Our GAN approach in Section 4.4 deals with the infinite set  $S_{att}$  using techniques in state-transition diagrams (STD), and turns subtask (b1) into a terminating process. However, the same approach cannot be used to handle the infinite set  $S_{obs}$  in (a1), since the concept of states in STD relates directly to attributive equivalence rather than observational equivalence. Furthermore, the set  $S_{ob}$  in (b2) is small and finite whereas  $S_{oc}$  in (a2) is infinite in most cases. Hence, subtask (b2) is generally much more effective than subtask (a2).

As a result of these analyses, we recommend testing based on attributively nonequivalent terms. Thus, we shall present in Sections 4.4 to 4.7 a methodology to perform class-level testing using attributively nonequivalent terms as test cases.

#### 4.4 Generating Nonequivalent Terms from State-Transition Diagrams

In this section, we consider how to generate representative attributively nonequivalent ground terms as test cases. Given a canonical specification of a class with proper imports, suppose  $T$  is the set of all ground terms. Suppose  $T$  is further partitioned into  $k$  equivalence classes<sup>4</sup>  $T_1, T_2, \dots, T_k$  with respect to the attributive equivalence of terms. If we randomly select a ground term  $u_i$  from each  $T_i$ , we will obtain  $k(k - 1)/2$  pairs of attributively nonequivalent terms  $\neg(u_i \sim_{att} u_j)$  as test cases, where  $i \neq j$ , and  $i, j = 1, 2, \dots, k$ .

The remaining question is how  $T$  can be partitioned with respect to the attributive equivalence of terms. Intuitively speaking, if two ground terms in a class  $C$  are attributively equivalent, the corresponding two objects in  $C$  have the same set of attributive values. In other words, the two objects have the same state. More formally, a state represents an equivalence class of objects in  $C$ . The set of all the states in  $C$  is called the *state space* of  $C$ .

If the state space of  $C$  is finite and not large, we can perform the partitioning as follows: Construct the *state-transition diagram* for the given class  $C$ , where each *node* denotes a state, and each *arc* represents an operation transforming one state into another. A *path* is a sequence of contiguous arcs, corresponding to a sequence of operations, or in other words, a term. The state established by the creator is called the *initial state*. Let the node corresponding to the initial state be called the *initial node* and labeled by  $n_0$ . All the terms corresponding to the paths from the initial node  $n_0$  to a given node  $n_i$  form an equivalence class  $T_i$ .

If the state space of the class  $C$  is infinite or large, we can use the approach proposed by Turner and Robson [1993a; 1993b; 1995] to partition the state space into a finite number of *subspaces*. Each node in the state-transition diagram denotes a subspace, rather than a concrete state. For example, consider a scenario where each object in a given class  $C$  has two attributes  $a$  and  $b$ . Suppose that according to the functional specification the domain of  $a$  can be partitioned into three subdomains  $A_1 = \{a \mid a < 0\}$ ,  $A_2 = \{a \mid a = 0\}$ , and  $A_3 = \{a \mid a > 0\}$ , and the domain of  $b$  can be partitioned into two subdomains  $B_1 = \{b \mid -1 \leq b < 0\}$  and  $B_2 = \{b \mid 0 \leq b \leq 1\}$ . Then we partition the state space of  $C$  into six subspaces  $A_i \times B_j$ ,  $i = 1, 2, 3$  and  $j = 1, 2$ .

We observe the following:

- (i) In the case of state space partitioning, two terms corresponding to two paths from the initial node  $n_0$  to the same node  $n_i$  are not necessarily attributively equivalent. On the other hand, two terms corresponding to two paths from the initial node  $n_0$  to different nodes  $n_i$  and  $n_j$  must be attributively nonequivalent.

<sup>4</sup>Here, “equivalence class” is a discrete mathematics concept rather than an object-oriented concept.

- (ii) One potential problem from the above observation is that state transitions are not always deterministic. Consider, for example, a stack of Boolean values with operations *push*, *pop*, *top*, and *isEmpty*. Since *top* and *isEmpty* are the only observers, there will be three states: ( $n_0$ ) the empty stack; ( $n_1$ ) stacks with “true” at the top; and ( $n_2$ ) stacks with “false” at the top. A *pop* transition from state ( $n_1$ ) or ( $n_2$ ) can lead to any of the states, and is therefore “nondeterministic” at the state-transition level. This issue is an inherent limitation of most object-oriented testing methods based on state transitions. In the context of our approach, the problem can be solved as follows:

Let us define *current sequence* as the operation sequence on a current path from the initial node to the current node  $n_i$ , and denote it by *CS*. A “nondeterministic” transition from the current node will involve more than one transition arc from  $n_i$ . We label each arc with a *guard condition* after the operation name. In the above example, the *pop* transition arc from the node  $n_1$  to the node  $n_0$  is labeled with  $pop | CS.pop.top = nil$ ; the *pop* arc from  $n_1$  to itself is labeled with  $pop | CS.pop.top = true$ ; the *pop* arc from  $n_1$  to  $n_2$  is labeled with  $pop | CS.pop.top = false$ ; and so on.

When traversing a given path up to the current node, the current sequence *CS* is obviously unique and can be obtained. Once *CS* is known, the appropriate transition arc can be determined from the guard condition according the specification. Hence, every transition arc is well defined. In this way, we can refine a “nondeterministic” transition into deterministic ones.

For a given node, different paths leading to it have different values of *CS*, and hence we cannot write the actual value of *CS* on the guard condition in the state-transition diagram. The notation *CS* on the guard condition in the diagram serves only as an identifier. When we use rewriting to determine whether a guard condition is satisfied, we must replace *CS* by its actual value, which is the sequence of concrete operations from the initial node to the current node.

It should be noted that, for more complex classes, the guards may become quite complicated. Suppose, for instance, that there are  $k$  attributes  $a_1, a_2, \dots, a_k$  in a given class, and suppose that there are  $n_i$  values (or  $n_i$  subdomains)  $a_{i,1}, a_{i,2}, \dots, a_{i,n_i}$  for each attribute  $a_i$ . Suppose, further, that the observer to return the value of attribute  $a_i$  is  $ob_i$ . Then for a given current sequence *CS* and a given operation *op*, the general form of a guard will be

$$\bigwedge_{i=1,2,\dots,k} (CS.op.ob_i = a_{i,j})$$

for some  $j \in \{1, 2, \dots, n_i\}$ .<sup>5</sup>

Based on state-transition diagrams, we have:

*The GAN Approach (for Generating Attributively Nonequivalent Terms as Test Cases).* Given a canonical specification of a class with proper imports, the following steps generate attributively nonequivalent terms as test cases:

- (1) Based on the specification, construct a state-transition diagram (STD) for the class, including guard conditions, if any.
- (2) Let  $\{n_0, n_1, \dots, n_k\}$  denote the set of nodes in the STD, where  $n_0$  is the initial node. For *each* node  $n_i$  other than the initial node  $n_0$ , find a path  $p_i$  from  $n_0$  to  $n_i$ . Every guard condition along the path, if any, must be satisfied according to the specification.
- (3) Let  $u_i$  denote the ground term representing the operation sequence in the path  $p_i$ . Take the  $k(k - 1)/2$  pairs of attributively nonequivalent terms  $\neg(u_i \sim_{att} u_j)$  as test cases, where  $i \neq j$ , and  $i, j = 1, 2, \dots, k$ .
- (4) If one of the pairs generated in step (3) reveals an error, then exit from the procedure. Otherwise, generate more paths  $p_i$  from the initial node  $n_0$  to every node  $n_i$ . If a cycle  $cyc$  is encountered in a path, ask the user to determine a ceiling  $t_{cyc}$  for the number of iterations of  $cyc$ , or specify a global ceiling  $T$  for the system.<sup>6</sup> The ceilings  $t_{cyc}$  or  $T$  correspond to some boundary values in the code.

We use the following strategy to curtail the number of generated paths: Paths with lengths corresponding to the boundary values are generated first. Since boundary values are usually more sensitive to errors [Jeng and Weyuker 1994; White and Cohen 1980], the chances of exposing errors by such paths are higher. Once an error is revealed by a pair of paths, the execution of GAN will terminate, so that no other paths will need to be generated. If an error is not detected, then randomly generate some paths with lengths between the boundary values, to test the nonboundary cases. If no error is detected from the random paths, then report that no error has been revealed, and exit from the procedure. *End of algorithm.*

Readers may find the following points useful for understanding the GAN approach:

<sup>5</sup>When  $a_{i,j}$  denotes a subdomain, the corresponding item “ $CS.op.ob_i = a_{i,j}$ ” should be replaced by “ $CS.op.ob_i \in a_{i,j}$ ”.

<sup>6</sup>The determination of  $t_{cyc}$  and  $T$  remains a difficult problem. This is an inherent limitation of program testing. It has been addressed in Sections 2.5.2 and 3.3.3 of our companion paper [Chen et al. 1998] and will not be repeated here. Fortunately, this issue is alleviated for the case of the GAN approach in the light of discussion point (e) below.

- (a) Our techniques for partitioning the state space and constructing the state-transition diagram are similar to those described in Turner and Robson [1993a; 1993b; 1995].
- (b) The following theorem provides the theoretical justification for the selection strategy in steps (2) and (3). According to this theorem, if we can ensure that the equivalence criterion in Definition 12 has been satisfied, then for every node  $n_i$  other than the initial node  $n_0$ , we need only select one path from  $n_0$  to  $n_i$ .

**THEOREM 5.** *Consider a canonical specification of a class with proper imports and a complete implementation satisfying the equivalence criterion in Definition 12. If no error is revealed for some nonequivalent test case  $\neg(u_1 \sim_{att} u_2)$ , then no error will be revealed for every nonequivalence test case  $\neg(u'_1 \sim_{att} u_2)$  such that  $u'_1 \sim_{att} u_1$ .*

**PROOF.** Assume the contrary. Then there exist some test case  $\neg(u_1 \sim_{att} u_2)$  which does not reveal any error, and some  $u'_1 \sim_{att} u_1$  such that  $\neg(u'_1 \sim_{att} u_2)$  reveals an error. By Corollary 2,  $(\Theta(u'_1) \approx_{att} \Theta(u_2))$ . Since  $u_1 \sim_{att} u'_1$ , if the equivalence criterion has been fulfilled, by Theorem 3(d),  $\Theta(u_1) \approx_{att} \Theta(u'_1)$ . By the transitivity property of the equivalence relation, therefore,  $(\Theta(u_1) \approx_{att} \Theta(u_2))$ . This contradicts the assumption that the test case  $\neg(u_1 \sim_{att} u_2)$  does not reveal any error.  $\square$

- (c) In general situations, it is of course impossible to prove by means of software testing that the implementation fully satisfies the equivalence criterion. Hence, we may need more test paths from the initial node to the nodes. Step (4) serves this purpose.
- (d) Although the equivalence criterion cannot be proved by means of software testing, if the users have a certain confidence for their test results on equivalent ground terms, Theorem 5 will enable them to have the same confidence on the selection strategy in steps (2) and (3) of the GAN approach.
- (e) It may be argued that the number of test cases involved in step (4) may be unreasonably large in many situations. We observe, however, that this step has been introduced as an extra precaution because the equivalence criterion in Theorem 5 cannot be proved. Hence, users do not have to decide on the adequacy of the testing of nonequivalence of test cases based on step (4) alone, but make their decisions in conjunction with observations (b), (c), and (d).

#### 4.5 Determining whether a Test Case of Nonequivalent Terms Reveals an Error

Suppose the attributively nonequivalent terms  $\neg(u_1 \sim_{att} u_2)$  are selected as a class-level test case for a given specification using the GAN approach in Section 4.4. To apply this test case to an implementation of the

specification, we should map each operation in  $u_1$  and  $u_2$  to a method in the program. If the implementation is complete, this mapping is well defined. It can be supported either manually by the implementation designer or automatically according to a given interface specification. Suppose this mapping generates two method sequences  $s_1$  and  $s_2$  in the implementation. Let  $O_1$  and  $O_2$  be two objects resulting from the execution of  $s_1$  and  $s_2$ , respectively. In order to judge whether the test case  $\neg(u_1 \sim_{att} u_2)$  reveals an implementation error, by Corollary 2(b), we should use the set of all observers of the given class to determine whether  $O_1 \approx_{att} O_2$ . If so, an implementation error is revealed. Otherwise, this test case does not expose any implementation error. This decision is effective, since the set of all observers of the given class must be finite, and since its size is small in general.

#### 4.6 Discussions on Basic Assumptions

In the above approaches for class-level testing, we have assumed that the given specification is canonical and has proper imports and that the implementation is complete. We would like to discuss whether these assumptions are too restrictive and hence not useful in practice.

(1) *Canonical Specifications*: Intuitively, a ground term represents a sequence of operations on an object, while the normal form of the ground term denotes the “abstract object value” [Breu and Breu 1993]. Given a canonical specification, every ground term can be transformed into a unique normal form in a finite number of steps. This means that every sequence of operations on any object must result in a unique “abstract object value.” If we relaxed the canonical requirement for a specification, then two executions of the same sequence of operations on the same object might result in two different “abstract object values.” In that case, we would not be able to decide whether the software under test contains an error. Thus, it should be reasonable to do software testing against canonical specifications.

(2) *Proper Imports*: Consider the following example of improper imports:

*Example 7.* Suppose the classes *Stack'* and *List* in a specification import each other. The output class of an observer *top* of *Stack'* is *List*, while that of an observer *head* of *List* is *Stack'*. In this case, the imports to the specification are improper because infinite oc sequences such as *newStack'.top.head.top.head. . .* will result. *End of Example 7.*

Such specifications are a nuisance not only to software testing but also to software development in general.

(3) *Complete Implementations*: If an implementation is not complete, we will have the following situations:

(a) There exists some operation  $f_0$  that is (i) not implemented by any

method or (ii) implemented by two or more different methods in the same class. Case (i) is obviously an error, since the implemented system will fail when  $f_0$  is called. Case (ii) is ambiguous, since the implemented system can be executed with two different outcomes. On the other hand, the problem can easily be identified by comparing the list of operations in the specification with the list of methods in the implementation. We recommend that this trivial checking be done before any attempt to test the software comprehensively.

- (b) There exists some imported class in the specification that is not implemented by any imported class in the program, or implemented by two or more different imported classes. This trivial problem should also be identified before any attempt to test the software thoroughly.
- (c) There exists some primitive type in the specification that has not been implemented. Such kind of error can easily be detected.

In summary, it is reasonable to require a specification to be canonical and contain proper imports, since no useful conclusion can be drawn from the testing of a program against an ambiguous specification. It is also acceptable to require an implementation to be complete because the checking is trivial and because any incompleteness will lead to immediate problems.

#### 4.7 Implementation and Experimentation of the GAN Approach

In the GAN approach, a rewriting technique is used to construct state-transition diagrams from the algebraic specifications of given classes. This can easily be implemented in Prolog. Hence, we have developed an interactive prototype of the GAN approach using Arity/Prolog32 for Windows 98. Interested readers may refer to our supplementary report [Chen and Tse 2000] for the source code of the top-level module of the Prolog implementation.

We have experimented with a number of scenarios on the GAN prototype. One of them is related to an algebraic specification for the class *IntStack* of stacks of nonnegative integers with a maximum size of 10. The following are some of the axioms in the specification:

$$\begin{array}{ll}
 a_6 : S.push(N).height = S.height + 1 & \text{if } S.height < 10 \\
 a_7 : S.push(N).top = N & \text{if } S.height < 10 \\
 a_8 : S.push(N) = S & \text{if } S.height = 10 \\
 a_9 : S.push(N).pop = S & \text{if } S.height < 10
 \end{array}$$

Suppose an implementation of the class contains a single error as follows:

```

void intStack :: push(int i)
{
  if (ht < 9)      /* Error: The condition should be ht <= 9 */
  {
    ht = ht + 1;
    array[ht] = i;
  }
}

```

The above error cannot be revealed using equivalent ground terms as test cases. It can be revealed, however, by test cases of attributively nonequivalent ground terms through the GAN approach. First, a state transition diagram is constructed for the given class, consisting of four nodes:

$$\begin{aligned} n_0 & \text{ (initial node),} \\ n_1 & = (\text{empty} = \text{true}, \text{top} = \text{nil}, \text{ht} = 0), \\ n_2 & = (\text{empty} = \text{false}, \text{top} \geq 0, 1 \leq \text{ht} \leq 9), \\ n_3 & = (\text{empty} = \text{false}, \text{top} \geq 0, \text{ht} = 10). \end{aligned}$$

One path from  $n_0$  to  $n_1$ , two paths from  $n_0$  to  $n_2$ , and one path from  $n_0$  to  $n_3$  are then generated. The ground terms corresponding to these paths are:

$$\begin{aligned} u_1 & = \text{new}, \\ u_{21} & = \text{new.push}(1), \\ u_{29} & = \text{new.push}(1).\text{push}(2).\text{push}(3).\dots.\text{push}(8).\text{push}(9), \\ u_3 & = \text{new.push}(1).\text{push}(2).\text{push}(3).\dots.\text{push}(8).\text{push}(9).\text{push}(10). \end{aligned}$$

The following pairs of attributively nonequivalent ground terms are selected as test cases in the order as listed:

$$\begin{aligned} & \neg(u_1 \sim_{att} u_{21}), \neg(u_1 \sim_{att} u_{29}), \neg(u_1 \sim_{att} u_3), \neg(u_{21} \sim_{att} u_3), \text{ and} \\ & \neg(u_{29} \sim_{att} u_3). \end{aligned}$$

The last pair of nonequivalent terms,  $\neg(u_{29} \sim_{att} u_3)$ , reveals the implementation error above, since the execution results are, respectively,

$$\begin{aligned} & \textbf{(array, ht)} \\ O_{29} & = ([1, 2, \dots, 9], 9), \\ O_3 & = ([1, 2, \dots, 9], 9) \end{aligned}$$

and obviously  $O_{29} \approx_{att} O_3$ .

Readers may refer to our supplementary report [Chen and Tse 2000] for more examples and further details of the experimentation on the GAN prototype.

## 5. CONTRACT SPECIFICATIONS

Even if an implementation is consistent with respect to all equivalent and nonequivalent ground terms derived from the algebraic specification of each individual class, it may still have faults in the behavioral dependencies or interactions among cooperating objects of different classes in a given cluster, since these dependencies and interactions may not be expressed in terms of the algebraic specifications. Such faults cannot be covered by

class-level tests. Cluster-level testing is therefore necessary. In our approach, cluster-level testing is based on Contract specifications.

An algebraic specification places emphasis on the functions of the attributes and operations in a class. An interface specification stresses the description of the mapping from ground terms in the functional specification to method sequences in the implementation of the class. Neither of them is suitable for the specification of message passing and other interactions among cooperating classes in a given cluster for the purpose of testing. The following example illustrates this point.

*Example 8.* Suppose the operation

*SavingAccount.transferTo(CheckAccount, M)*

transfers money  $M$  from a saving account *SavingAccount* to a check account *CheckAccount* in the cluster of a bank system. When  $M$  does not exceed the balance of *SavingAccount*, this operation consists of two functions:

- (1) Debit money  $M$  from *SavingAccount*, i.e., send and activate the message *debit(M)* to *SavingAccount*.
- (2) Then credit  $M$  to *CheckAccount*, i.e., send and activate the message *credit(M)* to *CheckAccount*.

In an algebraic specification of *SavingAccount*, the first function may be described by an axiom:

$$\begin{aligned} & \textit{SavingAccount.transferTo(CheckAccount, M).balance} \\ & = \textit{SavingAccount.debit(M).balance} \\ & \textit{if } M \leq \textit{SavingAccount.balance}. \end{aligned}$$

The second function, however, cannot be described by pure algebraic specifications.

In formal specification languages with an object interface layer, such as in Larch [Gutttag and Horning 1993], these two functions can be described as follows:

$$\begin{aligned} & \textit{SavingAccount transferTo(CheckAccount CA, Money M)} \\ & \{ \\ & \quad \textit{if self.balance} \geq M \textit{ then} \\ & \quad \{ \\ & \quad \quad \textit{result} = \textit{self.debit(M)}; \\ & \quad \quad (\textit{CA})' = (\textit{CA})^\wedge.\textit{credit(M)}; \\ & \quad \quad \}; \\ & \quad \dots \\ & \} \end{aligned}$$

where the symbol  $\wedge$  refers to the value immediately before executing the procedure, and the prime symbol ( $'$ ) refers to the value immediately afterward. However, this description for message passing is implicit and scattered among different operations. In other functional object-oriented languages such as FOOPS [Goguen and Meseguer 1987; Borba and Goguen

1994] this is also implemented implicitly using side effects. Thus, these languages are not immediately suitable for specifying message-passing properties that are vital to cluster-level testing. *End of Example 8.*

On the other hand, Contract, a formal specification technique proposed by Helm et al. [1990], describes systematically and explicitly the interactions and message passing among cooperating classes in a given cluster. It “extends the usual type signatures to include constraints on behavior that capture the behavioral dependencies between objects of classes” [Helm et al. 1990]. We propose, therefore, that Contract specifications be used in cluster-level testing.

The main syntax of a Contract specification is the message-passing rule.

*Definition 13 (Message-Passing Rule).* In a given cluster, a statement  $r_i$  is called a **message-passing rule** or an **mp-rule** if and only if it contains the symbol  $=>$ . The left-hand side of the symbol  $=>$  is called the **head** of the rule, and the right-hand side is called the **body**. If the body contains the symbol “;” (semicolon), then each part separated by a semicolon is called an **mp-item** of the body. Otherwise, we say that the body consists of a single mp-item. Each mp-item specifies a message-passing expression, a group of message-passing expressions, a postcondition, or a related action such as setting a value or returning a value. A **message-passing expression** is of the form *Class*  $<-$  *Message*. A group of message-passing expressions may be specified in one of two forms; namely ( $expression_0 / expression_1 / \dots / expression_n$ ) or ( $/ Variable : condition : expression$ ).

The semantics of Contract specifications can be explained by the following example:

*Example 9.* A cluster *CustomerAccount* contains the class *Customer* and the class *Accounts*. Each object of the class *Customer* is a customer of the bank. An object of the class *Accounts* is a set of *Account*’s (such as saving account, check account, and fixed deposit account) that belong to one customer.

The message passing and interactions between the classes *Customer* and *Accounts* can be specified by the following Contract:

```

contract CustomerAccount
  Customer supports
  [
    address : String
    accounts : Accounts
    ...
    r1 : Customer <- setAddress(S : String) =>
      @Customer.address; {Customer.address = S};
      Customer <- notify()
    r2 : Customer <- getAddress() =>
      return Customer.address
    r3 : Customer <- notify() =>
      (/ Ac : Ac in accounts : Ac <- update())

```

```

r4 : Customer <- openAccount(Ac : Account) =>
    { Ac in accounts }
r5 : Customer <- closeAccount(Ac : Account) =>
    { Ac not_in accounts }
]
Accounts : SetOf(Account) where each Account supports
[
    customer : Customer
    freeze : Boolean
    ...
r6 : Account <- setFreeze(B : Boolean) =>
    @Account.freeze; { Account.freeze = B }
r7 : Account <- update() =>
    if Account.freeze then return "Account is frozen"
    else Account <- changeAddr()
r8 : Account <- changeAddr() =>
    customer <- getAddress();
    { Account reflects customer.address }
r9 : Account <- setCustomer(C : Customer) =>
    { customer = C }
]
instantiation
(/ Ac : Ac in Accounts : (Customer <- openAccount(Ac) /
    Ac <- setCustomer(Customer)))
end contract

```

*End of Example 9.*

Here, the words in **bold** are reserved words of the Contract language. An object in the class *Customer* has at least two attributes, namely, *address* of type *String* and *accounts* of the class *Accounts*. An object of the class *Customer* can accept the messages *setAddress*, *getAddress*, *notify*, *openAccount*, and *closeAccount*. The class *Account* contains at least two attributes; namely *customer* of the class *Customer* and *freeze* of type *Boolean*. *r1*, *r2*, . . . , *r9* are statement labels for the ease of reference.

*Class <- Message => ContractSequence*

means that an object of a *Class* receiving the *Message* will result in *ContractSequence*, where *ContractSequence* is a sequence of message-passing expressions, postconditions, or related actions. The message-passing expression

*Class <- Message*

means sending a *Message* to an object of the *Class*. Each message corresponds to a specific operation or method. For example, statement *r1* means

that when an object of the *Customer* class receives a message *setAddress*(*S* : *String*), it will result in a sequence

*@Customer.address*; {*Customer.address* = *S*}; *Customer* ← *notify*()

The notation *Object.attribute* denotes the value of the *attribute* of the *Object* while *Object.operation(Parameters)* denotes the result of executing the *operation* on the *Object* using the *Parameters*. The notation *@Object.attribute* sets a value to the *attribute* of the *Object*. A condition in curly brackets {}, such as {*Customer.address* = *S*}, is a postcondition. *P*; *Q* denotes two items *P* and *Q* occurring sequentially, while *P* / *Q* denotes two items occurring in any sequence. A notation of the form (*/ V : condition : expression*) means, for all the values of the variable *V* that satisfy the *condition*, perform the *expression* repeatedly in any sequence. For example,

(*/ Ac : Ac in accounts : Ac* ← *update*())

is interpreted as “*Ac*<sub>1</sub> ← *update*() / *Ac*<sub>2</sub> ← *update*() / ... for all *Ac*<sub>1</sub>, *Ac*<sub>2</sub>, ... **in** *accounts*”.

*Example 10.* Referring to Example 8, the two functions of the operation *SavingAccount.transferTo(CheckAccount, M)* can be described using the following statement *r1* in Contract:

```

contract Accounts
  SavingAccount supports
  [
    balance : Money
    ...
    r1 : SavingAccount ← transferTo(CheckAccount :
      CheckAccount, M : Money) =>
      if M ≤ SavingAccount.balance then
        [SavingAccount ← debit(M);
         CheckAccount ← credit(M)]
      else return "overdrawn"
    ...
  ]
  CheckAccount supports
  ...
end contract

```

*End of Example 10.*

Finally, to facilitate manipulations, we will represent every mp-rule of the form

*Class* ← *Message* => *Items*<sub>1</sub>; **if** *P* **then** *Q* **else** *R*; *Items*<sub>2</sub>

by two mp-rules

*Class*  $\leftarrow$  *Message*  $\Rightarrow$  *Items*<sub>1</sub>; if *P* then *Q*; *Items*<sub>2</sub>

and

*Class*  $\leftarrow$  *Message*  $\Rightarrow$  *Items*<sub>1</sub>; if not *P* then *Q*; *Items*<sub>2</sub>.

For instance, mp-rule *r1* of Example 10 will be represented by

```

r1a : SavingAccount  $\leftarrow$  transferTo(CheckAccount :
    CheckAccount, M : Money  $\Rightarrow$ 
    if M  $\leq$  SavingAccount.balance then
        [SavingAccount  $\leftarrow$  debit(M);
         CheckAccount  $\leftarrow$  credit(M)]
r1b : SavingAccount  $\leftarrow$  transferTo(CheckAccount :
    CheckAccount, M : Money  $\Rightarrow$ 
    if M  $>$  SavingAccount.balance then
        return 'overdrawn'

```

## 6. CLUSTER-LEVEL TESTING WITH INDIVIDUAL MP-RULES

The third phase of our TACCLE methodology covers cluster-level testing, including that related to individual mp-rules and that related to composite message-passing sequences.

Each mp-rule in the Contract can be individually used for cluster-level testing, since the behavioral dependencies and interactions among cooperating objects of the classes in a given cluster are described in the mp-rules of the Contract for the cluster.

### 6.1 The TIM Approach

We propose a TIM approach for **T**esting the interactions using **I**ndividual **M**p-rules. The approach is illustrated by the following example:

*Example 11.* Referring to Example 10, in order to use the mp-rule *r1* for cluster-level testing, the following steps should be taken:

- (1) Perform class-level testing on the class *SavingAccount* and the class *CheckAccount*, respectively.
- (2) Analyze the body of the mp-rule *r1* to find the messages passing across different classes, such as *CheckAccount*  $\leftarrow$  *credit*(*M*), from the class *SavingAccount* under the condition *M*  $\leq$  *SavingAccount.balance*. The message passing should take place in the operation *transferTo*.
- (3) Construct an object *O<sub>chk</sub>* of the class *CheckAccount* by running a sequence of operations in the program of the class *CheckAccount*, such as the sequence

*newCheckAccount*('John').*credit*(1500).*writeCheck*(1000).

Save its current state in the variable *Pre\_O<sub>chk</sub>*.

- (4) Construct an object  $O_{sav}$  of the class *SavingAccount* by running a sequence of operations in the program of the class *SavingAccount*, such as the sequence

*newSavingAccount('John').credit(2000).debit(300).*

We note that the sequence must not contain the operation *transferTo*.

- (5) Run  $O_{sav}.transferTo(O_{chk}, M)$  in the program of the cluster *Accounts*, where  $M$  is a value of the class *Money* satisfying the condition  $M \leq O_{sav}.balance$ . During the execution, the object  $O_{sav}$  will activate a method  $Md$  to be executed on  $O_{chk}$ . This method  $Md$  serves to implement the message *credit*( $M$ ) passed to  $O_{chk}$ . It will change the state of  $O_{chk}$ .
- (6) Run  $Pre\_O_{chk}.credit(M)$ , and examine whether the execution result is observationally equivalent to  $O_{chk}$  using the DOE algorithm described in Section 3. If not, report an implementation error on the message *CheckAccount*  $\leftarrow credit(M)$  in the operation *transferTo* of the class *SavingAccount*. *End of Example 11.*

In general, we have the following:

*The TIM Approach (for Cluster-Level Testing by Individual Mp-Rules).*

Let *Clus* be a cluster containing two classes, *sender* and *receiver*. Let  $O_{sender}$  and  $O_{receiver}$  denote the objects of the respective classes. Suppose

```

r : O_sender <- operation(O_receiver : receiver, Parameters) => ... ;
  if predicate1(Parameters), then
    O_receiver <- operation'1(Parameters'1) ; ... ;
  if predicate2(Parameters), then
    O_receiver <- operation'2(Parameters'2) ; ... ;
  if predicaten(Parameters), then
    O_receiver <- operation'n(Parameters'n) ; ... ;

```

is an mp-rule in the Contract for *Clus*. In other words, the body of the mp-rule contains  $n$  messages  $operation'_i(Parameters'_i)$ ,  $i = 1, 2, \dots, n$ , passed to the object  $O_{receiver}$ . The following are the steps for cluster-level testing based on the individual mp-rule:

- (1) Perform class-level testing on the classes *sender* and *receiver*, respectively.
- (2) Analyze the body of the mp-rule  $r$  to find the messages  $operation'_i(Parameters'_i)$ ,  $i = 1, 2, \dots, n$ , sent to the object  $O_{receiver}$ .
- (3) Based on the GAN approach, select a path  $p_j$  from the initial node to some node in the state-transition diagram (STD) of the class *receiver*. Construct a concrete object  $O_{receiver}$  in the class *receiver* by running the operation sequence corresponding to the path  $p_j$ . Save the current state of the object in the variable  $Pre\_O_{receiver}$ .

- (4) Similarly, select a path  $p_k$  from the initial node to some node in the STD of the class *sender*. Construct a concrete object  $O_{sender}$  in the class *sender* by running the operation sequence corresponding to the path  $p_k$ . Note that the sequence must not contain the operation *operation* in the mp-rule  $r$ .
- (5) Randomly select a set of values of *Parameters* that satisfy the conditions  $predicate_i(Parameters)$ ,  $i = 1, 2, \dots, n$ . Run  $O_{sender}.operation(O_{receiver}, Parameters)$  in the program of the cluster *Clus*. If the conditions are not specified, select any set of values from the domain(s) of *Parameters*. During the execution, the class  $O_{sender}$  will activate the corresponding methods  $Md_i$ ,  $i = 1, 2, \dots, n$ , to be executed on  $O_{receiver}$ . Each method  $Md_i$  serves to implement the message  $operation'_i(Parameters'_i)$  passed to  $O_{receiver}$ . These messages will change the state of  $O_{receiver}$ .  
If no value of *Parameters* satisfies the conditions  $predicate_i(Parameters)$ ,  $i = 1, 2, \dots, n$ , then backtrack to step (3) or (4) to traverse a path to another node in the STD and construct another  $O_{receiver}$  or  $O_{sender}$ . This is repeated until the conditions are satisfied or until every node  $n_k$  in the STD has been considered. In the latter case, report that no error has been found, and exit from the procedure.
- (6) Run  $Pre\_O_{receiver}.operation'_i(Parameters'_i)$ ,  $i = 1, 2, \dots, n$ , sequentially. Using the DOE algorithm described in Section 3 and in Chen et al. [1998], examine whether the final execution result is observationally equivalent to  $O_{receiver}$ . If not, report an implementation error corresponding to the message passed to  $O_{receiver}$ , and exit from the procedure. Otherwise, backtrack to step (3) or (4) to traverse a path to another node in the STD and construct another  $O_{receiver}$  or  $O_{sender}$ . If every node  $n_k$  in the STD has been considered in the backtracking process, report that no error has been found, and exit from the procedure.

## 6.2 Discussions on the TIM Approach

- (a) Consider step (5) of the TIM approach. Suppose in the implementation of

$$O_{sender}.operation(O_{receiver}, Parameters),$$

the implementor introduces a new condition  $p_2(O_{sender}, O_{receiver}, Parameters)$  under the condition  $predicate_i(Parameters)$ , thus resulting in two implementation subbranches. In order to improve on the comprehensiveness of test cases, we should select two groups of values of  $(O_{sender}, O_{receiver}, Parameters)$  such that one group satisfies the conditions

$$predicate_i(Parameters) = true$$

and

$$p_2(O_{sender}, O_{receiver}, Parameters) = true,$$

and the other satisfies the conditions

$$predicate_i(Parameters) = true$$

and

$$p_2(O_{sender}, O_{receiver}, Parameters) = false.$$

This partitioning is based on the implementation and hence is a white-box approach.

- (b) Instead of randomly selecting the values of *Parameters* in step (5) of the TIM approach, we can alternatively adopt the domain strategy of White and Cohen [1980] or the simplified testing strategy of Jeng and Weyuker [1994] so as to improve on the effectiveness.
- (c) Every mp-rule of the form

$$Class \leftarrow Message \Rightarrow Items_1; \text{ if } P \text{ then } Q \text{ else } R; Items_2$$

has been divided into two mp-rules as indicated at the end of Section 5. By applying the TIM approach to these two mp-rules, we are partitioning the input domain of parameters of *P* into two subdomains. *P* is true in one subdomain and false in the other. This partitioning is based on the Contract specification and hence is a black-box approach.

- (d) In step (5), we should select a set of values of *Parameters* that satisfy the conditions  $predicate_i(Parameters)$ ,  $i = 1, 2, \dots, n$ . According to the statistical investigations by White and Cohen [1980], most conditions in real-life programs are simple predicates. This is especially the case for class-level methods in object-oriented programs. Hence, this step is reasonable.
- (e) Suppose there are  $j$  nodes in the state-transition diagram (STD) of the class *receiver* and  $k$  nodes in the STD of the class *sender*. Then the maximum number of backtracking will be  $j \times k$ . This would not be excessive especially when each node in an STD denotes a subspace rather than a concrete state.

### 6.3 Implementation and Experimentation of the TIM Approach

In order to implement the TIM approach, we need only write a submodule AMP to Analyze the body of the given MP-rule to find the messages passing across different classes in the cluster. Then we can construct a control module CM to integrate AMP with GFT, DOE, and GAN. The module CM will call and coordinate AMP, GFT, DOE, and GAN to perform

the requirements described in Section 6.1. They will be incorporated into an integrated testing system in our future work, as outlined in Section 8.

A case study of the TIM approach has been conducted. It deals with the cluster *BankAccounts*, which contains the classes *SavingAccount* and *CheckAccount*. Suppose an implementation of the cluster contains a single error as follows:

```
void savingAccount :: transferTo(checkAccount *ca, money m)
{
  if (balance >= m)
  {
    debit(m);
    ca -> writeCheck(m);
    /* Error: writeCheck(m) should be credit(m) */
  }
  else cout << "overdrawn";
}
```

This error cannot be revealed by class-level testing, regardless of whether we use equivalent or nonequivalent ground terms as test cases. It can be revealed, however, by cluster-level testing using the TIM approach. Interested readers may refer to our supplementary report [Chen and Tse 2000] for more details.

## 7. CLUSTER-LEVEL TESTING WITH COMPOSITE MESSAGE-PASSING SEQUENCES

A *composite message-passing sequence* from the Contract specification contains message-passing expressions, postconditions, and related actions. The sequence is generated by joining the mp-rules in the Contract specification. In Example 9, for instance, the mp-rule *r8* can be joined with *r2* into a composite message-passing sequence

```
Account <- changeAddr(); Customer <- getAddress();
return Customer.Address; {Account reflects Customer.address}
```

which is obtained by inserting the body of *r2* into the body of *r8* at the position after the mp-item *Customer <- getAddress()*.

Each mp-rule corresponds to a program component, that is, a method in a class. When a program is an integration of smaller parts, both the isolated parts and the integrated whole must be tested. The success of one test may not guarantee the same for the other. This property of software testing is well known to most software testers, and has been formally stated as the anticomposition and antidecomposition axioms by Perry and Kaiser [1990] and Weyuker [1986]. Thus, we should test the interactions among classes using each individual mp-rule separately, as well as the interactions based on composite message-passing sequences. The former is discussed in the last section and the latter in this section.

The procedure for testing composite message-passing sequences can be summarized as three components as follows. They will be discussed in detail in Sections 7.2 to 7.5.

- (1) The GCS algorithm generates a composite message-passing sequence *CompositeContractSequence* from the Contract specification of *Class*  $\leftarrow$  *Message*.
- (2) The ESI and GCS algorithms together produce the corresponding composite message-passing sequence *CompositeImplementSequence* from the implementation of *Class*  $\leftarrow$  *Message*.
- (3) We then check whether the message-passing expressions and related actions in *CompositeImplementSequence* match those in *CompositeContractSequence*, and whether the postconditions in *CompositeContractSequence* really hold when performing *Class*  $\leftarrow$  *Message* in the program implementing the cluster. If not, then there is an error.

In our system, steps (1) and (2) are done interactively. However, step (3) can only be done manually for general situations.

### 7.1 The Necessity of Testing Composite Message-Passing Sequences

A question arises here. In step (5) of the TIM approach, when  $O_{receiver}.operation'(Parameters')$  is being run, *operation'* may automatically invoke another mp-rule  $r_{i_1}$ , which may invoke another mp-rule  $r_{i_2}$ , and so on. In such a situation, is it necessary to check the composite message-passing sequences of  $r$  joined with  $r_{i_1}, r_{i_2}, \dots, r_{i_k}$ ? The answer is yes, because of the following reasons:

- (a) Suppose a message-passing expression  $m_j$  that appears in *operation'* or in the body of one of the mp-rules  $r_{i_1}, r_{i_2}, \dots, r_{i_k}$  is missing from the implementation. Since neither  $O_{receiver}.operation'(Parameters')$  in step (5) of the TIM approach nor  $Pre\_O_{receiver}.operation'(Parameters')$  in step (6) executes  $m_j$ , their execution results should be observationally equivalent, and hence the error cannot be revealed by the TIM approach.
- (b) By running  $O_{receiver}.operation'(Parameters')$  and  $Pre\_O_{receiver}.operation'(Parameters')$ , the TIM approach traverses only one of the paths produced by joining *operation'* with  $r_{i_1}, r_{i_2}, \dots, r_{i_k}$ .
- (c) Suppose a message-passing expression  $m_i$  appears in the body of the mp-rule  $r_{i_1}$  and in the head of the mp-rule  $r_{i_2}$ , and suppose the implementations of  $r_{i_1}$  and  $r_{i_2}$  use different names for the same message-passing expression  $m_i$  by mistake. Since neither  $O_{receiver}.operation'(Parameters')$  nor  $Pre\_O_{receiver}.operation'(Parameters')$  invokes  $r_{i_2}$ , their execution results may be observationally equivalent, and hence the error cannot be exposed by the TIM approach.

In order to cater for such a situation, we should have an additional procedure to check the composite message-passing sequences. The GCS

algorithm for joining message-passing sequences, as presented in Section 7.2, will support this additional procedure.

## 7.2 Basic Idea of the GCS Algorithm

We propose a GCS Algorithm for **Generating Composite Message-passing** sequences, as outlined in this section.

Suppose the Contract specification of a given cluster is held in a file named *contractFile*. The composite message-passing sequences are placed in a file named *contractSequenceFile*. The GCS algorithm reads and analyzes the *contractFile*, and generates and outputs the composite message-passing sequences to the *contractSequenceFile*. It processes every mp-rule of the given Contract as follows.

- (1) If the first mp-item in the body of the current mp-rule is a postcondition or a related action, output it to the *contractSequenceFile* and then delete the item.
- (2) If the first mp-item is a message-passing expression  $Class \leftarrow Message$  that cannot match the head of any mp-rule of the Contract specification, output it to the *contractSequenceFile* and then delete this mp-item.
- (3) If the first mp-item is a message-passing expression  $Class \leftarrow Message$  that can match the head of another mp-rule  $Class \leftarrow Message \Rightarrow Body_1$  of the Contract specification, output it to the *contractSequenceFile* and then replace this mp-item by  $Body_1$ .
- (4) If the first mp-item is of the form “if  $P$  then  $Q$ ”, output “if  $P$ :” to the *contractSequenceFile* and then replace this mp-item by  $Q$ .
- (5) If the first mp-term is of the form  $( / V : condition : Class \leftarrow Message )$ , where  $Class \leftarrow Message$  can match the head of another mp-rule  $Class \leftarrow Message \Rightarrow Body_2$  of the Contract specification, output it to the *contractSequenceFile* and then replace this mp-item by  $Body_2$ .
- (6) If the first mp-item is of the form  $( / V : condition : Class \leftarrow Message )$ , but  $Class \leftarrow Message$  cannot match the head of any mp-rule of the Contract specification, output it to the *contractSequenceFile* and then delete this mp-item.
- (7) Repeat the above iteratively for the first mp-item of the updated body until the updated body is empty.

Readers may refer to our supplementary report [Chen and Tse 2000] for the details of the GCS Algorithm.

## 7.3 Testing Tool Based on the GCS Algorithm

Based on the GCS algorithm, we have constructed a testing tool in Arity/Prolog for generating composite message-passing sequences from Contract specifications. The tool is also known as GCS, and consists of a

translator and a generator. The translator transforms the given Contract specification into Prolog syntax, from which the generator produces the expected composite message-passing sequences.

**7.3.1 Reason for Using Prolog.** We appreciate that the semantics of an mp-rule in Contract is very different from that of a rule in Prolog. As far as control mechanism is concerned, however, the concepts of mp-rules, heads, bodies, and mp-items in Contract specifications used in the GCS algorithm are very similar to those of rules, heads, bodies, and subgoals in the Prolog inference engine, respectively. The techniques of searching, unification, and substitution in the GCS algorithm are fundamentally the same as those in the Prolog inference engine [Chen 1987; Chen and Wah 1987; Clocksin and Mellish 1994]. It is therefore appropriate and convenient to use the Prolog inference engine for implementing the GCS algorithm above.

**7.3.2 The Translator.** The first part of the GCS tool is a translator, which transforms every mp-rule in the given Contract specification into a Prolog rule. It transforms the symbols “=>” and “;”, the message-passing expression *Class* <- *Message*, postcondition {*Post*}, and the action *Act* in each mp-rule of the Contract into the Prolog operators “:-” and “;”, the Prolog structures *send(Class, Message)*, *postconditionPost*, and *action(Act)*, respectively. It also transforms the notation (*/ V : condition : expression*) in Contract into the Prolog structure *forall(/ V, condition, expression)*, and translates the mp-item of the form “if *P* then *Q*” into the Prolog structure *if(P, Q)*.

The transformed Contract specification is called a *Contract in Prolog form*. The translator inputs the *contractFile* and produces the corresponding Contract in Prolog form, which is output to the *contractPrologFile*. The translator has been implemented using standard techniques, and the implementation is quite straightforward.

**7.3.3 The Generator.** The second part of the GCS tool is a generator, which is an implementation of the GCS algorithm in Arity/Prolog. It accepts the *contractPrologFile* and generates the expected composite message-passing sequences to an output file *contractSequenceFile*.

The main program for the generator can be found in Chen and Tse [2000].

## 7.4 Capturing Composite Message-Passing Sequences from an Implementation

The capturing of composite message-passing sequences from an implementation involves two processes. The first process extracts a sequence of message-passing expressions and related actions from each given method by analyzing the source code of the method. The message-passing sequence from each method in a given implemented class is unique,<sup>7</sup> and corresponds to an individual mp-rule in the Contract. The second process joins these sequences using the GCS algorithm.

<sup>7</sup>Readers may recall that a message-passing sequence may contain not only message-passing expressions but also conditional expressions and related actions.

7.4.1 *Extracting Message-Passing Sequences.* Suppose the cluster of *Accounts* described in Example 10 is implemented by the following program:

*Example 12 (Program for the Cluster of Accounts).*

```

// Accounts.hpp
typedef float money;
class savingAccount
{
    private :
        money balance;
        ...
    public :
        void createSavingAccount();
        money bal();
        void credit(money m);
        void debit(money m);
        void transferTo(checkAccount *ca, money m);
        ...
};
class checkAccount
{
    private :
        money balance;
        ...
    public :
        void createCheckAccount();
        money bal();
        void credit(money m);
        void writeCheck(money m);
        ...
};
...

// Accounts.cpp
#include <iostream.h>
#include <string.h>
#include <Accounts.hpp>
void savingAccount :: transferTo(checkAccount *ca, money m)
{
    if (balance >= m)
    {
        debit(m);
        ca -> credit(m);
    }
    else cout << "overdrawn";
}
...

```

From the program, we can extract the following message-passing sequence for the method *transferTo*(\_, \_).

```
FOR MESSAGE transferTo(*ca, m) SENT TO savingAccount:
savingAccount <- transferTo(*ca, m);
if (savingAccount.balance >= m):
savingAccount <- debit(m);
(ca ->) <- credit(m);
end-if (savingAccount.balance >= m);
if not (savingAccount.balance >= m):
cout << 'overdrawn';
end-if not (savingAccount.balance >= m);
```

*End of Example 12.*

The task of extracting a message-passing sequence for each implemented method, as illustrated in Example 12, is handled by the following algorithm:

*The ESI Algorithm (for Extracting a Message-Passing Sequence for Each Method from the Implementation).* This algorithm inputs the program *Prog* implementing the given cluster and, for each method in the implementation, outputs a sequence of message-passing expressions and related actions to the *implementSequenceFile*.

- (1) Input *Prog*, and open the *implementSequenceFile* for writing;
- (2) Write “MESSAGE-PASSING AND ACTION SEQUENCES FROM THE IMPLEMENTATION *Prog*” to the current line of the *implementSequenceFile*;
- (3) For each *Class* in the given cluster, do (4);
- (4) For each *Method* in the *Class*, do (5) and (6);
- (5) Write a blank line to the *implementSequenceFile*;  
Start a new line for the *implementSequenceFile*;  
Write “FOR SENDING MESSAGE *Method*(*Parameters*) TO *Class*:” to the current line of the *implementSequenceFile*;  
Start a new line for the *implementSequenceFile*;  
Write “*Class* <- *Method*(*Parameters*);” to the current line of the *implementSequenceFile*;
- (6) Invoke the recursive procedure *process*(*Method*) to analyze the code of the *Method*.

```
process(Method):
for each statement ST in Method do
{
if ST is an if-then-else statement,
```

```

{
  recognize the condition CD, the then-body BD1,
    and the else-body BD2;
  start a new line in the implementSequenceFile
    and write "if CD:" to this line;
  process(BD1);
  start a new line in the implementSequenceFile
    and write "end-if CD;" to it;
  start a new line in the implementSequenceFile
    and write "if-not CD:" to it;
  process(BD2);
  start a new line in the implementSequenceFile
    and write "end-if-not CD;" to it;
};
if ST is an if-then, while, or for statement,
{
  recognize the condition CD and the body BD1;
  start a new line in the implementSequenceFile
    and write "if CD:" to this line;
  process(BD1);
  start a new line in the implementSequenceFile
    and write "end-if CD;" to it;
};
if ST is a return or "cout << . . ." statement,
  start a new line in the implementSequenceFile
    and write the statement to this line;
if ST is a statement of the form
"dataMember = constant", "dataMember = parameter",
or "dataMember = arithmetic expression",
  start a new line in the implementSequenceFile
    and write "@Class#dataMember;" to this line;
if ST is a statement of the form "ObjectOfClass1.Method1;",
  start a new line in the implementSequenceFile
    and write "ObjectOfClass1 <- Method1;" to this line;
if ST is a statement of the form "Class1 -> Method1;" (where
Class1 is a pointer to an object of a class different from Class),
  start a new line in the implementSequenceFile
    and write "(Class1 ->) <- Method1;" to this line;
if ST is a statement of the form
"dataMember = Class1 -> Method1;" (where Class1 is
a pointer to an object of a class different from Class),
{
  start a new line in the implementSequenceFile
    and write "(Class1 ->) <- Method1;" to this line;
  start a new line in the implementSequenceFile
    and write "@Class#dataMember;" to it;
}

```

```

if  $ST$  is a statement of the form “Method1,”
  (where Method1 is a method of Class),
  start a new line in the implementSequenceFile
  and write “Class <- Method1,” to this line;
}

```

*End of algorithm*

Note that we extract message-passing sequences from the implementation by static analysis of the code rather than by dynamic execution. When a “while” or “for” statement is encountered, we extract the message-passing sequence from the body of the statement once and only once, rather than once for each iteration of the loop. There is no nesting or loop problem. Readers may refer, for instance, to Example 12 above.

Since the time for parsing and handling every statement in *Prog* by the ESI algorithm has an upper bound, the complexity of the algorithm is  $O(n)$ , where  $n$  denotes the number of statements in the program *Prog*. Based on this algorithm, we have developed an automatic tool ESI for extracting sequences of message-passing expressions and related actions from the implementation. Since this algorithm must parse the code of the program *Prog*, a better way to implement it is to embed the algorithm into a compiler or interpreter, as in the case of the tool DOE [Chen et al. 1998].

**7.4.2 Joining Message-Passing Sequences.** In order to compare the message-passing sequences in the *implementSequenceFile* produced by the ESI algorithm with the composite message-passing sequences generated from Contract by the GCS algorithm, the sequences of message-passing expressions and related actions in the *implementSequenceFile* must be joined. This can be done by inputting *implementSequenceFile* to the GCS tool,<sup>8</sup> running the tool, and outputting the result into another file *CompositeImplementSequenceFile*.

Thus, message-passing sequences in the *CompositeImplementSequenceFile* are composite.

## 7.5 Determining whether a Composite Message-Passing Sequence Reveals an Error

Suppose *ContractSequence* is a composite message-passing sequence for *Class* <- *Message* in the *contractSequenceFile* generated by the GCS testing tool. Suppose, further, that *ImplementSequence* is the composite sequence for *Class* <- *Message* in the *CompositeImplementSequenceFile* produced by the testing tools ESI and GCS. We must verify whether *ImplementSequence* matches *ContractSequence*. If not, an implementation error is identified.

To verify whether *ImplementSequence* matches *ContractSequence*, we must ensure that every message-passing expression or related action in

<sup>8</sup>Since the format of message-passing sequence contained in the *implementSequenceFile* is a little different from that of mp-rule in the *contractFile*, there are two entry points for GCS tool. One is for the former, and the other is for the latter.

*ContractSequence* has been implemented by a method or a set of methods in *ImplementSequence*. To facilitate the checking, it is assumed that a mapping of the components of the Contract specification to the components in the implementation has been specified by the software designer. In case the message-passing expression contains a precondition, we must also check whether the precondition in *CompositeImplementSequence* satisfies its counterpart in *CompositeContractSequence*. The whole verification process is done manually.

Note that *ContractSequence* may contain postconditions, but *ImplementSequence* may not. It is impossible to extract postconditions from a program implemented in C++ into *ImplementSequence*, even though it is possible in better-designed languages such as Eiffel. If the above examination of *ImplementSequence* with reference to *ContractSequence* cannot reveal an implementation error, we need to check manually whether the postconditions in *ContractSequence* are really satisfied when executing the programs.

## 7.6 Discussions on GCS and ESI

- (a) Generating all composite message-passing sequences by statically analyzing the source code is generally an undecidable problem. Instead of doing this, we only analyze the source code to extract a sequence of message-passing expressions related to a given method. Every method in a given implemented class has one and only one such sequence, which corresponds to an individual mp-rule in the Contract. We then join the extracted sequences using the GCS algorithm.
- (b) The above task of extracting a sequence of message-passing expressions within a given method is done by the ESI algorithm. The complexity of the ESI algorithm is analyzed in Section 7.4.1.
- (c) Like the Prolog inference mechanism, the GCS algorithm is rule-based. The techniques of searching, unification, and substitution in the algorithm are fundamentally the same as those in Prolog. Hence, the GCS algorithm is as “reasonable” as the Prolog inference engine.
- (d) Instead of monitoring the passing messages by means of program instrumentation, we statically extract from the source code an individual sequence of passing messages and related actions *for each method*, and then join them using the GCS algorithm.
- (e) Like the Prolog inference mechanism, the working of the GCS algorithm is based on the rules given in the Contract specification. If its instantiation incurs a nonterminating computation, the latter is a result of problematic rules in the specification, rather than resulting from the GCS algorithm. In such a situation, the same problem will apply to a dynamic monitoring approach by means of program instrumentation.

## 7.7 Implementation and Experimentation on GCS and ESI

A more elaborate version of the GCS generator has been used in the actual prototype to provide user-friendly interfaces. Interested readers may refer to our supplementary report [Chen and Tse 2000] for the main program written. Some experimental results on GCS can also be found in the report.

Similarly to DOE, we have combined the implementation of the ESI algorithm with a compiler, since the algorithm must parse the code of the program under test. Readers may refer to our supplementary report [Chen and Tse 2000] for the main idea as well as further details. For example, we present an implementation of the cluster *CustomerAccount*, whose Contract specification is described in Example 9. For each method, we also present the message-passing sequence, which can be extracted from the implementation using the ESI algorithm.

## 8. OPEN ISSUES AND FUTURE WORK

The previous sections contain only some initial experimental results of the implementation of our proposal. An in-depth empirical study would be beyond the scope of this paper. We are setting up a new project to develop an integrated testing system that incorporates the prototypes GFT and DOE in Chen et al. [1998] with GAN, TIM, GCS, and ESI in this paper, with a view toward further experimentation on real-life programs.

The cluster-level testing with composite message-passing sequences given in this paper is only static. On the other hand, the original ESI algorithm can be revised easily to support program instrumentation, so that the passing of messages can be monitored dynamically. Schematically, we need only change all the statements

start a new line in the *implementSequenceFile* and write “X”  
to this line;

in step (6) of the algorithm into

insert the statement “*fprintf(aFile, %s. . . s%, 'X')*” to the code  
before or after the currently parsed statement;

We plan to supplement this dynamic monitoring approach by program instrumentation as an alternative for cluster-level testing. More details for this dynamic approach will be considered in our future work.

In this paper, we have not considered the nondeterministic choice operators and concurrency issues in object-oriented program testing. We will consider them as future work based on the Java language.

## 9. CONCLUSION

Our TACCLE methodology for object-oriented software testing consists of three components: testing fundamental pairs of equivalent ground terms at the class level; testing nonequivalent ground terms at the class level; and testing sequences of message-passing expressions and postconditions at the cluster level.

We can reduce the test case selection domain if we use fundamental pairs as class-level test cases, but the comprehensiveness is no less than that of using equivalent ground terms as test cases. This strategy is based on mathematical theorems proved in Chen et al. [1998]. An interactive tool DOE has been developed to support this strategy.

The generation of nonequivalent ground terms as test cases is a necessary and nontrivial task for object-oriented testing at the class level. In order to enhance the foundations and correct some nontrivial problems in related work, we have defined in this paper a few important concepts on term equivalence. They are rewriting relations, normal equivalence, observational equivalence, and attributive equivalence. We have investigated in detail the relationships among these four types of relations among terms. We find, that given a canonical specification of a class, if two ground terms are observationally or attributively nonequivalent, but their corresponding implemented method sequences produce observationally or attributively equivalent objects, respectively, then there is an error in the implementation. Based on these findings and using state-transition diagrams, we propose an approach to generate attributively nonequivalent ground terms as test cases in class-level testing.

Cluster-level testing for object-oriented programming is important but has seldom been investigated so far. This paper shows the feasibility of using Contract, a formal specification language, for black-box testing at the cluster level. An approach for cluster-level testing using every individual message-passing rule in the Contract has been given. An algorithm for cluster-level checking using the composite message-passing rules has also been proposed. The similarities and distinctions between the control mechanism of the algorithm and the Prolog inference engine have been analyzed. Based on this analysis, an implementation of the algorithm for generating composite message-passing sequences by Arity/Prolog has been presented. Two automatic tools GCS and ESI have been developed to support cluster-level testing.

#### ACKNOWLEDGMENTS

We highly appreciate the anonymous referees for their invaluable comments and suggestions, which has led our paper to a greater depth. We are also grateful to Mr. Yue Tang Deng, now with Polytechnic University, Brooklyn, NY, and Mr. Shun Long of Jinan University for their help in the implementation and experimentation of some of the prototypes.

#### REFERENCES

- BERNOT, G., GAUDEL, M.-C., AND MARRE, B. 1991. Software testing based on formal specifications: A theory and a tool. *Softw. Eng. J.* 6, 6 (Nov.), 387–405.
- BORBA, P. AND GOGUEN, J. A. 1994. An operational semantics for FOOPS. In *Proceedings of the International Workshop on Information Systems: Correctness and Reusability (IS-CORE '94, Amsterdam)*, R. J. Wieringa and R. B. Feenstra, Eds.
- BOUGE, L., CHOQUET, N., FRIBOURG, L., AND GAUDEL, M.-C. 1986. Test sets generation from algebraic specifications using logic programming. *J. Syst. Softw.* 6, 343–360.

- BREU, R. 1991. *Algebraic Specification Techniques in Object-Oriented Programming Environments*. Springer-Verlag, New York, NY.
- BREU, R. AND BREU, M. 1993. 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. Springer-Verlag, Berlin, Germany, 343–348.
- CHEN, H. Y. 1987. The heuristic search algorithm A\*LP for logic programs. In *Proceedings of the 2nd International Symposium on Intelligent System Methodologies*. 51–65.
- CHEN, H. Y. AND TSE, T. H. 2000. Prototypes and initial experimentation on the tools of the TACCLE methodology. <http://www.csis.hku.hk/tse/Papers/staccSupp.pdf>.
- CHEN, H. Y. AND WAH, B. 1987. The “rid-redundant” procedure in C-Prolog. In *Proceedings of the 2nd International Symposium on Intelligent System Methodologies*. 71–83.
- CHEN, H. Y., TSE, T. H., CHAN, F. T., AND CHEN, T. Y. 1998. In black and white: An integrated approach to class-level testing of object-oriented programs. *ACM Trans. Softw. Eng. Methodol.* 7, 3, 250–295.
- CLARKE, E. M. AND WING, J. M. 1996. Formal methods: State of the art and future directions. *ACM Comput. Surv.* 28, 4, 626–643.
- CLOCKSIN, W. F. AND MELLISH, C. S. 1994. *Programming in Prolog*. 4th ed. Springer-Verlag, Berlin, Germany.
- DOONG, R.-K. AND FRANKL, P. G. 1991. Case studies on testing object-oriented programs. In *Proceedings of the Symposium on Testing, Analysis, and Verification (TAV4, Victoria, British Columbia, Oct. 8–10)*, W. Howden, Chair. ACM Press, New York, NY, 165–177.
- DOONG, R.-K. AND FRANKL, P. G. 1994. The ASTOOT approach to testing object-oriented programs. *ACM Trans. Softw. Eng. Methodol.* 3, 2 (Apr.), 101–130.
- FIEDLER, S. P. 1989. Object-oriented unit testing. *Hewlett-Packard J.* 40, 4, 69–74.
- FRANKL, P. G. AND DOONG, R.-K. 1990. Tools for testing object-oriented programs. In *Proceedings of 8th Pacific Northwest Conference on Software Quality*. 309–324.
- GOGUEN, J. A. AND MALCOLM, G. 2000. A hidden agenda. *Theor. Comput. Sci.* 245, 1, 55–101.
- GOGUEN, J. A. AND MESEGUER, J. 1987. 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 Series in Computer Systems. MIT Press, Cambridge, MA, 417–478.
- GUTTAG, J. V. AND HORNING, J. J., EDS. 1993. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, New York, NY.
- HARROLD, M. J., MCGREGOR, J. D., AND FITZPATRICK, K. J. 1992. Incremental testing of object-oriented class structures. In *Proceedings of the 14th International Conference on Software Engineering (ICSE '92, Melbourne, Australia, May 11–15)*, T. Montgomery, Chair. ACM Press, New York, NY, 68–80.
- HELM, R., HOLLAND, I. M., AND GANGOPADHYAY, D. 1990. Contracts: Specifying behavioral compositions in object-oriented systems. In *Proceedings of the Joint ACM European Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA/ECOOP '90, Ottawa, Canada, Oct. 21–25)*, N. Meyrowitz, Ed. ACM Press, New York, NY, 169–180.
- JENG, B. AND WEYUKER, E. J. 1994. A simplified domain-testing strategy. *ACM Trans. Softw. Eng. Methodol.* 3, 3 (July), 254–270.
- JORGENSEN, P. C. AND ERICKSON, C. 1994. Object-oriented integration testing. *Commun. ACM* 37, 9 (Sept.), 30–38.
- KUNG, D. C. H., GAO, J. Z., HSIA, P., TOYOSHIMA, Y., AND CHEN, C. 1995. A test strategy for object-oriented programs. In *Proceedings of the 19th Annual International Conference on Computer Software and Applications (COMPSAC '95)*. IEEE Computer Society Press, Los Alamitos, CA, 239–244.
- KUNG, D. C. H., SUCHAK, N., HSIA, P., TOYOSHIMA, Y., AND CHEN, C. 1994. On object state testing. In *Proceedings of the 18th Annual International Conference on Computer Software and Applications (COMPSAC '94)*. IEEE Computer Society Press, Los Alamitos, CA, 222–227.

- PERRY, D. E. AND KAISER, G. E. 1990. Adequate testing and object-oriented programming. *J. Object Oriented Program.* 2, 5 (Jan./Feb.), 13–19.
- SMITH, M. D. AND ROBSON, D. J. 1992. A framework for testing object-oriented programs. *J. Object Oriented Program.* 5, 3 (June), 45–53.
- TURNER, C. D. AND ROBSON, D. J. 1993a. Guidance for the testing of object-oriented programs. TR-2/93. University of Durham, Durham, UK.
- TURNER, C. D. AND ROBSON, D. J. 1993b. State-based testing and inheritance. TR-1/93. University of Durham, Durham, UK.
- TURNER, C. D. AND ROBSON, D. J. 1995. A state-based approach to the testing of class-based programs. *Software: Concepts and Tools* 16, 3, 106–112.
- WEYUKER, E. J. 1986. Axiomatizing software test data adequacy. *IEEE Trans. Softw. Eng. SE-12*, 12 (Dec.), 1128–1138.
- WHITE, L. J. AND COHEN, E. I. 1980. A domain strategy for computer program testing. *IEEE Trans. Softw. Eng. SE-6*, 3, 247–257.

Received: July 1998; revised: January 1999 and January 2000; accepted: May 2000