2009 Ninth International Conference on Quality Software

Automatic Generation of Normal Forms for Testing Object-Oriented Software*

Huo Yan Chen
Department of Computer Science
Jinan University
Guangzhou 510632, P.R. China
tchy@jnu.edu.cn

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

***Abstract*—Testing object-oriented software is more complex than its procedural counterpart. The concept of "fundamental pairs" was introduced in our TACCLE methodology for testing object-oriented software. It was proved that, although the set of fundamental pairs is only a proper subset of equivalent ground terms, the use of fundamental pairs as test cases covers the use of equivalent ground terms. Recently, we found that any normal form consists of only a creator, or a creator followed only by constructors but not transformers; and yet the reverse is not necessarily true. Thus, the generation of patterns of normal forms is non-trivial and warrants further study. Motivated by this finding and based on further pattern analyses of normal forms and tree models with pruning techniques, we propose an algorithm to generate representative normal forms according to patterns and develop a corresponding automatic tool. This work improves the automation, coverage, and adequacy of selecting (equivalent) fundamental pairs as test cases.**

***Keywords*—algebraic specification; object-oriented program; software testing; equivalent fundamental pair; normal form**

I. INTRODUCTION

The testing of object-oriented software requires new theories and techniques that are different from those for its procedural counterpart [16]. Algebraic specifications have been used to generate test cases since the 1980s. The DAISTS system developed by Gannon et al. [9] inputs a tuple of arguments into both sides of an algebraic axiom, and employs a user-supplied equality function to check the output. An error is revealed when the outputs from the two sides do not agree.

A general theory for software testing based on algebraic specifications, including regularity, uniformity, and oracle hypotheses, was proposed by Bernot et al. [1]. Based on the theory, they developed a tool to generate test cases by replacing all the variables in the axioms with ground terms [1][2][6].

Doong and Frankl proposed an algebraic specification language LOBAS, which is more suitable for object-oriented programming. Based on LOBAS, a tool named ASTOOT was developed to generate class-level test cases, including equivalent terms through rewriting and non-equivalent terms by exchanging path conditions [7][8].

Following up on ASTOOT, Chen et al. [3][4][4] further proposed a methodology called TACCLE for object-oriented class-level testing. The concept of a *fundamental pair*, defined as a pair of terms constructed by replacing all the variables on both sides of an axiom by normal forms, is proposed. Obviously, the set of fundamental pairs is a proper subset of the set of equivalent ground terms. It is proved 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 testers need only concentrate on the testing of fundamental pairs. In order to do this, an algorithm GFT has been proposed for generating finite number of equivalent fundamental pairs as test cases using axioms and normal forms. The normal forms used in the original GFT algorithm are generated manually.

The TACCLE project has attracted a lot of attention among researchers on software testing. More notable examples are [12][14][15][17][19][20].

Recently, we found that any normal form consists of only a creator, or a creator following only by constructors but not transformers; *but the reverse is not necessarily true*. Thus, the generation of representative normal forms is not trivial.[1] An algorithm and the corresponding tool to aid the generation of representative normal forms are necessary and significant. If the generated normal forms do not cover all possible patterns within a given length, the resulting tests will not be adequate. Based on pattern analysis, a tree model, and pruning technique, we propose an algorithm GNF to generate normal forms. We also

* This research is supported by a Union Grant of the Guangdong Province and National Natural Science Foundation of China (#U0775001), a Grant of National Natural Science Foundation of China (#60773083), grants of the Guangdong Province Science Foundation (#7010116 and #8151063201000022), and the General Research Fund of the Research Grant Council of Hong Kong (project no. 717308).

** All correspondence should be addressed to Prof. T. H. Tse at Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong. Tel: (+852) 2859 2183. Email: thtse@cs.hku.hk.

[1] See, for instance, Example 2 and Proposition 4 in Section 3.2.

1550-6002/09 $26.00 © 2009 IEEE
DOI 10.1109/QSIC.2009.22

implement a tool to aid the process.

This paper is organized as follows: Section 2 presents the basic concepts used in the paper. Section 3 describes the structural pattern analysis of normal forms. Section 4 investigates the generation of representative normal forms using Algorithm GNF and a computer-aided tool. Finally, Section 5 gives a conclusion.

II. Basic Concepts

We summarize the basic concepts of algebraic specifications in this section. More details can be found in [2][3][9][10][13][18].

The syntax declaration and semantic definition are the two main components of an *algebraic specification* of a class. The *syntax declaration* specifies the input and output parameters of each operator. The *semantic definition* specifies the equational *axioms*, which describe the behavior related to each operator.

Example 1. The following example (from [2]) shows an algebraic specification of the class *IntStack* of integer stacks:

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

A syntactically valid sequence of operations in a given algebraic specification is called a *term*. For example, *new.push*(1).*push*(2).*pop* is a term in the class of integer stacks above, but *new.push*(1).*push*(2).*top.pop* is not.

A term without variables is known as a *ground term*. In this paper, we will only consider ground terms because actual test cases do not involve variables.

Suppose a ground term u_0 contains a sub-term v_0 that is an instance of the left-hand side v of the axiom a: $v = v'$. Suppose further that, after replacing the sub-term v_0 by the corresponding instance of the right-hand side v', we obtain a new ground term u_1. Then, we say that the ground term u_0 can be *transformed* into (or *rewritten* as) the ground term u_1 using the axiom a as a *rewriting rule*. We denote this process by the notation $u_0 ==(a)==> u_1$. For example,

new.push(1).*push*(2).*pop* ==(a_4)==> *new*. *push*(1).

A term is said to be a *normal form* if it cannot be transformed using any axiom in the specification. For instance, *new.push*(1).*push*(2) is a normal form but *new.push*(1).*push*(2).*pop* is not.

A fundamental pair is defined as a pair of terms generated by substituting all the variables on both sides of an axiom by their normal forms.

An algebraic specification is said to be *canonical* if any ground term will be transformed by the axioms in the specification into a unique normal form. For instance, the specification in Example 1 is canonical, but it is not if we change the axiom a_4 to $S = S.push(N).pop$. We only discuss canonical specifications in this paper.

Two ground terms u_1 and u_2 in a canonical specification are said to be *equivalent* (denoted by $u_1 \sim u_2$) if they have same normal form. For instance,

new.push(1).*push*(2).*pop* ~ *new.push*(2).*pop.push*(1).

In a given class *C*, operations or methods generating new objects of *C* are called *creators*. Operations or methods changing the values of attributes of object in *C* are called *constructors* or *transformers*. The difference between constructors and transformers is that *constructors* can appear in normal forms but *transformers* cannot. Operations that only output the values of attributes of objects in *C* are called *observers*. In Example 1, for instance, the operation *new* is a creator, *_.push*(*N*) is a constructor, *_.pop* is a transformer, and *_.empty* and *_.top* are observers.

III. Pattern Analysis of Normal Forms

A. Properties of Normal Forms

The following important properties of normal forms follow directly from the definitions of terms, axioms, normal forms, constructors, and transformers:

Proposition 1. For any sub-term t' of a term t, if t' is not a normal form, then t is not a normal form either.

Proposition 2. A term is not a normal form if any of its sub-terms matches the left-hand side of an axiom.

Proposition 3. A term is a normal form if none of its sub-terms matches the left-hand side of any axiom.

B. A Case Study

We begin the investigation of structural patterns of normal forms with a case study.

Example 2. The following is an algebraic specification for a class *Book* in a simplified library system, where a book can only be reserved by one borrower. For ease of presentation, the side effects of some operations are omitted and some attributes are also omitted.

module *BOOK*
class *Book*
import classes
String
Integer
Loc = {*onShelf*, *atCounter*, *onLoan*, *onLoan&reserved*}
operations
newBook(_, _): *String Integer* → *Book*
// First parameter captures the name of the book; second parameter captures the call number
_.*name*: *Book* → *String*
_.*number*: *Book* → *Integer*
_.*location*: *Book* → *Loc*
_.*borrow*: *Book* → *Book*
_.*return*: *Book* → *Book*
variables
B: *Book*
S: *String*
I: *Integer*
axioms
a_{11}: *newBook*(*S*, *I*).*name* = *S*
a_{12}: *newBook*(*S*, *I*).*number* = *I*
a_{13}: *newBook*(*S*, *I*).*location* = *onShelf*
a_{21}: *B*.*borrow*.*name* = *B*.*name*
a_{22}: *B*.*borrow*.*number* = *B*.*number*
a_{23}: *B*.*borrow* = *B if B*.*location* = *onLoan&reserved*
a_{24}: *B*.*borrow*.*location* = *onLoan* *if B*.*location* ∈ {*onShelf*, *atCounter*}
a_{25}: *B*.*borrow* = *newBook*(*B*.*name*, *B*.*number*).*borrow* *if B*.*location* ∈ {*onShelf*, *atCounter*} *and B* ≠ *newBook*(_, _)
a_{26}: *B*.*borrow*.*location* = *onLoan&reserved* *if B*.*location* = *onLoan*
a_{31}: *B*.*return*.*name* = *B*.*name*
a_{32}: *B*.*return*.*number* = *B*.*number*
a_{33}: *B*.*return* = *B* *if B*.*location* ∈ {*onShelf*, *atCounter*}
a_{34}: *B*.*return*.*location* = *onShelf* *if B*.*location* = *onLoan*
a_{35}: *B*.*return*.*location* = *atCounter* *if B*.*location* = *onLoan&reserved*
a_{41}: *B*.*borrow*.*return* = *B* *if B*.*location* = *onShelf*

The normal form patterns of the class *Book* are analyzed as follows. For simplicity, we will use *nw*, *b*, and *r* to denote *newBook*(*S*, *I*), *borrow*, and *return*, respectively. We will also use b^k to denote *k* consecutive *borrow* operations, r^j to denote *j* consecutive *return* operations, and "..." to denote a finite sequence of operations.

Analysis.

(1) ***nw* is a normal form pattern** because it cannot be rewritten according to any axiom.[2]

(2) ***nw.b* is also a normal form pattern** because it cannot be rewritten according to any axiom.

(3) ***nw.b*^{2} is also a normal form pattern** because it cannot be rewritten according to any axiom.

(4) According to axioms a_{13}, a_{24}, and a_{26}, *nw*.b^2.*location* can be transformed into *onLoan&reserved*. As a result, according to axiom a_{23}, the term *nw*.b^3 can be transformed into *nw*.b^2. Thus *nw*.b^3 is not a normal form pattern.

(5) Hence, *nw*.b^3... is not a normal form pattern.

(6) *nw.r* is not a normal form pattern because *nw.r* ==(a_{13}, a_{33})==> *nw*.

(7) Hence, *nw.r*... is not a normal form pattern.

(8) *nw.b.r* is not a normal form pattern because *nw.b.r* ==(a_{13}, a_{41})==> *nw*.

(9) Hence, *nw.b.r*... is not a normal form pattern.

(10) ***nw.b*^{2}*.r* is a normal form pattern** because it cannot be rewritten according to any axiom.

(11) *nw*.b^2.*r.b* is not a normal form pattern because *nw*.b^2.*r.b* ==(a_{13}, a_{24}, a_{26}, a_{35}, a_{25})==> *nw.b*.

(12) Hence, *nw*.b^2.*r.b*... is not a normal form pattern.

(13) *nw*.b^2.r^2 is not a normal form pattern because *nw*.b^2.r^2 ==(a_{13}, a_{24}, a_{26}, a_{35}, a_{33})==> *nw*.b^2.*r*.

(14) Hence, *nw*.b^2.r^2... is not a normal form pattern.

Notes.

(a) Each of the four *normal form patterns* (*nw*, *nw.b*, *nw*.b^2, *nw*.b^2.*r*) represents a set of normal forms of the class *Book*. For example,

$$nw.b^2.r = \{newBook(S, I).borrow.borrow.return \mid S \in String, I \in Integer\}.$$

(b) Both *borrow* and *return* are constructors. There is no transformer in the class *Book*.

(c) The above analysis covers all possible terms that consist of only a creator, or a creator followed only by constructors but not transformers.

(d) The left-hand sides of the axioms a_{11}, a_{12}, a_{13}, a_{21}, a_{22}, a_{24}, a_{26}, a_{31}, a_{32}, a_{34}, and a_{35} end with observers. They are transformed into normal forms representing the values of the attributes of the class *Book*.

(e) The values of the *location* attribute of the normal forms patterns *nw*, *nw.b*, *nw*.b^2, and *nw*.b^2.*r* are *onShelf*, *onLoan*, *onLoan&reserved*, and *atCounter*, respectively.

[2] The term or any part of it does not match the left-hand side of any axiom.

From the above analysis and notes, we observe that the class *Book* has four and only four normal form patterns:

newBook(*S*, *I*), *newBook*(*S*, *I*).*borrow*,
newBook(*S*, *I*).*borrow*.*borrow*, and
newBook(*S*, *I*).*borrow*.*borrow*.*return*.

In Example 2, even though *b* and *r* are constructors of the class *Book*, $nw.b^3$, *nw.b.r*, $nw.b^2.r^2$, $nw.b^k.r^j...b^{k'}.r^{j'}.b^{k''}$ (k, ..., k', k'', i, ..., $i' \geq 1$), and $nw.b^k.r^j...b^{k'}.r^{j'}.b^{k''}.r^{j''}$ (k, ..., k', k'', i, ..., i', $i'' \geq 1$) are not normal forms. Hence, we have:

Proposition 4. If a ground term *T* is a normal form of a class *C*, then *T* consists of either a creator of *C*, or a creator of *C* following only by constructors but not transformers of *C*. However, the converse does not hold.[3]

C. More Examples

Example 3. The *normal form patterns* of the class *IntStack* in Example 1 are as follows:

t_0: *new*,
t_1: *new*.*push*(N_1),
t_2: *new*.*push*(N_1).*push*(N_2),
t_3: *new*.*push*(N_1).*push*(N_2).*push*(N_3),
t_4: *new*.*push*(N_1).*push*(N_2).*push*(N_3).*push*(N_4), ...

Example 4. The following, adapted from [3], is a simplified algebraic specification of a class *SavAcct*1 of savings accounts in a banking system. For ease of presentation, some attributes are omitted and the side effects of some operations are ignored.

class *SavAcct*1
import classes
Money
String
operations
newAc(_, _, _): *String String Money* → *SavAcct*1
// The input parameters denote the customer name, address, and account balance, respectively
name: *SavAcct*1 → *String*
address: *SavAcct*1 → *String*
balance: *SavAcct*1 → *Money*
setAddress(_): *SavAcct*1 *String* → *SavAcct*1
credit(_): *SavAcct*1 *Money* → *SavAcct*1
debit(_): *SavAcct*1 *Money* → *SavAcct*1
variables
S, *S*′: *String*
A: *SavAcct*1
M: *Money*
axioms
a_1: *newAc*(*S*, *S*′, *M*).*name* = *S*
a_2: *newAc*(*S*, *S*′, *M*).*address* = *S*′
a_3: *newAc*(*S*, *S*′, *M*).*balance* = *M*
a_4: *A*.*credit*(*M*).*balance* = *A*.*balance* + *M*
a_5: *A*.*debit*(*M*).*balance* = *A*.*balance* – *M*
if A.*balance* ≥ *M*
a_6: *A*.*debit*(*M*).*balance* = *A*.*balance*
if A.*balance* < *M*
a_7: *A*.*setAddress*(*S*).*balance* = *A*.*balance*
a_8: *A*.*credit*(*M*).*address* = *A*.*address*
a_9: *A*.*debit*(*M*).*address* = *A*.*address*
a_{10}: *A*.*setAddress*(*S*).*address* = *S*
a_{11}: *A*.*credit*(*M*).*name* = *A*.*name*
a_{12}: *A*.*debit*(*M*).*name* = *A*.*name*
a_{13}: *A*.*setAddress*(*S*).*name* = *A*.*name*

For simplicity, we will use *new*, *c*, *d*, and *st* to denote *newAc*, *credit*, *debit*, and *setAddress*, respectively. The *normal form patterns* are:

new(*S*, *S*′, *M*), *new*(*S*, *S*′, *M*).*c*(*M*′),
new(*S*, *S*′, *M*).*d*(*M*′), *new*(*S*, *S*′, *M*).*st*(*S*′′),
new(*S*, *S*′, *M*).$op_1.op_2...op_k$, ...

where each op_i is the pattern $c(M_i)$, $d(M_i)$, or $st(S_i)$.

The notion of normal forms depends greatly on the system of axioms in the algebraic specification of a given class. When the axioms are changed, the set of normal forms may also be different. Consider, for instance, the simplified algebraic specification of the class *SavAcct*1 of savings accounts in Example 4. Suppose we revise the axioms to produce a new specification *SavAcct*2 as shown in Example 5 below. The set of normal forms will be significantly changed.

Example 5. The following is a simplified algebraic specification of another class *SavAcct*2 of savings accounts in a banking system. Again, some attributes and side effects are omitted.

class *SavAcct*2
import classes
Money
String
operations
newAc (_, _, _): *String String Money* → *SavAcct*2
// The input parameters denote the customer name, address, and account balance, respectively
name: *SavAcct*2 → *String*
address: *SavAcct*2 → *String*
balance: *SavAcct*2 → *Money*
setAddress(_): *SavAcct*2 *String* → *SavAcct*2
credit(_): *SavAcct*2 *Money* → *SavAcct*2
debit(_): *SavAcct*2 *Money* → *SavAcct*2
variables
S, *S*′: *String*
A: *SavAcct*2
M, *M*′: *Money*
axioms
a_1: *newAc*(*S*, *S*′, *M*).*name* = *S*

[3] This does not contradict the statement "*constructors* can appear in normal forms but *transformers* cannot" in Section 2.

a_2: $newAc(S, S', M).address = S'$
a_3: $newAc(S, S', M).balance = M$
a_4: $A.credit(M).balance = A.balance + M$
a_5: $A.debit(M).balance = A.balance - M$
$\quad if\ A.balance \geq M$
a_6: $A.setAddress(S).balance = A.balance$
a_7: $A.credit(M).address = A.address$
a_8: $A.debit(M).address = A.address$
$\quad if\ A.balance \geq M$
$\quad //$ The condition makes a_8 independent of a_{16}
a_9: $A.setAddress(S).address = S$
a_{10}: $A.credit(M).name = A.name$
a_{11}: $A.debit(M).name = A.name$
$\quad if\ A.balance \geq M$
$\quad //$ The condition makes a_{11} independent of a_{16}
a_{12}: $A.setAddress(S).name = A.name$
a_{13}: $A.credit(M).debit(M') = A.credit(M - M')$
$\quad if\ M > M'$
a_{14}: $A.credit(M).debit(M') = A.debit(M' - M)$
$\quad if\ M < M' \wedge A.balance \geq M'-M$
a_{15}: $A.credit(M).debit(M') = A\ if\ M = M'$
a_{16}: $A.debit(M) = A\ if\ A.balance < M$
a_{17}: $A.debit(M).credit(M') = A.credit(M' - M)$
$\quad if\ M \leq A.balance \wedge M < M'$
a_{18}: $A.debit(M).credit(M') = A.debit(M - M')$
$\quad if\ M \leq A.balance \wedge M > M'$
a_{19}: $A.debit(M).credit(M') = A$
$\quad if\ M \leq A.balance \wedge M = M'$
a_{20}: $A.credit(M).credit(M') = A.credit(M + M')$
a_{21}: $A.debit(M).debit(M') = A.debit(M + M')$
$\quad if\ M + M' \leq A.balance$
a_{22}: $A.setAddress(S_1).setAddress(S)$
$\quad = A.setAddress(S)$

We will again use *new*, *c*, *d*, and *st* to denote *newAc*, *credit*, *debit*, and *setAddress*, respectively. The *normal form patterns* are:

$new(S, S', M)$,
$new(S, S', M).c(M')$,
$new(S, S', M).d(M')$,
$new(S, S', M).st(S'')$,
$new(S, S', M).st(S_1).op_1.st(S_2).op_2...st(S_k).op_k$,
$new(S, S', M).st(S_1).op_1.st(S_2).op_2...st(S_k)$,
$new(S, S', M).op_1.st(S_1).op_2.st(S_2)...op_k.st(S_k)$,
$new(S, S', M).op_1.st(S_1).op_2.st(S_2)...op_k$, ...

where op_i is one of the patterns $c(M_i)$ or $d(M_i)$. Notice that, although *c* and *d* are constructors, ground terms of the form $new.op_1.op_2...op_k$ (where $k > 1$ and each op_i is one of the patterns $c(M_i)$ or $d(M_i)$) are not normal forms as they can be rewritten as one of the patterns $new.c(M)$ or $new.d(M)$.

IV. Generating Normal Forms from Patterns

From Proposition 4 and the examples above, we realize that the generation of normal form patterns is non-trivial. This section presents an algorithm GNF and a corresponding tool to aid the generation process. The algorithm requires the following additional definitions.

A. Principal Operators, Normal Forms, and Axioms

It is obvious from the definitions of creators, constructors, transformers, and observers that, by using only the syntax declaration in the algebraic specification of a given class, we can distinguish creators and observers from each other, and also distinguish them from constructors and transformers. We cannot, however, distinguish constructors and transformers from each other using only the syntax declaration. We need to use the semantic definitions in the algebraic specification for this purpose. For this reason, we propose to bundle constructors and transformers together, thus:

> **Definition 1 (Principal Operator).** We refer to a constructor or transformer in the algebraic specification of a class *C* as a *principal operator* of *C*.

In Example 1, the normal form of the term $new.push(1).push(2).top$ is 2. It is, in fact, a value of the attribute *top* of the class *IntStack*. We do not consider such kind of simple normal form in this paper. We only consider *principal normal forms*, which are defined as follows:

> **Definition 2 (Principal Normal Form).** A normal form in the algebraic specification of a class *C* is known as a *principal normal form* of *C* if it ends with a principal operator of *C*.

Obviously, a principal normal form of *C* returns an object of class *C*.

> **Definition 3 (Principal Axiom).** An axiom in the algebraic specification of a class *C* is called a *principal axiom* if its left-hand side ends with a principal operator of C, and is called an *attributive axiom* if its left-hand side ends with an observer of C.

In example 1, for instance, a_4 is a principal axiom and a_6 is an attributive axiom.

> **Definition 4 (Length).** The number of operators in a given normal form is known as the *length* of the normal form.

In Example 2, for instance, the length of the normal form $newBook(S, I).borrow.borrow.return$ is 4.

B. Tree model

In this subsection, we set up a tree model for principal normal forms.

> **Definition 5 (Principal Normal Form Pattern Tree).** Suppose the algebraic specification of a given class *C* contains creators $cr_1, cr_2, \ldots, cr_i$, and principal operators $pr_1, pr_2, \ldots, pr_j$. We construct a tree T_C using the following procedure:
>
> (i) Construct a node "○" as the root, and take $cr_1, cr_2, \ldots, cr_i$ as child nodes of "○".

(ii) For each node cr_k (k = 1, 2, …, i), take pr_1, pr_2, …, pr_j as its child nodes.

(iii) For each node pr_m (m = 1, 2, …, j), take pr_1, pr_2, …, pr_j again as its child nodes, and so on.

(iv) The sequence of operators in the nodes along a path from the root "○" to any node n_r is a term of C. We refer to this term as an *associated term* of the node n_r. For any given node n_s, if its associated term t_s is not a normal form, we can conclude from Proposition 1 that the associated terms for all child nodes of n_s are all not normal forms either. Hence, we remove n_s and any of its child nodes from the tree T_C. In other words, we *prune* the subtree with root n_s. The associated terms of the remaining nodes are normal forms.

We refer to the tree T_C constructed by the above procedure with pruning as the *principal normal form pattern tree* of class C. In fact, it is a model of the structural pattern of the principal normal forms of C.

We note also the following:

(a) In step (4) of Definition 5, we can use Proposition 2 to determine that an associated term t of a node is not a normal form, and use Proposition 3 to determine that an associated term t is a normal form. Since we construct the tree T_C top-down, we need only check whether the *current* associated term t matches with the left-hand sides of axioms. We need not check this for the proper sub-terms of t, as all the proper sub-terms of t have already been verified to be normal forms.

(b) Step (4) in Definition 5 provides a *pruning rule* for principal normal form pattern trees.

(c) A principal normal form pattern tree may be finite or infinite. In the following figures, "…" denotes potentially infinite subtrees.

Example 6. A principal normal form pattern tree for the class *Book* in Example 2 is shown in Figure 1.

Example 7. A principal normal form pattern tree for the class *IntStack* in Example 3 is shown in Figure 2.

Example 8. A principal normal form pattern tree for the class *SavAcct*1 in Example 4 is shown in Figure 3.

Example 9. A principal normal form pattern tree for the class *SavAcc*2 in Example 5 is shown in Figure 4.

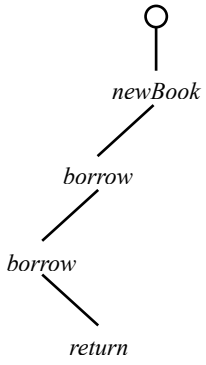

Figure 1. Principal normal form pattern tree for the class *Book* in Example 2.

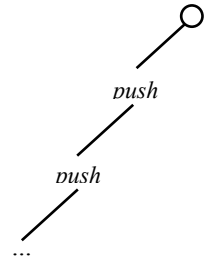

Figure 2. Principal normal form pattern tree for the class *IntStack* in Example 3.

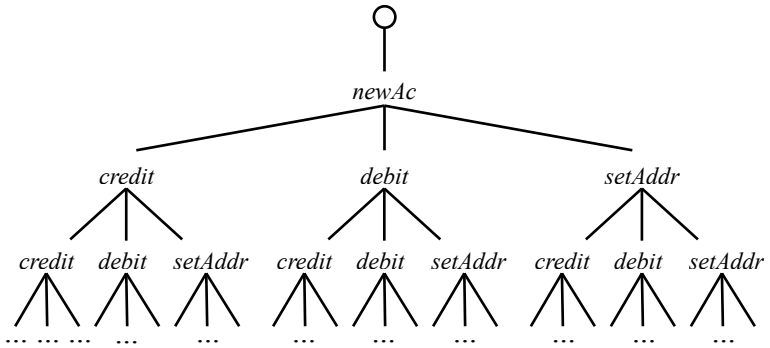

Figure 3. Principal normal form pattern tree for the class *SavAcct*1 in Example 4.

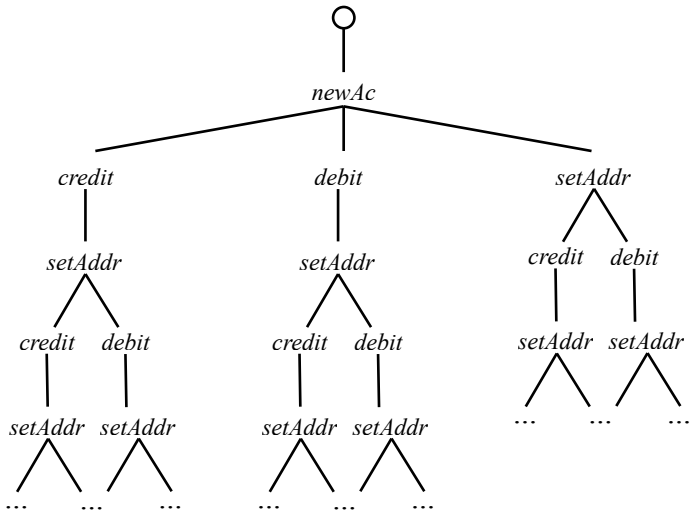

Figure 4. Principal normal form pattern tree for the class *SavAcct*2 in Example 5.

C. Algorithm GNF

We can now describe Algorithm *GNF* to aid the ***G***eneration of principal ***N***ormal ***F***orm patterns from the algebraic specification of a given class *C*. The fundamental concept behind the algorithm is as follows: In order to generate principal normal form patterns for class *C*, we need only enumerate the associated terms by traversing the nodes in the principal normal form pattern tree of class C using a top-down breadth-first strategy in conjunction with the pruning technique described in Section 4.3. On the other hand, we need not actually construct and store the tree structure. The tree model simply helps us visualize the (traversing) strategy and the (pruning) technique. We need not even refer to the notions of *trees*, *paths*, *nodes*, *or associated terms* in the actual algorithm.

Algorithm *GNF*.

(1) Analyze the syntax declaration of the algebraic specification of class *C* (or interact with the requirements analyst) to determine the set *CR* of creators and the set *CT* of principal operators. (They may contain parameters.) Obtain the set *AX* of principal axioms of the algebraic specification.

(2) Ask the requirements analyst or tester to specify the maximum length *Lmax* of the target principal normal forms, where $Lmax \geq 1$.

(3) Let *PPNF* denote the set of all *Potential Principal Normal Form patterns* generated so far, *PPNF*0 denote the set of the *Potential Principal Normal Form patterns* generated in the last execution of step (4), and *PPNF*1 denote the set of the *Potential Principal Normal Form patterns* that will be generated in the next execution of step (4). Initialize *PPNF to CR* [4]*, PPNF0 to CR, and PPNF1 to* $\varnothing$*, where the* parameters for each $cr \in CR$ are bound to the corresponding variables. Let *L* denote the maximum length of the principal normal forms in the current *PPNF*. Initialize *L* to 1.

(4) For each $t \in PPNF0$, do {
  For each $ct \in CT$ (where the parameters in *ct* are all be bound to corresponding variables), do {
    If *t.ct* can *match* [5] the left-hand side of an axiom in *AX*
      skip the term *t.ct*; [6]
    Else, $PPNF1 = PPNF1 \cup \{ t.ct \}$; [7]
    };
  };
  If $PPNF1 \neq \varnothing$, {
    $L = L + 1$;
    $PPNF = PPNF \cup PPNF1$;
    };
  $PPNF0 = PPNF1$;
  $PPNF1 = \varnothing$;

(5) If $L \neq Lmax$ and $PPNF0 \neq \varnothing$, go to (4);

(6) Interact with the requirements analyst or tester to review the *PPNF* and produce a set *FPNF* of *Final Principal Normal Form patterns* of the given class *C*.

(7) Output the *FPNF*.

---

[4] Obviously, creators must be principal normal forms.

[5] We need not check whether each proper sub-term of *t.ct* can match with the left-hand side of an axiom in *AX*, as it has been checked in a previous execution of step (4).

[6] This is equivalent to pruning in the tree model. The term *t.ct* here is not a normal form.

[7] According to Proposition 3, the term *t.ct* here must be a principal normal form.

We note that the concept of *term matching* in step (4) of Algorithm *GNF* is also known as *unification* in artificial intelligence. Effective algorithms for unification can be found in standard texts on logic programming. In fact, if Prolog were used to implement Algorithm *GNF*, the term matching process would be rather simple. The drawback, of course, would be the user interface.

In step (2) of Algorithm *GNF*, if the target maximum length *Lmax* of principal normal forms specified by the requirements analyst or tester is greater than the actual maximum length of principal normal forms of the given class [8], the execution of Algorithm *GNF* will be automatically terminated by the condition $PPNF0 = \varnothing$ in step (5). If the actual maximum length of principal normal forms of the given class is infinite, or is larger than the target maximum length *Lmax*, then Algorithm GNF will enumerate principal normal forms from a length of 1 to the given length *Lmax*.

For a given class, the numbers of creators, principal operators, and principal axioms are not large in general, and hence the number of loops in Algorithm *GNF* is not large. Furthermore, the pruning technique in *GNF* significantly improves the efficiency of the algorithm.

D. Computer-Aided Tool GNF

A prototype computer-aided tool *GNF* for the algorithm has been implemented using Java in *JDK*-6- *windows-i*586 on *Eclipse-SDK*-3.2.1-*win*32 under *Microsoft Windows XP*. It contains eight main class modules, namely *TxtFile*, *FileTransformer*, *XmlFile*, *Specification*, *StringTransformer*, *ConditionHandler*, *NormalformGenerator*, and *OutputNF*. The algebraic specifications are expressed in *XML*.

Experiments have been conducted on the classes *Book*, *IntStack*, *SavAcct*1, and *SavAcct*2, and have generated 4, 20, 40, and 30 principal normal forms, respectively. The experimental results are shown in Figures 1 to 4, respectively, and confirm the analyses in Examples 2 to 5, respectively. More details can be found in [11].

V. Limitation and Future Work

In this paper, we assume that any given specification of a class must be canonical with proper imports. We have explained in our earlier paper [4] that this assumption is reasonable.

As future work, we will consider the optimization of Algorithm GNF, an implementation of this optimized algorithm in Prolog, and more experiments on the new implementation.

VI. Conclusion

The testing of equivalent fundamental pairs is an effective approach for object-oriented class level testing. However, in the previous algorithm for selecting equivalent fundamental pairs, normal forms have to be supplied manually. We find that the structural patterns of normal forms and their generation are nontrivial. Hence, a new algorithm is necessary for improving the coverage of normal forms and the efficiency of testing fundamental pairs.

Based on normal form patterns, a tree model, and a pruning technique, this paper presents an algorithm to generate representative normal forms. A corresponding computer-aided tool has been developed. Experimentation shows that it can cover patterns of normal forms within a user-defined maximum length. The pruning technique based on the tree model improves the time and space performances of the tool.

[8] In Example 2, for instance, the actual maximum length of the principal normal forms of the class *Book* is 4.

Acknowledgment

Our thanks are due to Miss Mei-Ling He for her contributions to the implementation and experimentation of the prototype tool *GNF*.

References

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

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

[3] H. Y. Chen, T. H. Tse, F. T. Chan, and T. Y. Chen. In black and white: an integrated approach to class-level testing of object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 7 (3): 250–295, 1998.

[4] H. Y. Chen, T. H. Tse, and T. Y. Chen. TACCLE: a methodology for object-oriented software testing at the class and cluster levels. *ACM Transactions on Software Engineering and Methodology*, 10 (1): 56– 109, 2001.

[5] H. Y. Chen, T. H. Tse, and Y. T. Deng. ROCS: an object-oriented class-level testing system based on the relevant observable contexts technique. *Information and Software Technology*, 42 (10): 677–686, 2000.

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

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

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

[9] J. D. Gannon, P. R. McMullin, and R. Hamlet. Data abstraction, implementation, specification, and testing. *ACM Transactions on Programming Languages and Systems*, 3

(3): 211–223, 1981.

[10] J. A. Goguen and R. Diaconescu. Towards an algebraic semantics for the object paradigm. In *Recent Trends in Data Type Specification: Proceedings of the 9th International Workshop on Specification of Abstract Data Types*, volume 785 of Lecture Notes in Computer Science, pages 1–29. Springer, Berlin, Germany, 1994.

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

[12] M.-L. He. *The Investigation on a Semi-Automatic Tool for Generating Normal Forms for Object-Oriented Software Testing at Class Level*. Master's Thesis, Department of Computer Science, Jinan University, Guangzhou, China, 2008.

[13] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Luettgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward, and H. Zedan. Using formal specifications to support testing. *ACM Computing Surveys*, 41 (2): 1–76, 2009.

[14] P. Jalote. Specification and testing of abstract data types. In *Proceedings of the 7th Annual International Computer Software and Applications Conference* (*COMPSAC 1983*), pages 508–511. IEEE Computer Society Press, New York, NY, 1983.

[15] L. Mariani and M. Pezze. Testing object-oriented software. In *Emerging Methods, Technologies, and Process Management in Software Engineering*, A. De Lucia, F. Ferrucci, G. Tortora, and M. Tucci (editors), pages 85–105. Wiley, New York, NY, 2008.

[16] A. J. H. Simons. JWalk: a tool for lazy, systematic testing of Java classes by design introspection and user interaction. *Automated Software Engineering*, 14 (4): 369–418, 2007.

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

[18] T. H. Tse, F. C. M. Lau, W. K. Chan, P. C. K. Liu, and C. K. F. Luk. Testing object-oriented industrial software without precise oracles or results. *Communications of the ACM*, 50 (8): 78–85, 2007.

[19] D. A. Wolfram and J. A. Goguen. A sheaf semantics for FOOPS expressions. In *Object-Based Concurrent Programming: Proceedings of the ECOOP 1991 Workshop*, volume 612 of Lecture Notes in Computer Science, pages 81–98. Springer, Berlin, Germany, 1992.

[20] B. Yu, L. Kong, Y. Zhang, and H. Zhu. Testing Java components based on algebraic specifications. In *Proceedings of the 1st International Conference on Software Testing, Verification, and Validation* (*ICST 2008*), pages 190–199. IEEE Computer Society Press, Los Alamitos, CA, 2008.

[21] H. Zhu. A note on test oracles and semantics of algebraic specifications. In *Proceedings of the 3rd International Conference on Quality Software* (*QSIC 2003*), pages 91–98. IEEE Computer Society Press, Los Alamitos, CA, 2003.