Prototypes and Initial Experimentation on the Tools of the TACCLE Methodology *

HUO YAN CHEN
Jinan University, China
and
T. H. TSE
The University of Hong Kong

Object-oriented programming poses new challenges to software testing, since objects may interact with one another with unforeseen combinations and invocations, which are much more complex to simulate and test than the hierarchy of modules in conventional programs. We would like to use our experiences in formal techniques and software testing to solve the problem. We propose a new methodology for object-oriented software testing at the class and cluster levels. It is known as TACCLE, which stands for object-oriented software **T**esting **A**t the **C**lass and **C**luster **L**evels [2]. Potentially problematic scenarios such as the non-equivalence of objects at the class level and interactions at the cluster level can be identified from the formal specifications and generated as test cases.

Four prototype tools, known as DOE, GAN, GCS, and ESI, have been developed to support the methodology. This report summarizes the implementation and the initial experimentation on the prototypes.

1 DOE

DOE is a semi-automatic tool to help determine the observational equivalence of objects. A prototype of DOE has been developed using Borland C++. It consists of five modules: *pigeonC.c*, *pigeonC.h*, *subLib.c parser.c*, and *drg.c*. The module *pigeonC.c* serves as the main module of the prototype. It reads the C++ source code for a given class under test, allocates memory for the program, pre-scans it, and calls and coordinates other modules to perform the tasks. The module *pigeonC.h* defines the main data structure, and *subLib.c* defines the interfaces to internal library functions. The module *parser.c* consists of a lexical analyzer and a recursive descent parser. The module *drg.c* constructs a data member relevance graph, traverses executable paths by backtracking, and generates and executes the corresponding relevant observable contexts for any two given objects.

Some experimental results on the prototype have been presented in our earlier paper [1].

---

* 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.

Authors' addresses: Huo Yan Chen, Department of Computer Science, Jinan University, Guangzhou 510632, China. Email: "tchy@maina.jnu.edu.cn". (Part of the research was performed when Chen was on leave at the University of Hong Kong.) T. H. Tse (**Contact Author**), Department of Computer Science and Information Systems, the University of Hong Kong, Pokfulam Road, Hong Kong. Email: "tse@csis.hku.hk". (Part of the research was performed when Tse was on leave at the Vocational Training Council, Hong Kong.)

2 GAN

GAN is a semi-automatic tool to help generate attributively non-equivalent terms as test cases.

2.1 The implementation of GAN

In the GAN approach, rewriting technique is used in the construction of a state-transition diagram from the algebraic specification of the given class. This can easily be implemented using Prolog. Hence, the prototype of GAN has been developed using Arity/Prolog32 in Windows 98.

First, the given algebraic specification must be represented as a Prolog structure. For example:

(a) In the class *intStack* of the stack of non-negative integers, for instance, the operation *push*(*x*) can be represented as follows:

operation(*constructor*, *push*(*x*) *if x* >= 0, [*intStack*], [*integer*], *intStack*, _),

which means that *push*(*x*) under the condition *x* >= 0 is a *constructor* of the class. Its main parameter is an object of the class *intStack*. Its other parameter is an *integer*. The output class is *intStack*.

(b) The Prolog structure

operation(*observer*, *height*, [*intStack*], [], *integer*,
[*s#height* = 0, *s#height* >= 1 and *s#height* = <9, *s#height* = 10])

denotes that *height* is an *observer* of the class. Its main parameter is an object of the class *intStack*. There is no other parameter. The output class (type) is *integer*. The output domain is partitioned into three sub-domains {0}, {1, 2, …, 9}, and {10}.

(c) The axiom "*S.push*(*X*).*height* = *S.height* + 1 *if s.height* < 10 *and X* >= 0" is denoted by the Prolog structure

axiom(*s#push*(*x*)*#height*, *s#height*+1, *s#height* < 10 *and x* >= 0).

The top-level module of the prototype is as follows.

```
:- op(250, yfx, #).
:- op(910, xfy, if).
:- op(740, yfx, and).

generateANE :-
        % ANE stands for Attributive Non-Equivalence.
        % Construct a state-transition diagram from a given algebraic specification.
        % Search the paths from the initial node n_0 to other nodes n_i.
        % Any two operation sequences (terms) on two different paths
        %       from n_0 to the same n_i will be attributively non-equivalent.
    nl, nl, write('Please input the name of the file containing the given specification in Prolog
        form: '), read(AlgProFl), nl, nl,
            % See the file algSpc.ari for the format of the algebraic specification.
    reconsult(AlgProFl),
    addNodes, addArcs,
        % Construct and insert nodes and arcs into the database.
        % The format of a node is node(Number, State), where State is a list.
```

```
        % The format of an arc is
        %       arc(StartNodeNumber, EndNodeNumber, Operation, ConditionsList, 0).
        % The parameter 0 of arc denotes the number of path traversals.
    write('The state-transition diagram has been generated.
        Do you want to review it (y/n)? '), read(ReviewSTD), nl, nl,
    ( ReviewSTD = y, listing(node/2), nl, nl,
      write('Enter "c." to continue:  '), read(XX), nl, nl,
      listing(arc/5)
      ;
      ReviewSTD \= y  ), nl,
    generatePaths, nl, nl,
        % The format of the path is
        %       path(StartNodeNumber, EndNodeNumber, [Operation if ConditionsList | T]),
        %       where "if ConditionsList" may be absent.
    write('The paths have been generated.
        Do you want to review them (y/n)? '), read(ReviewPath), nl, nl,
    ( ReviewPath = y,
      ( clause(path(St, En, OpCs), _),
        write(path(St, En, OpCs)), nl, nl,
        write('Do you want to review the next path (y/n)? '), read(W), W = n
        ;
        write('There is no more path.')  )
      ;
      ReviewPath \= y, !  ), nl, nl,
    write('Please input the name of the file for storing the state-transition diagram
        and the paths:  '), read(STDiagram), nl, nl,
    file_list(STDiagram, [node/2, arc/5, path/3]).

generatePaths :-
    nl, write('Please determine the global upper-bound for the number of iterations
          for any cyclic arc:  '),
    read(Itr), assertz((iterNoMax(Itr))), nl,
    node(N, St),
    N \= 0,
    [!  nl, write('Please determine the upper-bound for the number of paths
                from the initial node n_0 to the current node '), write(N), write(':  '),
      read(UpBound), assertz((pathNoMax(UpBound))),
      pathsToN(N)  !], fail
    ;
    retract(iterNoMax(_)).

addNodes :-
    assertz((workList([[]]))),
    operation(observer, _, _, _, _, SbDnList),
    retract(workList(WkList)),
    combine(WkList, SbDnList, List),
    assertz((workList(List))), fail
    ;
    workList(NodesList),
    assertzNodes(NodesList), retract(workList(_)).
```

```
addArcs :-
    node(N1, St), N1 > 0,
    operation(Tp, Op1, _, _, _, _),
    [!  (  Tp = constructor
           ;
           Tp = transformer  ),
        (  Op1 = (Op if CnOp),
           CnOpList = [CnOp]
           ;
           Op1 \= (_ if _),
           Op = Op1, CnOpList = []  ),
        transferTo(N1, St, CnOpList, Op, St2Cons),
           % The format of St2Cons is [S1 if C1, ...].
        assertzLinks(N1, Op, St2Cons)  !], fail.
addArcs :- node(0, []), addArc0.
```

2.2 Initial Experimentation on GAN

We have experimented with some examples on the prototype as follows.

Example 1

The following is an algebraic specification of the class *IntStack* of the stack of non-negative integers with a maximum size of 10.

class *IntStack*
imported classes *Int Bool*
operations
new: $\rightarrow IntStack$
_.empty: $IntStack \rightarrow Bool$
.push(): $IntStack\ Int \rightarrow IntStack$
_.pop: $IntStack \rightarrow IntStack$
_.top: $IntStack \rightarrow Int \cup \{nil\}$
_.height: $IntStack \rightarrow Int$
variables
S: *IntStack*
N: *Int*
axioms
a_1: $new.empty = true$
a_2: $new.height = 0$
a_3: $new.top = nil$
a_4: $new.pop = new$
a_5: $S.push(N).empty = false$
a_6: $S.push(N).height = S.height + 1$ if $S.height < 10$
a_7: $S.push(N).top = N$ if $S.height < 10$
a_8: $S.push(N) = S$ if $S.height = 10$
a_9: $S.push(N).pop = S$ if $S.height < 10$

An implementation of the class is as follows.

```
#include <iostream.h>
#define NIL 0
enum bool { false, true };

class intStack
    {
    private:
        /*  intStack consists of 2 data members:  */
        int array[SIZE];
        int ht;
    public:
        void newStack( );
        bool empty( );
        int height( );
        int top( );
        void push(int i);
        void pop( );
    };

void intStack :: newStack( )
    {
    ht = 0;
    for  ( int  j = 1;  j < = 10;  j++ )
    array[j] = NIL;
    }

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

int intStack :: height( )
    {
    return  ht;
    }

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

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

int intStack :: top( )
    {
    if  (ht > 0)  return  array[ht];
    else  return  NIL;
    }
```

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

n_0 (initial node),
n_1 = (*empty* = *true*, *top* = *nil*, *ht* = 0),
n_2 = (*empty* = *false*, *top* ≥ 0, 1 ≤ *ht* ≤ 9),
n_3 = (*empty* = *false*, *top* ≥ 0, *ht* = 10).

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:

u_1 = *new*,
u_{21} = *new*.*push*(1),
u_{29} = *new*.*push*(1).*push*(2).*push*(3)…*push*(8).*push*(9),
u_3 = *new*.*push*(1).*push*(2).*push*(3)…*push*(8).*push*(9).*push*(10).

The following pairs of attributively non-equivalent ground terms are selected as test cases:

$\neg(u_1 \sim u_{21})$, $\neg(u_1 \sim u_{29})$, $\neg(u_1 \sim u_3)$, $\neg(u_{21} \sim u_3)$, and $\neg(u_{29} \sim u_3)$.

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

	(**array**,	**ht**)
O_{29} =	([1, 2, …, 9],	9)
O_3 =	([1, 2, …, 9],	9)

and obviously $O_{29} \approx O_3$. ■

Example 2

The following is an algebraic specification of a class *Book* in a library system.

class *Book*
import classes *Integer Real Bool String*
operations
newBook(_, _, _, _): *String String String Real* → *Book*
// Denoting name, authors, accessCode, and price, respectively.
name: *Book* → *String*
authors: *Book* → *String*
accessCode: *Book* → *String*
price: *Book* → *Real*
// There may be other attributes such as publisher, date of publication, edition, and so on.
borrow: *Book* → *Book*
return: *Book* → *Book*
toShelve: *Book* → *Book*
reserve: *Book* → *Book*
state: *Book* → *Bool Bool Bool Bool*
// Denoting onShelve, atCounter, onLoan, and reserved, respectively.
variables
B: *Book*
S: *String*
I: *Integer*
R: *Real*
axioms
a_1: *newBook*(S, _, _, _).*name* = *S*
// Similarly for authors, accessCode, and price
a_2: *newBook*(_, _, _, _).*state* = (*t*, *f*, *f*, *f*)
// t means true and f means false.
a_3: *B.borrow.name* = *B.name*
// Similarly for authors, accessCode, and price
a_4: *B.borrow.state* = (*f*, *f*, *t*, *f*) if *B.state* = (*t*, *f*, *f*, *f*)
a_5: *B.borrow.state* = (*f*, *f*, *t*, *f*) if *B.state* = (*f*, *t*, *f*, *f*)
a_6: *B.borrow.state* = (*f*, *f*, *t*, *f*) if *B.state* = (*f*, *t*, *f*, *t*)
a_7: *B.return.state* = (*f*, *t*, *f*, *f*) if *B.state* = (*f*, *f*, *t*, *f*)
a_8: *B.return.state* = (*f*, *t*, *f*, *f*) if *B.state* = (*f*, *f*, *t*, *t*)
a_9: *B.toShelve.state* = (*t*, *f*, *f*, *f*) if *B.state* = (*f*, *t*, *f*, *f*)
a_{10}: *B.reserve.state* = (*f*, *f*, *t*, *t*) if *B.state* = (*f*, *f*, *t*, *f*)
// A book cannot be reserved by more than one person.
a_{11}: *B.reserve.state* = (*f*, *t*, *f*, *t*) if *B.state* = (*f*, *t*, *f*, *f*)

Suppose an implementation of the class *Book* contains an error as follows: The implemented method *return* erroneously transforms the attribute *state* from (*f*, *f*, *t*, *f*) to (*f*, ***f, t, t***). The correct transformation should be from (*f*, *f*, *t*, *f*) to (*f*, ***t, f, f***).

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

n_0 (initial node),
n_1 = (*t*, *f*, *f*, *f*),
n_2 = (*f*, *f*, *t*, *f*),
n_3 = (*f*, *t*, *f*, *f*),
n_4 = (*f*, *f*, *t*, *t*),
n_5 = (*f*, *t*, *f*, *t*).

Suppose the maximum number M_i of paths to each node n_i from n_0 is determined by user to be 2. Then fifteen paths are generated. The ground terms corresponding to these paths are:

To node n_1:
u_{11} = *newBook*,
u_{12} = *newBook.borrow.return.toShelve*.

To node n_2:
u_{21} = *newBook.borrow*,
u_{22} = *newBook.borrow.return.borrow*.

To node n_3:
u_{31} = *newBook.borrow.return*,
u_{32} = *newBook.borrow.return.toShelve.borrow.return*.

To node n_4:
u_{41} = *newBook.borrow.reserve*,
u_{42} = *newBook.borrow.return.borrow.reserve*,

To node n_5:
u_{51} = *newBook.borrow.reserve.return*,
u_{52} = *newBook.borrow.return.reserve*,

One of the pairs of attributively non-equivalent ground terms selected as test cases is

$\neg(u_{31} \sim u_{41})$.

The pair reveals the implementation error above, since the execution results are, respectively,

O_{31} = (*f*, *f*, *t*, *t*)
O_{41} = (*f*, *f*, *t*, *t*)

and obviously $O_{31} \approx O_{41}$. ■

3 TIM

The TIM approach deals with cluster-level **T**esting by **I**ndividual **M**essage-passing rules.

3.1 Implementation of TIM

In order to implement the TIM approach, we need only write a sub-module AMP to **A**nalyze the body of the given **MP**-rule to find the messages passing across different classes in the cluster, and then construct a **C**ontrol **M**odule 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 of our TACCLE paper [2]. They will be consolidated into an integrated testing system as future work. Please refer to Section 8 of the paper.

3.2 A Case Study of TIM

In this section, we present a case study of the TIM approach.

Example 3

The following is the specification of the cluster *BankAccounts*, which contains the classes *SavingAccount* and *CheckAccount*.

contract *BankAccounts*
SavingAccount **supports**
[
balance: *Money*
name: *String*
address: *String*
r_1: *SavingAccount* ← *transferTo*(*ChkAcct*: *CheckAccount*, *M*: *Money*) ⇒
if *M* ≤ *SavingAccount.balance* **then** [*SavingAccount* ← *debit*(*M*); *ChkAcct* ← *credit*(*M*)]
else return "*overdrawn*"
r_2: *SavingAccount* ← *bal* ⇒ **return** *SavingAccount.balance*
r_3: *SavingAccount* ← *name* ⇒ **return** *SavingAccount.name*
r_4: *SavingAccount* ← *addr* ⇒ **return** *SavingAccount.address*
r_5: *SavingAccount* ← *changAddr*(*S*: *String*) ⇒
@SavingAccount.address; { *SavingAccount.address = S* }
]
CheckAccount **supports**
[
bal: *Money* // *bal means balance*
name: *String*
addr: *String* // *addr means address.*
r_6: *CheckAccount* ← *bal* ⇒ **return** *CheckAccount.balance*
r_7: *CheckAccount* ← *name* ⇒ **return** *CheckAccount.name*
r_8: *CheckAccount* ← *addr* ⇒ **return** *CheckAccount.address*
r_9: *CheckAccount* ← *changAddr*(*S*: *String*) ⇒
@CheckAccount.address; { *CheckAccount.address = S* }
]
end contract

class *SavingAccount*
imported classes *Money String*
operations
overdrawn: → *Money*
newSavAcct(_, _, _): *String String Money* → *SavingAccount*
// The input parameters denote the name, address, and balance of the account,
// respectively.
name: *SavingAccount* → *String*
addr: *SavingAccount* → *String* // *addr means address.*
bal: *SavingAccount* → *Money* // *bal means balance.*
changAddr(_): *SavingAccount String* → *SavingAccount*
// *changAddr means changing the value of the address*
credit(_): *SavingAccount Money* → *SavingAccount*

debit(_): *SavingAccount Money* → *SavingAccount*
transferTo(_, _): *SavingAccount CheckAccount Money* → *SavingAccount*
// *transfer Money from SavingAccount to CheckAccount*.

variables
S: *String*
A: *SavingAccount*
M: *Money*

axioms
a_1: *newSavAcct*(*S*, _, _).*name* = *S*
a_2: *newSavAcct*(_, *S*, _).*addr* = *S*
a_3: *newSavAcct*(_, _, *M*).*bal* = *M*
a_4: *A.credit*(*M*).*bal* = *A.bal* + *M*
a_5: *A.debit*(*M*).*bal* = *A.bal* - *M* if *A.bal* ≥ *M*
a_6: *A.debit*(*M*).*bal* = *overdrawn* if *A.bal* < *M*
a_8: *A.credit*(_).*addr* = *A.addr*
a_9: *A.debit*(_).*addr* = *A.addr*
a_{10}: *A.changAddr*(*S*).*addr* = *S*
a_{11}: *A.changAddr*(*S*).*name* = *A.name*
a_{12}: *A.credit*(_).*name* = *A.name*
a_{13}: *A.debit*(_).*name* = *A.name*
a_{14}: *A.transferTo*(_, *M*).*bal* = *A.debit*(*M*).*bal*
a_{15}: *A.transferTo*(_, _).*addr* = *A.addr*
a_{16}: *A.transferTo*(_, _).*name* = *A.name*

class *CheckAccount*
imported classes *Money String*
operations
overdrawn: → *Money*
newChkAcct(_, _, _): *String String Money* → *CheckAccount*
// The input parameters denote the name, address, and balance of the account,
// respectively.
name: *CheckAccount* → *String*
addr: *CheckAccount* → *String* // *addr means address*.
bal: *CheckAccount* → *Money* // *bal means balance*.
changAddr(_): *CheckAccount String* → *CheckAccount*
// *changAddr means changing the value of the address.*
credit(_): *CheckAccount Money* → *CheckAccount*
writeCheck(_): *CheckAccount Money* → *CheckAccount*

variables
S: *String*
C: *CheckAccount*
M: *Money*

axioms
a_1: *newChkAcct*(*S*, _, _).*name* = *S*
a_2: *newChkAcct*(_, *S*, _).*addr* = *S*
a_3: *newChkAcct*(_, _, *M*).*bal* = *M*
a_4: *C.credit*(*M*).*bal* = *C.bal* + *M*
a_5: *C.writeCheck*(*M*).*bal* = *C.bal* - *M* if *C.bal* ≥ *M*
a_6: *C.writeCheck*(*M*).*bal* = *overdrawn* if *C.bal* < *M*
a_8: *C.credit*(_).*addr* = *C.addr*

a_9: *C.writeCheck*(_).*addr = C.addr*
a_{10}: *C.changAddr*(*S*).*addr = S*
a_{11}: *C.changAddr*(*S*).*name = C.name*
a_{12}: *C.credit*(_).*name = C.name*
a_{13}: *C.writeCheck*(_).*name = C.name*

The following implementation of the cluster *BankAccounts* contains an error.

```
// BankAccounts.hpp

# include <iostream.h>
# include <stdio.h>
# include <string.h>

typedef float money;

class savingAccount
    {
    private:
        /*  3 data members:  */
        money balance;
        string name;
        string address;
    public:
        void newSavingAccount(string &, string &, money ba);
        money bal( );
        string name( );
        string addr( );
        void changAddr(string &);
        void credit(money m);
        void debit(money m);
        void transferTo(checkAccount *ca, money m);
    };

class checkAccount
    {
    private:
        /*  3 data members:  */
        money balance;
        string name;
        string address;

    public:
        void newCheckAccount(string &, string &, money ba);
        money bal( );
        string name( );
        string addr( );
        void changAddr(string &);
        void credit(money m);
        void writeCheck(money m);
        };
```

```
// BankAccounts.cpp
# include <BankAccounts.hpp>

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";
     }

void savingAccount :: newSavingAccount(string &na, string &ad, money ba)
     {
     name = na;
     address = ad;
     balance = ba;
     }

void savingAccount :: credit(money m)
     {
     balance = balance + m;
     }

void savingAccount :: debit(money m)
     {
     if (balance >= m)
          balance = balance - m;
     else cout << "overdrawn";
     }

void savingAccount :: changAddr(string &ad)
     {
     address = ad;
     }

money savingAccount :: bal( )
     {
     return balance;
     }

string savingAccount :: name( )
     {
     return name;
     }
```

```
string savingAccount :: addr( )
    {
    return address;
    }

void checkAccount :: newCheckAccount(string &na, string &ad, money ba)
    {
    name = na;
    address = ad;
    balance = ba;
    }

void checkAccount :: credit(money m)
    {
    balance = balance + m;
    }

void checkAccount :: writeCheck(money m)
    {
    if (balance >= m)
        balance = balance - m;
    else cout << "overdrawn";
    }

void checkAccount :: changAddr(string &ad)
    {
    address = ad;
    }

money checkAccount :: bal( )
    {
    return balance;
    }

string checkAccount :: name( )
    {
    return name;
    }

string checkAccount :: addr( )
    {
    return address;
    }
```

The above error in the implementation of the cluster *BankAccounts* cannot be revealed by class-level testing, regardless whether equivalent or non-equivalent ground terms are used as test cases. It can be revealed, however, by cluster-level testing through the TIM approach. Using TIM, the following tasks will be performed:

(1) The module CM invokes GFT, DOE, and GAN to perform class-level testing on the classes *SavingAccount* and *CheckAccount*. No error is revealed.

(2) CM calls AMP to analyze the body of the mp-rule r_1 to find any message passing across different classes, namely *ChkAcct* ← *credit*(*M*) from the class *SavingAccount* under the condition $M \leq$ *SavingAccount.balance*. Message passing is identified in the operation *transferTo*.

(3) CM invokes GAN to construct a ground term by traversing a path in the state-transition diagram of the class *CheckAccount*, and then executes the method sequence corresponding to the ground term on the given program to obtain an object O_{chk} of the class *CheckAccount*. The object is O_{chk} = (‘*John*’, ‘2 *University Drive*’, 3000). Its current state is saved in the variable *Pre-*O_{chk}.

(4) Similarly to step (3), CM constructs an object O_{sav} = (‘*John*’, ‘2 *University Drive*’, 8000) for the class *SavingAccount*.

(5) CM executes O_{sav}.*transferTo*(O_{chk}, 2000) on the given program of the cluster. After execution, O_{sav} becomes O_{sav} = (‘*John*’, ‘2 *University Drive*’, 6000). During the execution, the object O_{sav} also activates the method *writeCheck*(2000) on object O_{chk}. As a result, O_{chk} becomes (‘*John*’, ‘2 *University Drive*’, 1000).

(6) CM executes *Pre-*O_{chk}.*credit*(2000). The result is *Pre-*O_{chk} = (‘*John*’, ‘2 *University Drive*’, 5000). Then CM invokes DOE to examine whether *Pre-*O_{chk} is observationally equivalent to O_{chk}. Since the answer is no, an implementation error is reported. ■

4 GCS

GCS is an automatic tool to **G**enerating **C**omposite message-passing **S**equences for a given cluster.

4.1 Implementation of GCS

We have used Arity/Prolog32 in Windows 98 to implement a prototype for GCS. The reason for using Prolog is discussed in the TACCLE paper [2]. The main program for GCS is as follows:

```
generateCMS :- nl,
        % CMS means Composite Message Sequences.
    write('Please input the name of the file containing the contract in Prolog form: '),
    read(ConProFl), nl, nl,
    reconsult(ConProFl),
    ( clause(send(Ob, _), _), var(Ob)
      ;
      asserta((send(Ob, Me) :- nl, write(Ob), write(' <- '), write(Me),
      write('; '), fail)) ),
    ( clause(if(_, _), _)
      ;
      asserta((if(P, Q) :- nl, write('if '), write(P), write(': '), Q,
      nl, write('end-if '), write(P), write(', ''))) ),
    ( clause(action(_), _)
      ;
      assertz((action(Ac) :- nl, write('<< '), write(Ac), write(' >>'),
      write('; '))) ),
    ( clause(postcondition(_), _)
      ;
      assertz((postcondition(Pr) :- nl, write('{ '), write(Pr), write(' }'),
      write('; '))) ),
```

```
( clause(forAll(_, _, _), _)
  ;
  assertz((forAll(Op, Co, send(O, M)):- nl, write('('), write((Op: Co)),
  write('):'), send(O, M))) ),
write('Please input the name of the file for storing the expected composite sequences: '),
read(TesCasFl), nl, tell(TesCasFl), nl,
clause(contract(ContName), _),
write('COMPOSITE MESSAGE SEQUENCES FOR THE CONTRACT '),
write(ContName), write(':'),
clause(send(Oj, Ms), _), nonvar(Oj), nl, nl, nl,
write('FOR MESSAGE '), write(Ms),
write(' SENT TO '), write(Oj), write(': '),
nl, call(send(Oj, Ms)), fail
;
nl, nl, nl, write('END'), told.
```

4.2 Initial Experimentation of GCS

Example 4

We have run GCS with the following Contract in Prolog form for the cluster *CustomerAccount*.

```
:- op(500, fx, @).   % The sign @ means "set value".
:- op(400, yfx, #).
:- op(500, fx, return).
:- op(500, fx, /).
:- op(500, xfx, in).
:- op(500, xfx, not_in).
:- op(500, xfx, reflects).
:- op(450, fy, not).

contract(customerAccount).

send('Customer', setAddress('V')) :-
    action(@'Customer'#address),
    postcondition('Customer'#address = 'V'), send('Customer', notify).
send('Customer', getAddress) :- action(return 'Customer'#address).
send('Customer', notify) :-
    forAll( / 'Account', 'Account' in 'Customer'#accounts,
    send('Account', update) ).
send('Customer', openAccount('Account')) :-
    postcondition('Account' in 'Customer'#accounts).
send('Customer', closeAccount('Account')) :-
    postcondition('Account' not_in 'Customer'#accounts).
send('Account', setFreeze('B')) :- action(@'Account'#freeze),
    postcondition('Account'#freeze = 'B').
send('Account', update) :-
    if('Account'#freeze, action(return ': Account is frozen'))
    ;
    if(not 'Account'#freeze, send('Account', changeAddr)).
```

send(*‘Account’, changeAddr*) :- *send*(*‘Customer’, getAddress*),
postcondition(*‘Account’ reflects ‘Customer’#address*).
% where ‘Customer’ = ‘Account’#customer
send(*‘Account’, setCustomer*(*‘Cust’*)) :-
postcondition(*‘Account’#customer = ‘Cust’*).

After execution, we obtain the following composite message-passing sequences from the Contract:

FOR MESSAGE setAddress(*V*) *SENT TO Customer*:

Customer <- *setAddress*(*V*);
<< *@ Customer # address* >>;
{ *Customer # address = V* };
Customer <- *notify*;
((*/ Account*) : *Account in Customer # accounts*):
Account <- *update*;
if Account # freeze:
<< *return* : *Account is frozen* >>;
end-if Account # freeze;
if not Account # freeze:
Account <- *changeAddr*;
Customer <- *getAddress*;
<< *return Customer # address* >>;
{ *Account reflects Customer # address* };
end-if not Account # freeze;

FOR MESSAGE getAddress SENT TO Customer:

Customer <- *getAddress*;
<< *return Customer # address* >>;

FOR MESSAGE notify SENT TO Customer:

Customer <- *notify*;
((*/ Account*) : *Account in Customer # accounts*):
Account <- *update*;
if Account # freeze:
<< *return* : *Account is frozen* >>;
end-if Account # freeze;
if not Account # freeze:
Account <- *changeAddr*;
Customer <- *getAddress*;
<< *return Customer # address* >>;
{ *Account reflects Customer # address* };
end-if not Account # freeze;

FOR MESSAGE openAccount(*Account*) *SENT TO Customer*:

Customer <- *openAccount*(*Account*);
{ *Account in Customer # accounts* };

FOR MESSAGE closeAccount(*Account*) *SENT TO Customer*:

Customer <- *closeAccount*(*Account*);
{ *Account not_in Customer # accounts* };

FOR MESSAGE setFreeze(*B*) *SENT TO Account*:

Account <- *setFreeze*(*B*);
<< @ *Account # freeze* >>;
{ *Account # freeze = B* };

FOR MESSAGE update SENT TO Account:

Account <- *update*;
if Account # freeze:
<< *return* : *Account is frozen* >>;
end-if Account # freeze;
if not Account # freeze:
Account <- *changeAddr*;
Customer <- *getAddress*;
<< *return Customer # address* >>;
{ *Account reflects Customer # address* };
end-if not Account # freeze;

FOR MESSAGE changeAddr SENT TO Account:

Account <- *changeAddr*;
Customer <- *getAddress*;
<< *return Customer # address* >>;
{ *Account reflects Customer # address* };
FOR MESSAGE setCustomer(*Cust*) *SENT TO Account*:

Account <- *setCustomer*(*Cust*);
{ *Account # customer = Cust* };

END. ■

5 ESI

ESI is an automatic tool for **E**xtracting a message-passing **S**equence for each method from the **I**mplementation.

5.1 Implementation of ESI

Similarly to DOE, we have combined the implementation of the ESI algorithm with a C++ compiler since this algorithm must parse the code of the program under test.

In the implementation, a function *scan_cluster*( ) scans the cluster according to the ESI algorithm. To do so, it recursively invokes another function *scan_cluster_block*( ). The functions *scan_cluster*( ) and *scan_cluster_block*( ) makes use of the existing modules *parser.c*, *drg.c*, *subLib.c*, and *pigeonC.c* in the prototype DOE. Please refer to Section 3.4 of [1] for the details of these modules.

5.2 A Case Study of ESI

Example 5

A Contract specification of the cluster *CustomerAccount* is described in Example 9 of our TACCLE paper [2]. The following is an implementation of the cluster.

```
// CustomerAccount.hpp
# include <iostream.h>
# include <stdio.h>
# include <string.h>
# include <classlib\sets.h>

typedef float money;
enum bool {false, true};

class account;
class CheckAccount;
class SavingAccount;
typedef class account* AccountPtr;
typedef TSet<AccountPtr> TaccountSet;

class accountSet :  public TaccountSet
    {
    private:
        string customerName;

    public:
        accountSet(string &na);
    };
```

```
class customer
    {
    private:
        string name;
        string address;
        accountSet accounts;
    public:
        customer(string &, string &);
        string getAddress( );
        void setAddress(string &);
        void notify( );
        void openAccount(account *);
        void closeAccount(account *);
    };

class account
    {
    private:
        customer *custmr;
        bool freeze;
        string address;
        money balance;
    public:
        account(customer *c, bool frz, string &ad, money ba);
        void credit(money m);
        money bal( );
        void setFreeze(bool frz);
        void update( );
        void changeAddr( );
        void setCustomer(customer *c);
    };

class SavingAccount:  public account
    {
    public:
        SavingAccount(customer *c, bool frz, money ba);
        void debit(money m);
        void TransferTo(CheckAccount *ca, money m);
    };

class CheckAccount:  public account
    {
    public:
        CheckAccount(customer *c, bool frz, money ba);
        void writeCheck(money m);
    };
```

```
// CustomerAccount.cpp
# include <CustomerAccount.hpp>

customer::customer(string &na, string &ad):accounts(na)
    {
    name = na;
    address = ad;
    }

string customer::getAddress( )
    {
    return address;
    }

void customer :: setAddress(string &ad)
    {
    address = ad;
    notify( );
    }

void update(account* &ac, void* v)  // Note:  It does not belong to any class.
    {
    ac->update( );
    }

void customer::notify( )
    {
    accounts.ForEach(update, "") ;
    }

void customer::openAccount(account* ac)
    {
    accounts.Add(ac)
    }

void customer::closeAccount(account* ac)
    {
    accounts.Detach(ac) ;
    }

account::account(customer *c, bool frz, string &ad, money ba)
    {
    custmr = c;
    freeze = frz;
    address = ad;
    balance = ba;
    }
```

```
void account::setFreeze(bool frz)
    {
    freeze = frz;
    }

void account::update( )
    {
    if (freeze == true)
        cout << "Account is frozen" << endl;
    else changeAddr( );
    }

void account::changeAddr( )
    {
    address = (custmr -> getAddress( ));
    }

void account::setCustomer(customer *c)
    {
    custmr = c;
    }

void account::credit(money m)
    {
    balance += m;
    }

money account::bal( )
    {
    return balance;
    }

accountSet::accountSet(string &na)
    {
    customerName = na;
    }

SavingAccount::SavingAccount(customer *c, bool frz, money ba):
    account(c, frz, c->getAddress( ), ba)

void SavingAccount::debit(money m)
    {
    if (bal( ) >= m)
        balance -= m ;
    else cout << "overdrawn" << endl;
    }
```

```
void SavingAccount::TransferTo(CheckAccount *ca, money m)
    {
    if (bal( ) >= m)
        { debit(m);  ca -> credit(m); }
    else cout << "overdrawn" << endl;
    }

CheckAccount::CheckAccount(customer *c, bool frz, money ba):
    account(c, frz, c->getAddress( ), ba)

void CheckAccount::writeCheck(money m)
    {
    if (bal( ) >= m)
        balance -= m ;
    else cout << "overdrawn" << endl;
    }
```

The following message-passing sequences will be extracted from the above implementation:

MESSAGE-PASSING AND ACTION SEQUENCES FROM THE IMPLEMENTATION OF CustomerAccount

FOR MESSAGE getAddress( ) SENT TO Customer:

```
Customer <- getAddress;
return Customer # address;
```

FOR MESSAGE setAddress(string &ad) SENT TO Customer:

```
Customer <- setAddress(string &ad);
@ Customer # address;
Customer <- notify;
```

FOR MESSAGE notify( ) SENT TO Customer:

```
Customer <- notify( );
accounts <- ForEach(update( ));
```

FOR MESSAGE openAccount(account &ac) SENT TO Customer:

```
Customer <- openAccount(account &ac);
accounts <- Add(ac);
```

FOR MESSAGE closeAccount(*account &ac*) *SENT TO Customer*:

Customer <- *closeAccount*(*account &ac*);
accounts <- *Detach*(*ac*);

FOR MESSAGE setFreeze(*bool frz*) *SENT TO Account*:

Account <- *setFreeze*(*bool frz*);
@ Account # freeze;

FOR MESSAGE update( ) *SENT TO Account*:

Account <- *update*( );
if (*Account # freeze = true*):
cout<< *"Account is frozen"*;
end-if (*Account # freeze = true*);
if not (*Account # freeze = true*):
Account <- *changeAddr*( );
end-if not (*Account # freeze = true*);

FOR MESSAGE changeAddr( ) *SENT TO Account*:

Account <- *changeAddr*( );
(*custmr* ->) <- *getAddress*( );
@ Account # address;

FOR MESSAGE setCustomer(*customer &cu*) *SENT TO Account*:

Account <- *setCustomer*(*customer &cu*);
@ Account # custmr;

FOR MESSAGE credit(*money m*) *SENT TO Account*:

Account <- *credit*(*money m*);
@ Account # balance;

FOR MESSAGE bal( ) *SENT TO Account*:

Account <- *bal*( );
return Account # balance;

FOR MESSAGE debit(*money m*) *SENT TO savingAccount*:

savingAccount <- *debit*(*money m*);
if (*savingAccount # balance* >= *m*):
@ savingAccount # balance;
end-if (*savingAccount # balance* >= *m*);
if not (*savingAccount # balance* >= *m*):
cout << *"overdrawn"*;
end-if not (*savingAccount # balance* >= *m*);

FOR MESSAGE transferTo(*checkAccount *ca, money m*) *SENT TO savingAccount*:

savingAccount <- *transferTo*(*checkAccount *ca, money 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*);

FOR MESSAGE writeCheck(*money m*) *SENT TO checkAccount*:

checkAccount <- *writeCheck*(*money m*);
if (*checkAccount # balance* >= *m*):
@ checkAccount # balance;
end-if (*checkAccount # balance* >= *m*);
if not (*checkAccount # balance* >= *m*):
cout << *"overdrawn"*;
end-if not (*checkAccount # balance* >= *m*);

END. ■

REFERENCES

[1] 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 Transactions on Software Engineering and Methodology* **7**, 3, 250–295.

[2] Chen, H. Y., Tse, T. H., and Chen, T. Y. 2001. 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.