Generating Test Cases for Invariant Properties from Proof Scores in the OTS/CafeOBJ Method*

Masaki NAKAMURA†(a), Member and Takahiro SEINO††, Nonmember

SUMMARY In the OTS/CafeOBJ method, software specifications are described in CafeOBJ executable formal specification language. This paper proposes a test case generator from an OTS/CafeOBJ specification together with a proof score. Our test case generator gives test cases by analyzing the proof score. The quality of test cases is estimated by giving scripts to the CafeOBJ system. The script is called a proof score. In this study, we propose a test case generator from an OTS/CafeOBJ specification together with a proof score. Our test case generator gives test cases by analyzing the proof score. The test cases are used to test whether an implementation satisfies the specification and the property verified by the proof score. Since a proof score involves important information for verifying a property, the generated test cases are also expected to be suitable to test the property.

key words: formal specification, proof score, software testing, OTS, CafeOBJ

1. Introduction

Nowadays, since software plays an important role in our society, people requires software quality assurance. To obtain reliable software, formal methods are useful at the requirement and design phases. On the other hand, software testing plays important role at the implementation and debugging phases. The usefulness of software testing depends on the quality of test cases. How to design test cases which are exhaustive and efficient is important but not so easy task.

The OTS/CafeOBJ (Observational Transition Systems in CafeOBJ) method is a formal method. In this method, it is verified that a specification satisfies a given desired property by describing a proof score. There are many successful case studies of formal verification with proof scores [6], [8]–[10]. On the other hand, there is no established method to obtain test cases which test whether an implementation satisfies the property verified by a proof score.

A proof score consists of proof passages. Each proof passage can be regarded as a test case in an abstract level of the target software. If we obtain a complete proof score for some property, the set of proof passages is guaranteed to be exhaustive and to be enough detailed for the verification to succeed. By generating test cases from a proof score, it is expected that (1) exhaustive test cases can be obtained systematically and semi-automatically, and (2) the quality of test cases can be improved. The test case generator we propose in this paper takes an OTS/CafeOBJ specification and its proof score, and generates (a) Java skeleton code and (b) test cases. The generated test cases are used for testing an implementation which is obtained by instantiating the generated Java skeleton code. To handle generated test cases effectively and efficiently, we use JUnit testing framework*** for the Java programming language**** (Fig. 1).

In the rest of this paper, we first introduce the OTS/CafeOBJ method with an example of a specification of an automated teller machine of a bank and its proof score. In Sect. 3, we propose a test case generator from an OTS/CafeOBJ specification and its proof score. We show how to use generated test cases in Sect. 4. In Sect. 5, we give theorems on the completeness and the exhaustiveness of our test case generator. In Sect. 6, we give some improvement of our proposed generator, and conclude with some of the future work in Sect. 7.

2. Preliminaries

We assume the reader is familiar with the Java programming language. In this section, we introduce OTS/CafeOBJ specifications and proof scores [3], [8], [9], which are inputs of our test case generator.

2.1 Data Specification

A CafeOBJ specification consists of modules. The following is a CafeOBJ module whose name is USER.

```cafe
mod USER { [ User ]
  op _=_ : User User -> Bool { comm }
  var U : User
  eq (U = U) = true .
}
```

Module USER has a declaration of a sort User between the square brackets [ ], and a binary operator _=_ on User after the keyword op. In an operator declaration, the sequence of sorts at the left-hand side of _-> (e.g. User User for _=_) is called the arity, and the sort at the right-hand side (e.g. Bool) is called the co-arity. Underlines (_) are positions of arguments, i.e., _u_ = _u_′ is a term of Bool for terms _u_ and _u_′ of
Fig. 1 Test case generator.

User. comm at the end of an operator declaration means that the operator is commutative. var is a keyword of variable declaration. U is a variable of User, which is used in the following equation declaration. Equations are declared with eq. The equation in USER denotes that Term \( u = u \) is equivalent to true for any \( u \) of User. A set of sorts and operators is called a signature, and equations is called axioms.

A CafeOBJ specification denotes an algebra which has carrier sets and functions which correspond to the sorts and the operators respectively and which satisfies the equations. USER denotes an algebra which has a carrier set (User) and a commutative and reflexive function (predicate) \( (=, \_\_\_\_\_) \). CafeOBJ contains built-in modules for standard data types, like integers, strings, etc. For example, INT is a built-in module together with constants \( ... , -2, -1, 0, 1, 2, ... \), and operators \( +, - \) etc.

2.2 System Specification

An OTS is a state machine whose states are identified with its observations. A CafeOBJ specification which denotes an OTS is called an OTS/CafeOBJ specification. An OTS/CafeOBJ specification consists of data specifications, like USER and INT, and a system specification. We show ATM automated teller machine of a bank, an example of system specifications. First, the signature part of ATM is given as follows:

```
mod* ATM {
  pr(INT + USER)
  *[ Atm ]*
  op init : -> Atm
  bop balance : User Atm -> Int
  bop deposit : User Int Atm -> Atm
  bop withdraw : User Int Atm -> Atm
  op c-deposit : User Int Atm -> Bool
  op c-withdraw : User Int Atm -> Bool
}
```

Module ATM imports INT and USER in the protect mode (with the keyword pr(...)). Roughly speaking, a protectively imported module is used by the importing module with no change. A special sort, called a hidden sort, is declared enclosing \( *[ \_\_\_\_] \). Non hidden sorts are called visible. A hidden sort is used to denote the state space of a target system. An initial state init is declared as a constant, an operator whose arity is empty. Special operators, called behavioral operators, are declared after the keyword bop. A behavioral operator should have at most one hidden sort in its arity. When its co-arity is hidden, the behavioral operator is called a transition. When its co-arity is visible, it is called an observation. In ATM, balance is an observation, which observes the balance of a user. deposit is a transition, which deposits money into a user’s account. The other transition withdraw withdraws money from a user’s account. Operators c-deposit and c-withdraw are used to define pre-conditions of deposit and withdraw respectively. Next the axioms part of ATM is given as follows:

```
vars U U' : User
var I : Int
var A : Atm

eq c-deposit(U,I,A) = I >= 0.

eq c-withdraw(U,I,A) = balance(U,A) >= I and I >= 0.

eq balance(U,init) = 0.

ceq balance(U, deposit(U',I, A)) =
  (if U = U' then I + balance(U,A)
  else balance(U,A) fi)
  if c-deposit(U',I,A) .

ceq deposit(U',I,A) = A
  if not (c-deposit(U',I,A)) .

ceq balance(U,withdraw(U',I, A)) =
  (if U = U' then balance(U,A) - I
  else balance(U,A) fi)
  if c-withdraw(U',I,A) .

ceq withdraw(U',I,A) = A
  if not(c-withdraw(U',I,A)) .
```

The first two equations define pre-conditions of deposit and withdraw. An amount of deposit should not be a negative integer. An amount of withdrawal should be at least zero and at most the balance of the user who is withdrawing. The third equation defines the initial balances of all users as zero. The keyword ceq is used for declaring a conditional equation. The conditional equation ceq e if c means that the equation e holds whenever the condition c holds. In this paper, both conditional and unconditional equations are simply called equations. The fourth equation defines the balance of each user after applying deposit to a state A which satisfies the pre-condition of deposit. Only the balance of the user who makes the deposit is increased. The fifth equation in ATM means an application of deposit has no effect when the pre-condition does not hold. The sixth and seventh equations define the behavior of withdraw in a similar way. Note that we adopt the syntactic definition of OTS/CafeOBJ specifications proposed in [7].

2.3 Verification with Proof Score

One of the important purposes of formal methods is to verify a specification satisfies requirements. The executable specification language CafeOBJ supports the reduction command red which supports automatic equational reasoning. A proof passage is a set of declarations of constants, equations and reductions, which correspond to arbitrary elements, antecedents and a consequent respectively. The following is an example of proof passages:

open ATM .
ops u1 u2 : -> User .
ops a1 a2 : -> Atm .
eq (u1 = u2) = false .
eq a1 = deposit(u2,200,init) .
eq a2 = deposit(u1,100,a1) .
red balance(u2,a2) .
close

The first line is an instruction of opening Module ATM. While opening a module, we may declare operators, equations and reductions on the module. ops is a keyword for declaring multiple operators whose arities and co-arities are same. The first equation means that the users u1 and u2 are different. The second equation means that a1 is the result state of applying deposit with 200 by User u2 to the initial state. The third equation means that a2 is the state after deposit with 100 by u1 to a1. The reduction command is an instruction of reducing balance(u2,a2), the balance of u2 at a2. For the above proof passage, the CafeOBJ system returns 200. Thus, it guarantees the following sentence:

\[ \forall u_1, u_2 \in \text{User}, \forall a_1, a_2 \in \text{Atm}. \]
\[ ((u_1 = u_2) = \text{false}) \]
\[ \land (a_1 = \text{deposit}(u_2,200,\text{init})) \]
\[ \land (a_2 = \text{deposit}(u_1,100,a_1)) \]
\[ \Rightarrow \]
\[ \text{balance}(u_2,a_2) = 200 \]

For a given property, the set of all proof passages which guarantee the specification to satisfy the given property is called a proof score. In this paper we focus on a proof score for invariant properties. A predicate inv on the set of states is invariant if and only if inv holds for any state which can be obtained by applying transitions to the initial state, called a reachable state. The following is a preliminary part of a proof score for verifying that the predicate \( \forall u. \text{balance}(u,s) \geq 0 \) is invariant for ATM.

mod INV {
  pr(ATM)
  op inv : User Atm -> Bool
  eq inv(U:User,A:Atm) = balance(U,A) \geq 0 .
  ops s s’ : -> Atm
  op istep : User -> Bool
  eq istep (U:User) =
  \text{inv}(U, s) \text{ implies } \text{inv}(U, s’) .
}

Module INV specifies a predicate inv, which will be proved invariant, states s, s’, which are used as a current state and a successor state, and a predicate istep, which is used to prove that if a current state satisfies inv, so does a successor state. In the following, we prove that inv satisfies the initial state (Base Step), and istep holds when s’ = \( \tau(s) \) for any transition \( \tau \), i.e. \( \tau \) preserves inv (Induction Step). The following is a proof passage for the property that the initial state init satisfies inv (for any user u1):

open INV
  op u1 : -> User .
  red inv(u1, init) .
close

The reduction command returns true, whose trace is

\( \text{inv}(u1,\text{init}) \implies \text{balance}(u1,\text{init}) \geq 0 \implies \emptyset \implies \emptyset \implies \emptyset \implies \emptyset \implies \text{true} \)

The following is one of the proof passages for the property that deposit preserves inv:

open INV
  op u1 u2 : -> User .
  op i1 : -> Int .
  eq u2 = u1 .
  eq i1 \geq 0 = \text{false} .
  eq s’ = \text{deposit}(u2, i1, s) .
  red istep(u1) .
close

Arbitrary users u1 and u2, and an arbitrary integer i1 are declared such that u1 and u2 are a same user and i1 is negative. The last equation means that s’ is obtained by applying deposit(u2, i1, ) to s. The reduction command returns true, and it is guaranteed that deposit preserves inv under the above conditions: u1 = u2 and i1 \( \geq 0 \) = false. To complete a proof score, proof passages should be prepared for all cases, for example, u1 and u2 are different, i1 is positive, and for all transitions. When CafeOBJ returns true for the all cases, it is guaranteed that any user’s balance should not be negative for all reachable state. We
have verified the property with a proof score consisting of eleven proof passages (with some lemma). See [8], [9] for more details of proof scores in OTS/CafeOBJ method. In this paper, for each proof passage, we assume all equations except for the last one \( eq \ s' = deposit(u_2, i_1, s) \) do not have any transition operators which is a natural assumption, and most existing proof scores of the OTS/CafeOBJ method satisfy this as far as we know.

2.4 JUnit Testing Framework

JUnit automated unit testing framework has been developed by Kent Beck and Erich Gamma, and is one of the most popular supporting tools for Test Driven Development (TDD) for Java programs [1]. Once making a set of test cases, we can run all the test cases automatically and repeatedly. Classes for testing are inherited from \texttt{JUnit.TestCase} provided by JUnit. Inside the test class inherited from \texttt{TestCase}, a test code for each item (method or case) is written in a method whose name begins with \texttt{test}, e.g. \texttt{testInv_deposit_1}.

3. Generating Java Codes

In this section, we propose a rule generating a skeleton code and test cases for an implementation of an OTS/CafeOBJ specification from the specification and a proof score. Table 1 is a correspondence between an input OTS/CafeOBJ specification with a proof score and the output Java codes. A sort declared in data specifications corresponds to a class, called a data class. A non-behavioral operator corresponds to a class method (a static method) of the special class \texttt{Ops}. A class generated from a system specification (a hidden sort) is called a system class. Observations, transitions and an initial state correspond to instance variables (attributes), instance methods and a constructor respectively. A class for testing is generated from a proof score. Each proof passage corresponds to an instance method for testing. Hereafter, we give a definition of our test case generator. We use the notation “...” as an omission mark. Some codes which can be easily complemented are there. The notation \( \square \) stands for the empty, i.e., no codes are there. \( \square \) should be filled by an implementer after generating skeleton codes. We assume any input CafeOBJ code is finite, that is, the sizes of data and system specifications and proof scores are finite. Thus, the numbers of sorts, operators, equations, proof passages and so on are also finite.

3.1 Data Class

Each sort in data specifications are translated into an empty class with the same name\(^†\).

**Definition 3.1:** Let \( S \) be a visible sort. The generated data class \( S \) is defined as class \( S \{ \square \} \).

**Example 3.2:** We show the data classes generated from visible sorts \texttt{Int} and \texttt{User}:

```java
class Int {}
class User {}
```

Especially for the CafeOBJ built-in module \texttt{BOOL} we let it correspond to the Java primitive data type \texttt{boolean}. A data class (or its object) is expected to be immutable, i.e., the state cannot be modified after the object is created, like \texttt{java.lang.String} class.

Class \texttt{Ops} is a class containing class methods generated from non-behavioral operators, which is regarded as a bridge between CafeOBJ operators and Java data classes.

**Definition 3.3:** Let \( f_i : S_{i1} S_{i2} \cdots \rightarrow S_i \) be the set of all non-behavioral operators declared in the OTS/CafeOBJ specification. The generated operator class \texttt{Ops} is defined as follows:

```java
class Ops{
    static \( S_1 f_1(S_{11}, S_{12}, \ldots) \{ \square \} \)
    static \( S_2 f_2(S_{21}, S_{22}, \ldots) \{ \square \} \)
    \ldots
}
```

We assume that the input OTS/CafeOBJ specification includes operators (predicates) \( \_\_\_ : S \rightarrow \text{Bool} \) for each visible sort \( S \). Thus, \texttt{Ops} includes class methods \texttt{boolean equal(S s_1, S s_2)}\(^††\).

**Example 3.4:** We show the operator class generated from \texttt{ATM} \(^†††\):

```java
class Ops {
    static boolean equal(User u1, User u2){ } 
    static boolean equal(Int i1, Int i2){ } 
    static boolean ge(Int i1, Int i2){ } 
}
```

\(^†\)CafeOBJ can treat an order relation on sorts, however, for simplicity, we assume that there are no ordered sort.

\(^††\)Since the syntax of Java does not allow \( \_\_\_ \) as a method name, we let \texttt{equal} be the name of the generated method. For other such cases, we also rename them similarly.

\(^†††\)A built-in module may have a lot of operators. Some of them are not used in the input data and system specifications, e.g. not only \( \_\_\_\_\_\_\_\_\_, \_\_\_\_\_\_\_\_\_\_, \_\_\_\_\_\_\_ \) but also \( \_\_\_\_\_\_\_\_\_\_\_\_ \) are included in the built-in \texttt{INT}. For built-in modules, we do not generate methods for such unused operators.

---

**Table 1** OTS/CafeOBJ and Java.

<table>
<thead>
<tr>
<th>OTS/CafeOBJ specification</th>
<th>Java code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data specification</td>
<td></td>
</tr>
<tr>
<td>Sort</td>
<td>Data class</td>
</tr>
<tr>
<td>Non-behavioral operator</td>
<td>Class method (Ops)</td>
</tr>
<tr>
<td>System specification</td>
<td></td>
</tr>
<tr>
<td>Hidden sort</td>
<td>System class</td>
</tr>
<tr>
<td>Observation</td>
<td>Instance variable</td>
</tr>
<tr>
<td>Transition</td>
<td>Instance method</td>
</tr>
<tr>
<td>Initial state</td>
<td>Constructor</td>
</tr>
<tr>
<td>Non-behavioral operator</td>
<td>Class method (Ops)</td>
</tr>
<tr>
<td>Proof score</td>
<td></td>
</tr>
<tr>
<td>Proof passage</td>
<td>Instance method</td>
</tr>
</tbody>
</table>
static Int zero(){
}
static Int plus(Int i1, Int i2){
}
static Int minus(Int i1, Int i2){
}
static boolean c_deposit(User u, Int i, 
    Atm a){
    public Int balance(User u){
        return s.balance(u)
    }
    public void deposit(User u, Int i){
    }
    public void withdraw(User u, Int i){
    }
}

3.3 Test Class

We give a translating rule from CafeOBJ terms to Java codes.

Definition 3.8: Let \( t \) be a term constructed from operators in an input OTS/CafeOBJ specification except transition operators. The translated Java code \( t' \) is recursively defined as follows:

\[
\begin{align*}
\begin{cases}
\text{Ops.} f(s(t')) & \text{if } t = f(t), \\
\text{s} \circ o(s(t')) & \text{if } t = o(t', s), \\
0 \circ w. & \text{otherwise}
\end{cases}
\end{align*}
\]

Here, \( s \) in \( t' \) is the instance variable of the system class \( H \) used for testing by a test class \( HTest \).

For example, Terms \( t_1 = i1 \gg \emptyset \) and \( t_2 = \text{balance}(u, s) \gg \emptyset \) are translated into Java codes \( t'_1 = \text{ge}(i1, \text{zero}()) \) and \( t'_2 = \text{ge}(\text{s.balance}(u), \text{zero}()) \). Especially for constructors of \( \text{Bool} \), they are translated into the corresponding constructors on the Java boolean type, e.g., Term \( t_1 \) and \( t_2 \) are translated into Java code \( t'_1 \) and \( t'_2 \).

Definition 3.9: A test class \( HTest \) is defined as follows:

```java
import junit.framework.TestCase;
public class HTest extends TestCase{
    TestAxiom
    TestInv
}
```

TestAxiom consists of test methods testAxiom_init_o1 and testAxiom_init_o2 for all observations and transitions. For the equations \( \text{o}_1 \rightarrow (\text{X}_1, \text{init}) = \text{rinit}_1 \), of the initial state, test methods testAxiom_init_o1, which test new \( H() \) against the axiom, are defined as follows:

```java
public void testAxiom_init_o1(){
    // begin setup
    V_{init}, X_{init} = \emptyset; \\
    V_{init}, X_{init} = \emptyset; \\
    Atm s = new H();
    // end setup 
    V_{init}, X_{init} = \text{rinit}_1; \\
    assertEquals("Axiom_init_o1\rightarrow axiom", 
        Ops.equal(s.o1(\text{X}_1), \text{right}));
}
```

Notes: \( \text{AX} \) is an abbreviation of a sequence \( A_1, \ldots, A_n \).
If we write `assertTrue(str, cond)` and cond is false, i.e., the assertion fails, then we receive an assertion failed error message together with `str`.

For `eq oj(τ(S, Yi), Xj) = roj τi` if `c−τi(S, Yj) of the transitions τi`, test methods `testAxiom_τi,oj`, which test the transition method τ against the axiom, are defined as follows:

```java
public void testAxiom_τi,oj(){
    // begin setup
    V_oj Xj1 = □;
    V_oj Xj2 = □; ... 
    V_oj Yj1 = □;
    V_oj Yj2 = □; ...
    Atm s = □
    // end setup
    boolean pre = crτ'1;
    assertTrue("Axiom_τi,oj:setup", pre);
    V_oj right = roj τi;
    s.τi(Yj);
    assertTrue("Axiom_τi,oj:axiom",
                Ops.equal(s.oj(Xj), right));
}
```

TestInv consists of test methods `testInv_init.i` and `testInv_τi,j` for all proof passages. Let `eq inv(X, S : H) = rinv(X, S)` be an equation defining the invariant property in INV. Let `eq limitij = initij (j = 1,2,...)` be the set of all equations except the last `eq s' = ...` included in the i-th proof passage for the initial state. Test methods `testInv_init.i (i = 1,2,...)`, which test new `H` against `inv`, are defined as follows:

```java
public void testInv_init.i(){
    // begin setup
    □
    V1 x1 = □;
    V2 x2 = □;
    ...
    Atm s = new H();
    // end setup
    boolean pre = equal(limit1, init1) &
                  equal(limit2, init2) & ...
    assertTrue("Init:setup", pre);
    boolean inv = rinv(X, s');
    assertTrue("Init:inv", inv);
}
```

Let `eq τijk = τij (k = 1,2,...)` be the set of all equations except the last `eq s' = ...` included in the j-th proof passage for transitions τ. Test methods `testInv_τi,j (j = 1,2,...)`, which test τ against `inv`, are defined as follows:

```java
public void testInv_τi,j(){
    // begin setup
    V1 x1 = □;
    V2 x2 = □;
    ...
    Vτ1 y1 = □;
    Vτ2 y2 = □;
    ...
    Atm s = □
    // end setup
    boolean pre = equal(τij1, τij2) &
                  equal(τij2, τij3) & ...
    assertTrue("Inv_τi,j:setup", pre);
    s.τi(Y)
    boolean inv = rinv(X, s');
    assertTrue("Inv_τi,j:inv", inv);
}
```

**Example 3.10:** We show a part of the test class AtmTest generated from ATM.

```java
public class AtmTest extends TestCase {
    public void testAxiom_init_balance(){
        // begin setup
        User U = Atm s =
        // end setup
        boolean pre = equal(lτ1, rτ1) &
                      equal(lτ2, rτ2) & ...
        assertTrue("Axiom_init_balance:setup", pre);
        s.τi(Y)
        boolean inv = rinv(X, s');
        assertTrue("Axiom_init_balance:axiom",
                    Ops.equal(s.balance(U),
                               right));
    }

    public void testAxiom_deposit_balance(){
        // begin setup
        Int I =
        User U = User U2 =
        Atm s =
        // end setup
        boolean pre = equal(lτ1, rτ1) &
                      equal(lτ2, rτ2) & ...
        assertTrue("Axiom_deposit_balance:setup", pre);
        Int right = Ops.if_then_else_fi(0,
                                         Ops.equal(U, U2),
                                         Ops.plus(I, s.balance(U)),
                                         s.balance(U));
        int s.deposit(U2, I);
        assertTrue("Axiom_deposit_balance:axiom",
                    Ops.equal(s.balance(U),
                               right));
    }

    public void testInv_init_1(){
        // begin setup
        User u1 =
        Atm s = new ATM();
        // end setup
        boolean pre = true;
    }
```
4. Implementation with Generated Test Cases

We show how to use generated test cases (AtmTest) to complete a Java skeleton code (Class Atm, etc).

4.1 Preparations

The first error we face when executing the generated test class is a syntax error between // begin setup and // end setup, called a preparation part, in a test method. Each test method tests whether Class Atm satisfies an assertion “P implies Q”. Filling a preparation is choosing an instance of P. For example, to initialize User U and Int I, data classes User and Int should be completed first. In this example, for both data classes we give classes whose attributes are private variables of the Java primitive data type int and methods getVal to refer the values. We do not give public methods which change their values, like setVal, in order for those classes to be immutable. The soundness of test methods depend on the property that all local variables initiated in the preparation part are not changed through the test execution. More precisely, the value of I at boolean pre = Ops.ge(I, Ops.zero()) in testAxiom_deposit_balance should be preserved at the later occurrence at s.deposit(U2, I).

4.2 Operation Class

The next error may be caused by methods in Class Ops, e.g. Ops.zero does not return Int value. In Class Ops, we need to define methods corresponding to the operators in the input specification. To avoid such error, for example, Ops.zero are given as follows:

```java
static Int zero(){
    Int j = new Int(0);
    return j;
}
```

4.3 Observation Class

Next, Interface IState should be implemented. Object State implementing IState for Atm is given as follows:

```java
public class State implements IState {
    private Hashtable h;
    public State(){
        h = new Hashtable();
    }
    public void setBalance(User u, Int i){
        h.put(u.getUid(),i);
    }
    public Int balance(User u) {
        if(h.get(u.getUid()) != null)
            return (Int) h.get(u.getUid());
        return Ops.zero();
    }
}
```

Our example implements IState by Java class Hashtable of a hash table. Class State keeps the balance for each user. We also define Constructor of Atm as public Atm() { s = new State(); }.

4.4 Assertion Failed Error from JUnit

When we face a runtime error AssertionError from junit.framework, it should be a preparation error, an axiom error, or an invariant error. It can be recognized by a message together with the error. The preparation error indicates that a test method has an error. The axiom error and the invariant error indicate that the implementation has an error.

4.4.1 Preparation Error

When the message ends with setup, the error is a preparation error which indicates that the preparation part does not satisfies the antecedent P of “P implies Q”. For example, if we initiate I = new Int(-1000) in testAxiom_deposit_balance, a preparation error is returned since it does not satisfies the precondition pre = Ops.ge(I, Ops.zero()).

4.4.2 Axiom Error

A message with axiom indicates that the implementation
does not satisfy the specification. For example, if the definition of deposit still empty, it does not satisfy the following equation:

\[
\text{ceq balance}(U, \text{deposit}(U', I, A)) = \begin{cases} 
\text{if } U = U' \text{ then } I + \text{balance}(U, A) \\
\text{else balance}(U, A) \end{cases} \\
\text{if } \text{c-deposit}(U', I, A).
\]

which means that deposit(U, I) increases the balance of U. To avoid the error, we give the definition of Method deposit as follows:

```java
public void deposit(User u, Int i){
    Int j = s.balance(u);
    Int k = (j.getVal() + i.getVal()) ;
    ((State) s).setBalance(u, new Int(k));
}
```

### 4.4.3 Invariant Error

A message with `inv` indicates that the implementation does not satisfy the invariant property. The invariant property of our example is that balances are not negative. The above definition of deposit does not satisfy the invariant property since we can deposit a negative value. To avoid the invariant error, we improve the definition of deposit as follows:

```java
public void deposit(User u, Int i){
    if (i.getVal() >= 0){
        Int j = s.balance(u);
        Int k = (j.getVal() + i.getVal()) ;
        ((State) s).setBalance(u, new Int(k));
    }
}
```

When no error is returned in the end, we obtain an implementation which passes all test cases.

### 5. Properties

The following properties hold under the assumption that each data class is immutable.

**Theorem 5.1:** If some test method in HTest returns an error `AssertionFailedError` with the message `Axiom_x:axiom`, the implementation does not satisfy the specification. More precisely, if `x` is `init`, it does not satisfy the equation \( o(..., \text{init}) = \cdots \). If `x` is a transition `τ`, it does not satisfy the equation \( o(τ(...)) = \cdots \). If the message is `Inv_x:inv`, the implementation does not satisfy the invariant property. More precisely, if `x` is `init`, the constructor definition `public H(){...}`, does not satisfy `inv`. If `x` is a transition `τ`, Method `τ` does not preserve `inv`.

**Proof.** A counter-example can be made from the preparation part of the indicated transition method. If the message is `Inv_deposit_1:inv` for example, the set of initiated variables in the preparation part of Method deposit is a counter-example, i.e., we have an state which does not satisfy `inv` by initiat-
are reduced into \texttt{false} for those four proof passage. The final test class does not include the test methods for the four proof passages, and the other test methods can be instantiated without any preparation error. Note that the theorems in the previous section hold even if our test case generator has been improved as above.

7. Related Work

There are two kinds of formal verification techniques: theorem proving and model checking. The CafeOBJ processor can be regarded as an interactive theorem prover. For theorem provers, for example Isabelle/HOL, PVS and so on, test generation is used for \textit{specification testing}, that is, testing is used to check the desired property with randomly produced parameters before or during the formal verification of that property [2],[11]. Those approaches are helpful in practice to find bugs in a given specification or a property to be verified. As far as we know, our test case generator is the first tool to generate test cases for \textit{program testing} from specifications together with interactive proofs.

The mainstream of studies on test generation for program testing is automated test generation with model checkers [13]. Model checkers are tools to verify that a system satisfies a given logical formula written in the propositional logic, the temporal logic or so on. One of the most important features of the model checkers is that it does not only prove a given property but also disprove the property with a counter example. The counter example is a sequence of reachable states beginning with an initial state and ending with a state where the input property is violated. In the literature [5], a method for generating test sequences with model checkers has been proposed. Let $P \Rightarrow Q$ be a property which already has been verified. Our goal is to obtain a meaningful test sequence, i.e. a sequence of transitions $\vec{t}$, from $P \Rightarrow Q$, that is, when applying $\vec{t}$ to the initial state, we obtain the state $s$ which satisfies $P$ and thus which should be tested to satisfy $Q$ since $P \Rightarrow Q$ has been verified. To obtain such a test sequence, we first translate $P$ to the negation of $P$, which is called a trap property [5], and then check whether $\neg P$ holds for any reachable state with a model checker. Then, the model checker returns a counter-example (if any), that is, a sequence of reachable states beginning with an initial state and ending with a state which satisfies $P$. In [5], the model checkers SMV and SPIN are used and compared. For reference, the literature [13] may be helpful as more detailed survey for automatic test generation with model checkers.

Relating with our study, the model checking approach may coexist with our test case generator. Our test case generator generates test cases whose preparation part, $P$, should be instantiated to run the tests. To instantiate the preparation part, model checkers approaches may be helpful. On the other hand, for the model checking approach, our test case generator gives suitable criteriaons $P$ to test an invariant property. There is another difference between our approach and the model checker approach on inputs which each technique can be applied to. Basically model checkers can be applied to only concrete finite state machines, while the OTS/CafeOBJ method can deal with more abstract specifications. For example, Module USER should be refined to a concrete finite data to do model checking.

8. Conclusion

Generated test cases are regarded as black-box testing [4], which tests whether the implementation satisfies the specification and the invariant property. Each test method for the invariant property is generated from each proof passage. The case analysis in the proof score has enough information to verify the invariant property, and we believe that it is also useful for the testing phase. Our test case generator can be improved by combining other software test methods and techniques. For each generated test method (skeleton code), we need to instantiate the preparation part. At that phase, we can apply existing useful methods and techniques used in the area of software testing: equivalence partitioning, the boundary value analysis, etc [4]. A study of those techniques for user-defined abstract data type, which is one of the most important features of algebraic specification languages, is one of the future work. We showed an example which has only one proof score for a given specification. A specification may have many proof scores if there are several desired properties, for example, liveness properties and so on. For such cases, our test case generator may generate too many test cases. A study of how to manage (merge or reduce) test cases of several proof scores is another one of the future work. We have implemented the proposed test case generator in XSLT (XSL transformations). The tool is a part of the family of supporting tools for the OTS/CafeOBJ formal method including, for example, Gateau [14] toolkit for generating and displaying proof scores.

Acknowledgement

This research was partially supported by the Ministry of Education, Culture, Sports, Science and Technology (MEXT), Grant-in-Aid for Young Scientists (B), 17700028 and 1870024.

References


Masaki Nakamura is an assistant professor at School of Electrical and Computer Engineering, College of Science and Engineering, Kanazawa University. He received his Ph.D. in information science from JAIST (Japan Advanced Institute of Science and Technology) in 2002. He was an assistant professor at Graduate School of Information Science, JAIST from 2002 to 2008. His research interest includes software engineering, formal methods, algebraic specification and term rewriting.

Takahiro Seino is a postdoc researcher at Center for Service Research, National Advanced Institute of Science and Technology (AIST). He received his Ph.D. in information science from Graduate School of Information Science, Japan Advanced Institute of Science and Technology (JAIST) in 2003. His research interests include combining formal methods and ontology for large-scale information systems. He is occupied on some actual projects which built up large-scale information systems applying his research results.