Journal of Theoretical and Applied Information Technology
31st July 2022. Vol.100. No 14
© 2022 Little Lion Scientific

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

ADJACENCY MATRIX, GRAPH THEORY AND EQUIVALENCE PARTITIONING FOR MODELLING CONFORMITY TESTING OF OBJECT ORIENTED PROGRAMS

KHALID BENLHACHMI [1], KHADIJA LOUZAOUI [2]

[1,2]Laboratory For Computer Science Research, Faculty of Science, Ibn Tofail University, Kenitra, Morocco

E-mail: [1]khalid.benlhachmi@uit.ac.ma , [2]khadija.louzaoui1@uit.ac.ma

ABSTRACT

We present in this work an approach for testing conformity behaviours of object oriented (OO) classes. Our approach can be used to test overridden and overriding methods during the inheritance process. The key idea of our work is to use a mathematical representation for developing some algorithms of test data generation to deduce all states of conformity in the general case where behaviours of methods are not necessarily similar.
Our mathematical model describes conformity contract of overridden and overriding methods during the inheritance mechanism by a graph of conformity states, adjacency matrix and equivalence partitioning. The technique of partitioning can define test cases that uncover classes of errors, thereby reducing the total number of test cases that must be developed. The second model is based on adjacency matrix and graphs of states to represent software behaviour and to simplify the test data generation.
We show in this paper that the test data generation can be represented by a graph of states and adjacency matrix. Thus, it is sufficient to consider the sink vertex of graphs to check the conformity behaviour of the program under test.

Keywords: *Software Verification, Formal Specification, Conformity Testing, Robustness Testing, Valid Data, Invalid Data, Test Data Generation, Equivalence Partitioning, Inheritance, Constraint Resolution.*

1. INTRODUCTION

Formal modelling is an important method of discovering O example of conformity and robustness testing.

Formal specifications represent an effective method to improve the efficiency and quality of software. However, the choice of test data has an important impact on the quality of software testing. Furthermore it is difficult to find out all anomalies in the system. This major problem of software testing throws an important question, as to what would be the approach we should adopt for generating test data. In this paper we develop a constraint model that includes various abstraction levels and corresponding methods for synthesis and verification of conformity properties: the first method is a behavioural equivalence partitioning of input domains of derived classes of inheritance. The second method is based on graph theory for describing all states of conformity of overridden and overriding methods. Our approaches are based on formal specifications and design by contracts (DBC) [1, 2, 3].

For Object Oriented (OO) programs, design by contract represent a powerful technique for robust and reliable software. DBC is based on three Boolean constraints: precondition, postcondition and invariant *(P, Q, Inv)* (Fig.1). The specification *(P, Q, Inv)* must be satisfied in input and output of programs under test, and can be used by different languages of constraints: OCL[4] and JML[5]… .

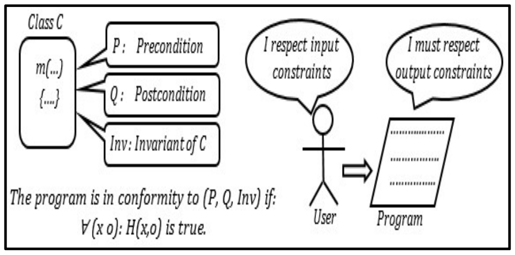

Class C
m(...)
{....}
P : Precondition
Q : Postcondition
Inv: Invariant of C
I respect input constraints
I must respect output constraints
The program is in conformity to (P, Q, Inv) if:
∀(x o): H(x,o) is true.
User
Program

Figure 1. Constraints and Conformity contract of an OO program

In an OO paradigm, the conformity contract is a property H defined for all elements of the input domain $E\times I_c$ of the program under test:

$$H : E\times I_c \rightarrow \{true, false\}$$
$$(x,o) \rightarrow H(x,o) \text{ is} : P(x,o)\wedge Inv(o_{(bef)}) \Rightarrow Q(x,o)\wedge Inv(o_{(aft)})$$

This conformity constraint of the program under test is satisfied if: "For all invocation of the program, output specifications (Q and Inv) are satisfied **if** input specifications (P and Inv) are satisfied" (Fig.1).

Our previous approaches of conformity testing [6] and robustness testing [7,8] in inheritance are based on similarity of behaviours [9] between overridden and overriding methods. In our basic approaches we have indeed tested the conformity of overriding methods in derived classes from test results of overridden methods in the super class. This reusability of test sequences of the super class is only possible if overriding and overridden methods have same basic behaviour [9]. In this paper we present an approach for testing the conformity of overriding methods in derived classes in the general case where overriding and overridden methods are not necessarily similar. In this work we use the technique of equivalence partitioning to reduce the number of test data. The second method is the adjacency matrix and graph of states to simplify the representation of test data results (Fig.2).

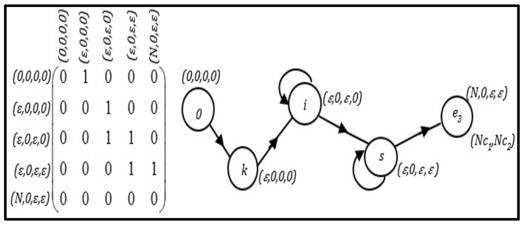


Figure 2. Graph of conformity states and its adjacency matrix

We organize our paper as follows: section 2 and 3 present similar approaches of software testing and our previous works of conformity constraints in derived classes. In section 4 we propose our approach of conformity testing of inheritance by using the graph of conformity states. Finally, our approach is evaluated by an OO example of conformity testing.

2. RELATED WORKS

Several works have been done to test the compliance of a component or system with formal specifications. In [4], authors propose an approach of test cases generation, the method is based on the constraints resolution and error anticipation in programs specifications. In [5], they present a method for writing all assertions of OO programs in the abstract state defined by Java 8 streams and Java Modelling Language (JML). In [6] we have used an optimal model of constraints only for testing conformity in inheritance. In [7], we propose an approach of robustness testing for derived classes of inheritance in an OO paradigm. The purpose of this approach is the opportunity to reuse test sequences of overridden methods in robustness testing of overriding methods. This reusability of test data of super classes is only possible if methods have same behavior. In [8] we have used an optimal model of constraints for testing both conformity and robustness of derived classes of inheritance.

In [9], we have proposed a model of similar behaviors based on an equivalence partitioning for testing the similarity between overridden and overriding methods of OO classes. In [10], the paper uses the EventML software and Nuprl proof assistant to implement a consensus protocol which must be fault tolerant. The approach shows how to prove sofety properties for the protocol in question. In [11], the paper proposes an approach of test data generation from JML specifications. This approach is based on a randomly generation method of test data.In [12], the paper focuses on the constraints resolution method to reduce number of test data for OO programs.

In [13], authors show that the ACO algorithm can be reformed and used to generate test data for white box testing. In order to generate test input values, they define and apply some techniques such as pheromone update, local transfer and global transfer. In [14], the paper presets an approach to generate test input values by using the Bi-Objective function. The objective function is based on genetic algorithm and is used to produce large spatial distribution of input space. In addition they apply the Clustering method to reduce the time of error finding ability in a test data generation.

In [15], they use metaheuristic algorithms to propose a fitness function to generate test data for the Simulink models. The new fitness function is based on the mutation techniques and has some useful features. Furthermore, this fitness function can be used to improve the mutation score in the Simulink environment. In [16], authors present an approach of test data generation based on the coverage optimization. They propose to do so by MOALO algorithm (Multi-Objective Ant Lion Optimization). This algorithm can be used to improve the coverage of paths.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

3. MODEL OF CONSTRAINTS FOR CONFORMITY TESTING

The work of [6,9] can be used for testing the conformity of an overriding method in a derived classes during the inheritance operation by using constraint model of basic classes and constraint propagation.

3.1 Model of constraint for basic classes

The conformity contract (Fig.1) can be represented by the constraint model *H*.

Definition:
The conformity constraint H of a method $m(x_1,x_2,\ldots,x_n)$ of a class C is a property of the pair (x,o) ($x=(x_1,x_2,\ldots,x_n)$ is the vector of input parameters and o is the receiver object) such that:

$$H(x,o):\left[P(x,o)\wedge Inv(o_{(bef)})\right]\Rightarrow\left[Q(x,o)\wedge Inv(o_{(aft)})\right],(x,o)\in E\times I_c$$

Where $o_{(bef)}$ is the class object o in the state before the calling of the method m( ) and $o_{(aft)}$ is the class object o in the state after the calling of the method m( ) (Fig.3).

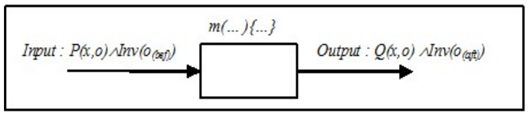


Figure 3. Input-Output constraints of a method m( )

3.2. Model of Conformity Testing in Inheritance

In [6,9] we have used the model of constraint *H* for testing the conformity of methods in derived classes (Fig.4).

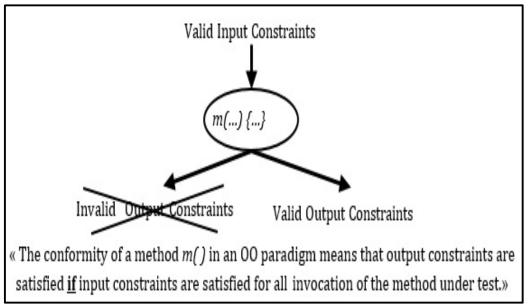

Valid Input Constraints
m(...){...}
Invalid Output Constraints
Valid Output Constraints
« The conformity of a method m( ) in an OO paradigm means that output constraints are satisfied if input constraints are satisfied for all invocation of the method under test.»

Figure 4. Principle of conformity testing

- Constraints propagation in inheritance

We consider a method *m* of a class C_2 which inherits from the class C_1 such that *m* overrides a method of C_1. The original method and its overriding method in the subclass C_2 will be denoted respectively by $m^{(1)}$, $m^{(2)}$ (Fig.5).

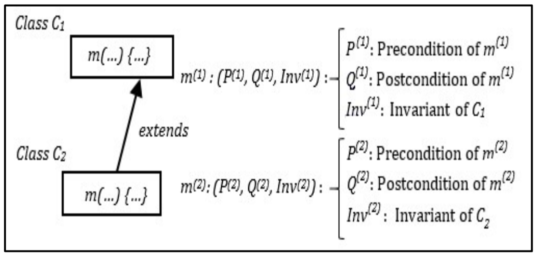

Class C1
m(...) {...}
m(1) : (P(1), Q(1), Inv(1)) :
P(1): Precondition of m(1)
Q(1): Postcondition of m(1)
Inv(1): Invariant of C1
extends
Class C2
m(...) {...}
m(2): (P(2), Q(2), Inv(2)) :
P(2): Precondition of m(2)
Q(2): Postcondition of m(2)
Inv(2): Invariant of C2

Figure 5. Constraints of (Overridden method $m^{(1)}$, Overriding method $m^{(2)}$)

The problem of behavioural constraints of types (Classes) and subtypes (subclasses) of object oriented programs is resolved by Meyer [1,2,3] and Liskov, Wing [17] (Fig.6).

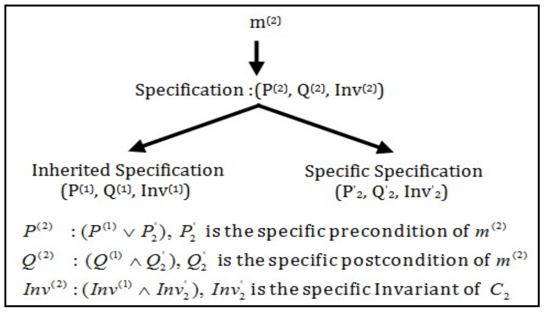

m(2)
Specification :(P(2), Q(2), Inv(2))
Inherited Specification (P(1), Q(1), Inv(1))
Specific Specification (P'2, Q'2, Inv'2)
P(2) : (P(1) ∨ P'2), P'2 is the specific precondition of m(2)
Q(2) : (Q(1) ∧ Q'2), Q'2 is the specific postcondition of m(2)
Inv(2) : (Inv(1) ∧ Inv'2), Inv'2 is the specific Invariant of C2

Figure 6. Specification of an Overriding method $m^{(2)}$

In this approach, the specification ($P^{(2)}$,$Q^{(2)}$,$Inv^{(2)}$) of the overriding method $m^{(2)}$ is constituted by two specifications (Fig.6).

- Constraint of conformity testing
  - Conformity testing of overridden methods [6,9]:
    - The overridden method $m^{(1)}$ is in conformity with its specification if:

      $\forall(x,o)\in E\times I_{C1}:H^{(1)}(x,o)$.
    - The overridden method $m^{(1)}$ is not in conformity with its specification if:

      $\exists(x,o)\in E\times I_{C1}:\ \overline{H^{(1)}(x,o)}$.
  - Conformity testing of overriding methods [6,9]:
    - The overriding method $m^{(2)}$ is in conformity with its specification if:

      $\forall(x,o)\in E\times I_{C2}:H^{(2)}(x,o)$.
    - The overriding method $m^{(2)}$ is not in conformity with its specification if:

      $\exists(x,o)\in E\times I_{C2}:\ \overline{H^{(2)}(x,o)}$.

3.3. Similarity Model

The similarity approach of our previous works [9] is used for assuring if the overriding method $m^{(2)}$ has the same behaviour as its original version $m^{(1)}$

in the superclass according to the inherited specification *$(P^{(1)},Q^{(1)},Inv^{(1)})$*.

For each input value*(x,o)* of the overriding method $m^{(2)}$, we associate the matrix *Similarity (x,o)=(a ,b, a', b')*. The matrix represents the 16 values of the quadruplet *(a,b,a',b')* (Fig.7).
The methods $m^{(1)}$ and $m^{(2)}$are similar to the specification *$(P^{(1)},Q^{(1)},Inv^{(1)})$* if and only if : *(a,b)=(a',b')* and *$(a,b,a',b')\in\{0,1\}^4$* (Fig.7).

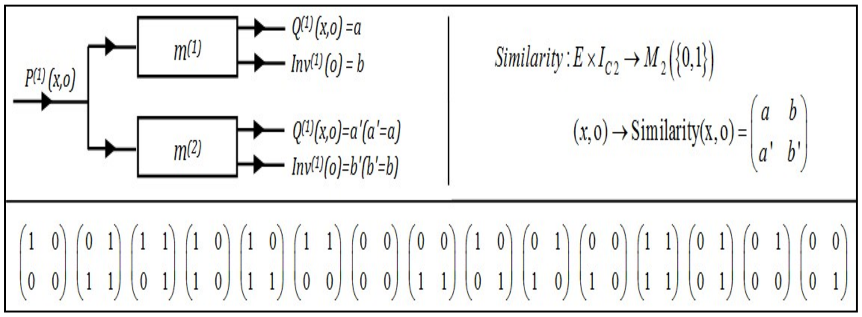

$P^{(1)}(x,o)$
$m^{(1)}$
$m^{(2)}$
$Q^{(1)}(x,o)=a$
$Inv^{(1)}(o)=b$
$Q^{(1)}(x,o)=a'(a'=a)$
$Inv^{(1)}(o)=b'(b'=b)$
$Similarity: E\times I_{C2}\rightarrow M_2(\{0,1\})$
$(x,o)\rightarrow Similarity(x,o)=\begin{pmatrix} a & b \\ a' & b' \end{pmatrix}$
$\begin{pmatrix}1&0\\0&0\end{pmatrix}\begin{pmatrix}0&1\\1&1\end{pmatrix}\begin{pmatrix}1&1\\0&1\end{pmatrix}\begin{pmatrix}1&0\\1&0\end{pmatrix}\begin{pmatrix}1&0\\1&1\end{pmatrix}\begin{pmatrix}1&1\\0&0\end{pmatrix}\begin{pmatrix}0&0\\0&0\end{pmatrix}\begin{pmatrix}0&0\\1&1\end{pmatrix}\begin{pmatrix}1&0\\0&1\end{pmatrix}\begin{pmatrix}0&1\\1&0\end{pmatrix}\begin{pmatrix}0&0\\1&0\end{pmatrix}\begin{pmatrix}1&1\\1&1\end{pmatrix}\begin{pmatrix}0&1\\0&1\end{pmatrix}\begin{pmatrix}0&1\\0&0\end{pmatrix}\begin{pmatrix}0&0\\0&1\end{pmatrix}$

Figure 7. Condition and equivalence partitioning of similarity

The problem of programs conformity is not restricted to classes and subclasses with similar behaviours: if the methods $m^{(1)}$ and $m^{(2)}$ are not similar we are unable to progress in the test process. In the approach of this paper we show that the similarity of behaviour is not obligatory for verifying conformity in sub classes. So the conformity of dissimilar methods can be tested. The purpose of the next section is to generalize the model of [6,9] in order to test the conformity of an overridden and overriding methods ($m^{(1)}$ and $m^{(2)}$) even if methods are not necessarily similar.

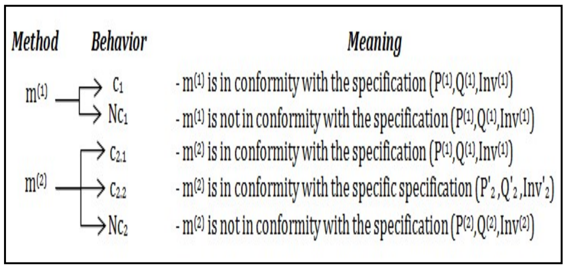
4. APPROACH OF CONFORMITY IN INHERITANCE BY GRAPH OF CONFORMITY STATES

The approach of conformity by similarity presented in the last paragraph can be used to test the conformity of overriding methods in derived classes from test result of overridden methods in the base class. This reusability of test sequences of the base class is only possible if overriding and overridden methods have same basic behaviour [9]. In this paragraph we present an approach of conformity for testing the conformity of overriding methods in derived classes in the general case where overriding and overridden methods are not necessarily similar.

4.1. Input Data Partitioning and Conformity Behaviors

In this work we define the relationship between conformity behaviours and the similarity partitioning. Then we propose an algorithm of conformity test data generation.

- Conformity behaviours (Fig.8)



Method	Behavior	Meaning
$m^{(1)}$	c_1	- $m^{(1)}$ is in conformity with the specification $(P^{(1)},Q^{(1)},Inv^{(1)})$
	Nc_1	- $m^{(1)}$ is not in conformity with the specification $(P^{(1)},Q^{(1)},Inv^{(1)})$
$m^{(2)}$	$c_{2.1}$	- $m^{(2)}$ is in conformity with the specification $(P^{(1)},Q^{(1)},Inv^{(1)})$
	$c_{2.2}$	- $m^{(2)}$ is in conformity with the specific specification (P'_2,Q'_2,Inv'_2)
	Nc_2	- $m^{(2)}$ is not in conformity with the specification $(P^{(2)},Q^{(2)},Inv^{(2)})$

Figure 8. Conformity behaviors of $(m^{(1)}, m^{(2)})$

- Analysis of input data partitioning

In this approach we test the conformity behaviours of $(m^{(1)},m^{(2)})$ by using the similarity partitioning (Fig.7):

$$X=\left\{(x,o)\in E\times I_{C2} : \mathrm{P}^{(1)}(\mathrm{x,o})\wedge\left(\mathrm{Similarity(x,o)}=\begin{pmatrix}1&1\\1&1\end{pmatrix}\right)\right\}$$

$$Y=\left\{(x,o)\in E\times I_{C2} : \mathrm{P}^{(1)}(\mathrm{x,o})\wedge\left(\mathrm{Similarity(x,o)}\in\left\{\begin{matrix}\begin{pmatrix}1&0\\0&0\end{pmatrix} & \begin{pmatrix}0&1\\0&0\end{pmatrix} & \begin{pmatrix}0&0\\0&0\end{pmatrix}\\ \begin{pmatrix}0&0\\1&0\end{pmatrix} & \begin{pmatrix}0&0\\0&1\end{pmatrix} & \begin{pmatrix}0&1\\1&0\end{pmatrix}\\ \begin{pmatrix}1&0\\0&1\end{pmatrix} & \begin{pmatrix}1&0\\1&0\end{pmatrix} & \begin{pmatrix}0&1\\0&1\end{pmatrix}\end{matrix}\right\}\right)\right\}$$

$$Z=\left\{(x,o)\in E\times I_{C2} : \mathrm{P}^{(1)}(\mathrm{x,o})\wedge\left(\mathrm{Similarity(x,o)}\in\left\{\begin{pmatrix}1&1\\1&0\end{pmatrix},\begin{pmatrix}1&1\\0&1\end{pmatrix},\begin{pmatrix}1&1\\0&0\end{pmatrix}\right\}\right)\right\}$$

$$T=\left\{(x,o)\in E\times I_{C2} : \mathrm{P}^{(1)}(\mathrm{x,o})\wedge\left(\mathrm{Similarity(x,o)}\in\left\{\begin{pmatrix}1&0\\1&1\end{pmatrix},\begin{pmatrix}0&1\\1&1\end{pmatrix},\begin{pmatrix}0&0\\1&1\end{pmatrix}\right\}\right)\right\}$$

This parts *X*, *Y*, *Z* and *T* represent behaviours of conformity of the methods $(m^{(1)},m^{(2)})$: $(c_1,c_{2.1})$, (Nc_1, Nc_2), (c_1, Nc_2), $(Nc_1, c_{2.1})$, $(c_1, c_{2.2})$, $(Nc_1,c_{2.2})$ (Fig.9):

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

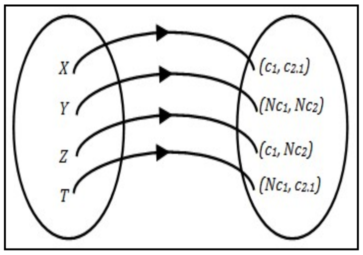


Figure 9. Classes of conformity behaviors of $(m^{(1)},m^{(2)})$

• Conformity testing of overriding methods

The algorithm of conformity testing (Fig.10) is developed for generating input test data of overridden and overriding methods. This algorithm is based on the specification $(P^{(1)},Q^{(1)},Inv^{(1)})$ of the methods $(m^{(1)},m^{(2)})$ and the domain partitioning (X,Y,Z,T) for testing the conformity behaviours of $(m^{(1)},m^{(2)})$ (The constant N is the test threshold limit) (Fig.7 and Fig.9).

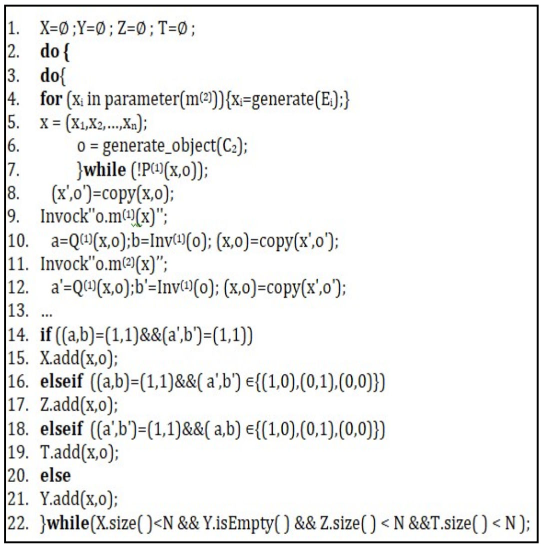

```
X=∅ ;Y=∅ ; Z=∅ ; T=∅ ;
do {
do{
for (xᵢ in parameter(m⁽²⁾)){xᵢ=generate(Eᵢ);}
x = (x₁,x₂,...,xₙ);
        o = generate_object(C₂);
        }while (!P⁽¹⁾(x,o));
   (x',o')=copy(x,o);
Invock"o.m⁽¹⁾(x)";
   a=Q⁽¹⁾(x,o);b=Inv⁽¹⁾(o); (x,o)=copy(x',o');
Invock"o.m⁽²⁾(x)";
   a'=Q⁽¹⁾(x,o);b'=Inv⁽¹⁾(o); (x,o)=copy(x',o');
...
if ((a,b)=(1,1)&&(a',b')=(1,1))
X.add(x,o);
elseif ((a,b)=(1,1)&&( a',b') ∊{(1,0),(0,1),(0,0)})
Z.add(x,o);
elseif ((a',b')=(1,1)&&( a,b) ∊{(1,0),(0,1),(0,0)})
T.add(x,o);
else
Y.add(x,o);
}while(X.size( )<N && Y.isEmpty( ) && Z.size( ) < N &&T.size( ) < N );
```

Figure 10. Algorithm of conformity testing

4.2. Graph of Conformity States of $(m^{(1)},m^{(2)})$

The graph is a powerful way to model various types of processes and relations in biological, physical and software systems. In computer science, many practical problems of software testing can be represented by graphs. The test data generation of inheritance can be represented by a directed graph, in which the vertices represent conformity states and directed edges represent transitions from one state to another.

In our approach, graphs are used to represent results of conformity test of overridden and overriding methods.

• Conformity states

Definition
The conformity state of $(m^{(1)},m^{(2)})$ is a value of the 4-tuple (x, y, z, t), where $x = |X|$, $y = |Y|$, $z = |Z|$ and $t = |T|$ are the cardinal number of the sets X, Y, Z and T after the current iteration of the do… while loop of the test data generation algorithm (Fig.10).

There are three values for each sets (X, Y, Z and T):
- Empty set ($|X|=0$): $x=0$.
- Saturated set ($|X|=N$): $x=N$ (N is the test threshold limit).
- Nonempty set ($|X|<N$ and $|X|\neq 0$): $x=\varepsilon$.

With 3 values $(x,y,z,t)\in\{0,\varepsilon,N\}^4$, we will have 81 conformity states (Fig.7 and Fig.9) (Table 1 and Table 2).

Table1. Realizable states of conformity

	Conformity state	Values	Behavio of $(m^{(1)},m^{(2)})$
Practicable states	State 1	$\{(0,,0,0),(0,\varepsilon,0,\varepsilon), (0,,\varepsilon,0),(0,\varepsilon,\varepsilon,\varepsilon), (\varepsilon,,0,0),(\varepsilon,\varepsilon,,0,\varepsilon), (\varepsilon,,\varepsilon,0),(\varepsilon,\varepsilon,\varepsilon,\varepsilon)\}$	(Nc_1, Nc_2)
	State 2	$\{(0,0,N,0), (\varepsilon,0,N,0)\}$	(c_1, Nc_2)
	State 3	$\{(0,0,0,N), (\varepsilon,0,0,N)\}$	$(Nc_1, c_{2.1})$
	State 4	$(N,0,0,0)$	$(c_1, c_{2.1})$
Paradoxial states	State e_1	$\{(0,0,N,\varepsilon), (\varepsilon,0,N,\varepsilon)\}$	(Nc_1, Nc_2)
	State e_2	$\{(0,0,\varepsilon,N), (\varepsilon,0,\varepsilon,N)\}$	(Nc_1, Nc_2)
	State e_3	$(N,0,\varepsilon,\varepsilon)$	(Nc_1, Nc_2)
	State f_1	$(N,0,0,\varepsilon)$	$(Nc_1, c_{2.1})$
	State f_2	$(N ,0,\varepsilon,0)$	(c_1, Nc_2)

The conformity states (e_1, e_2, e_3, f_1, f_2) have a paradoxical relationship with the threshold limit of test N. For example:

$$f_1 = (N,0,0,\varepsilon): \begin{cases} |X| = N \Rightarrow (m^{(1)},m^{(2)})\ \mathit{has\ the}\ \text{behavior}\ (c_1, c_{2.1}) \\ \quad \Rightarrow m^{(1)}\ \text{is in conformity with its specification.} \\ |T| = \varepsilon \Rightarrow \exists(x,o)\text{: for this (x,o)} \\ \quad \Rightarrow m^{(1)}\ \text{is not in conformity with its specification.} \end{cases}$$

It follows that the paradoxical states are convenient for detecting errors of the threshold limit N of test.

Table 2. Incomplete states of conformity of $(m^{(1)},m^{(2)})$

State 0 : $(0,0,0,0)$	State p : $(0,0,0,\varepsilon)$
State i : $(\varepsilon,0,\varepsilon,0)$	State q : $(0,0,\varepsilon,0)$
State j : $(\varepsilon,0,0,\varepsilon)$	State r : $(0,0,\varepsilon,\varepsilon)$
State k : $(\varepsilon,0,0,0)$	State s : $(\varepsilon,0,\varepsilon,\varepsilon)$

• Graph of conformity states

Definition
A graph of conformity states of $(m^{(1)},m^{(2)})$ (Fig.11) is a graph that satisfies the following conditions:
- Each vertex represents a conformity state of

$(m^{(1)},m^{(2)})$.

- Each edge represents a transition from a conformity state to another conformity state or a loop that is an edge that connects a conformity state to itself.

- The source vertex in the graph represents the input state and the sink vertex represents the output state of $(m^{(1)},m^{(2)})$.

The sink vertex of the graph (Fig.11) represents the paradoxical state $e_3=(N,0,\varepsilon,\varepsilon)$. And then the methods $(m^{(1)},m^{(2)})$ have the behavior (Nc_1,Nc_2).

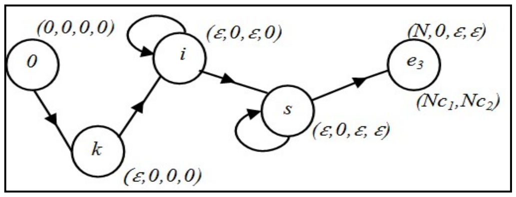


Figure 11. Graph of conformity states

5. EVALUATION

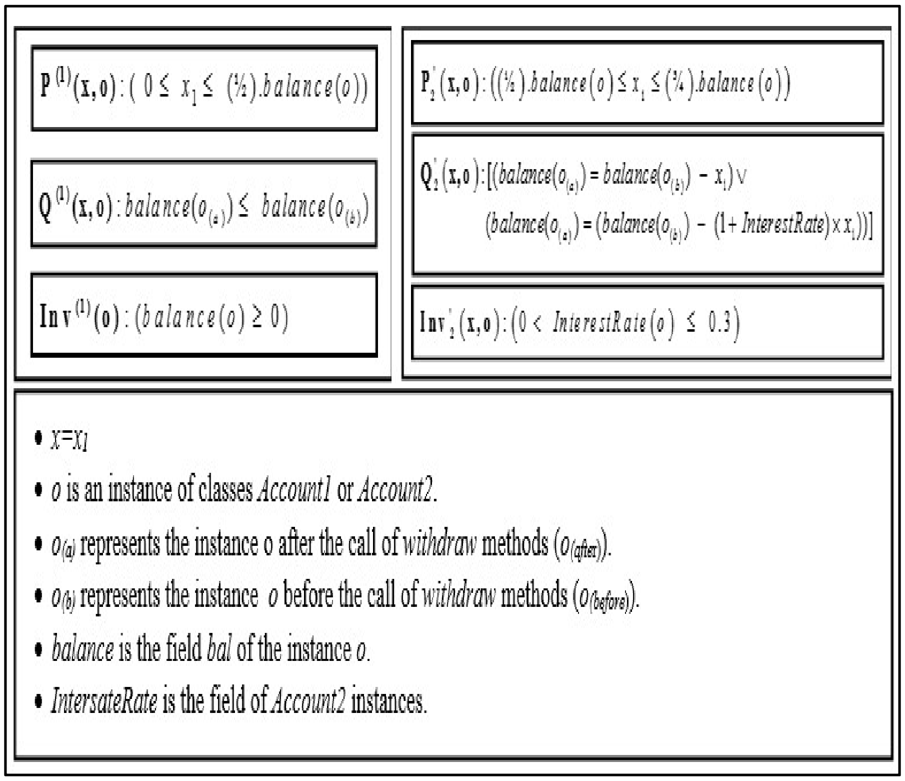
In this section we present an example of test data generation of the overridden method *withdraw*$^{(1)}$ and the overriding method *withdraw*$^{(2)}$ for two java classes (Fig.12).

```
class Account1
{    protected double bal;
      /* bal is the account balance */
      public Account1(double x1){
      this.bal=x1;
      }
      public void withdraw (int x1){
      this.bal=this.bal - x1;
      }
}
```

```
class Account2 extends Account1
{    private double InterestRate;
      public Account2(double x1, double x2)
      {super(x1); this.InterestRate=x2;}
      public void withdraw (int x1)
      {super.withdraw(x1);
      if ((x1>bal) && (x1<(bal/InterestRate)))
      this.bal=this.bal-(this.InterestRate)*x1;
      InterestRate = InterestRate/2;}
}
```

Figure 12. Java implementation of withdraw methods

In the figure 13 we present the specification of the overridden and the overriding methods *withdraw*:



$P^{(1)}(x,o):(0\leq x_1\leq (½).balance(o))$

$Q^{(1)}(x,o):balance(o_{(a)})\leq balance(o_{(b)})$

$Inv^{(1)}(o):(balance(o)\geq 0)$

$P'_2(x,o):((½).balance(o)\leq x_1\leq (¾).balance(o))$

$Q'_2(x,o):[(balance(o_{(a)})=balance(o_{(b)})-x_1)\vee$
$(balance(o_{(a)})=(balance(o_{(b)})-(1+InterestRate)\times x_1))]$

$Inv'_2(x,o):(0<InterestRate(o)\leq 0.3)$

- $x=x_1$
- *o* is an instance of classes *Account1* or *Account2*.
- $o_{(a)}$ represents the instance o after the call of *withdraw* methods ($o_{(after)}$).
- $o_{(b)}$ represents the instance *o* before the call of *withdraw* methods ($o_{(before)}$).
- *balance* is the field *bal* of the instance *o*.
- *IntersateRate* is the field of *Account2* instances.

Figure 13. specification $(P^{(1)},Q^{(1)},Inv^{(1)})$ and (P'_2,Q'_2,Inv'_2) of methods $(withdraw^{(1)},withdraw^{(2)})$

• Test Data Generation of Conformity Constraint

The table 3 illustrates an example of test data generation of methods *withdraw*$^{(2)}$ and *withdraw*$^{(1)}$ (x_1 and *balance(o)* are in *]-200,200[*; The threshold limit *N=100* and *X=Y=Z=T=∅*).

Table 3. Result of a conformity test of (withdraw$^{(1)}$, withdraw$^{(2)}$)

Iteration number	x_1	*O*	|*X*|	|*Y*|	|*Z*|	|*T*|
0: input state	-	-	*0*	*0*	*0*	*0*
1	*73*	*Account2(168,0.05)*	*1*	*0*	*0*	*0*
2	*42*	*Account2(99,0.13)*	*2*	*0*	*0*	*0*
3	*39*	*Account2(117,0.19)*	*3*	*0*	*0*	*0*
4	*48*	*Account2(129,0.03)*	*4*	*0*	*0*	*0*
5	*27*	*Account2(74,0.1)*	*5*	*0*	*0*	*0*
...	...	...	...	...	...	...
49	*86*	*Account2(191,0.25)*	*49*	*0*	*0*	*0*
50	*53*	*Account2(158,0.17)*	*50*	*0*	*0*	*0*
51	*61*	*Account2(146,0.06)*	*51*	*0*	*0*	*0*
...	...	...	...	...	...	...
81	*33*	*Account2(83,0.2)*	*81*	*0*	*0*	*0*
82	*45*	*Account2(111,0.27)*	*82*	*0*	*0*	*0*
83	*24*	*Account2(101,0.12)*	*83*	*0*	*0*	*0*
...	...	...	...	...	...	...
96	*90*	*Account2(185,0.08)*	*96*	*0*	*0*	*0*
97	*97*	*Account2(198,0.24)*	*97*	*0*	*0*	*0*
98	*49*	*Account2(126,0.15)*	*98*	*0*	*0*	*0*
99	*18*	*Account2(66,0.08)*	*99*	*0*	*0*	*0*
100	*81*	*Account2(177,0.11)*	*100*	*0*	*0*	*0*

The graph of conformity states of (*withdraws*$^{(1)}$, *withdraw*$^{(2)}$) is as follows (Fig.14):

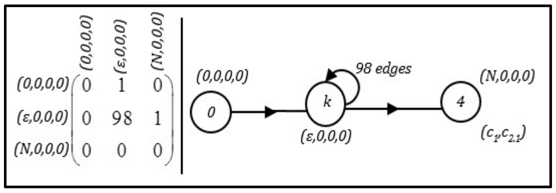


Figure 14. Graph of conformity states

In this example (*withdraw*$^{(1)}$, *withdraw*$^{(2)}$) changes from the input state *0* to the incomplete state *k=(ε,0,0,0)*, and we maintain the state *k* for 98 iterations.

In the end we have the output state: *State 2= (N,0,0,0)*.

The sink vertex represents the behaviour (c_1, $c_{2.1}$). Therefore, the methods *withdraw*$^{(1)}$ and *withdraw*$^{(2)}$ are in conformity to the specification ($P^{(1)}$,$Q^{(1)}$,$Inv^{(1)}$).

6. CONCLUSION

This paper proposes an approach of test data generation to validate conformity contracts of OO programs. Our work presents a formal model based on graph theory and equivalence partitioning technique to simplify the conformity verification process of OO classes.

Our model of graphs is an important way to represent and understand conformity behaviours of subclasses of inheritance mechanism. This approach can be applied to overriding and overridden methods even if super and sub classes do not have similar behaviours. The first approach of this work is an algorithm of test data generation based on a similarity partitioning for testing the conformity contract of an OO model. The second approach is a way to generate test data of conformity by using adjacency matrix and graphs of conformity states. This paper shows how the graph of states and equivalence partitioning can be used to reduce the test data generation and therefore, to improve software testing.

REFERENCES

[1] B. Meyer, "*Applying 'design by contract'*" , *Computer*, vol. 25, no. 10, pp. 40-51, 1992.

[2] B. Meyer, *Object-oriented software construction,* Upper Saddle River, N.J.: Prentice Hall PTR, 1997.

[3] B. Meyer, *Eiffel*, New York: Prentice-Hall, 1998.

[4] B. K. Aichernig and P. A. P. Salas, "Test case generation by OCL mutation and constraint solving", *Fifth International Conference on Quality Software (QSIC'05)*, Melbourne, Victoria, Australia, 2005, pp. 64-71.

[5] Y. Cheon, Z. Cao, and K. Rahad, "Writing JML specifications using Java 8 streams", *University of Texas at El Paso*, vol. 500, pp. 79968-0518, 2016.

[6] K. Louzaoui and K. Benlhachmi, "An Optimal Model of Conformity Constraints of Inheritance for an Object Oriented Specification", *In International Journal of Tomography and simulation*, Vol. 30, No. 3, pp. 86-102, 2017.

[7] K. Louzaoui and K. Benlhachmi, "A Robustness Testing Approach for an Object Oriented Model", *Journal of Computers*, vol. 12, no. 4, pp. 335-353, 2017.

[8] K. Louzaoui, "An Optimal Constraint Model of Robustness Behavior for Object Oriented Programs", *2018 International Conference on Electronics, Control, Optimization and Computer Science (ICECOCS)*, Kenitra, 2018, pp. 1-6.

[9] K. Benlhachmi and M. Benattou, "Similar Behaviours and Conformity Testing in Inheritance for an Object Oriented Model", *In IADIS International Journal on Computer Science and Information Systems*, Vol. 9, issue 1,pp. 30-42, 2014.

[10] V. Rahli, D. Guaspari, M. Bickford and R. Constable, "EventML: Specification, verification, and implementation of crash-tolerant state machine replication systems", *Science of Computer Programming*, vol. 148, pp. 26-48, 2017.

[11] Y. Cheon and C. E. Rubio-Medrano. "Random Test Data Generation for Java Classes Annotated with JML Specifications". *In Proceedings of the 2007 International Conference on Software Engineering Research and Practice Las Vegas*, Nevada, vol 2,June 2007, pp. 385–392.

[12] Y. Cheon, A. Cortes, M. Ceberio, and G. T. Leavens, "Integrating Random Testing with Constraints for Improved Efficiency and Diversity". *In Proceedings of SEKE 2008, The 20-th International Conference on Software Engineering and Knowledge Engineering*, San Francisco, CA, July 2008, pp. 861–866.

[13] C. Mao, L. Xiao, X. Yu and J. Chen, "Adapting ant colony optimization to generate test data for software structural testing", *Swarm and Evolutionary Computation*, vol. 20, pp. 23-36, 2015.

[14] R. L. Bai and C. P. Indumathi, "Test data generation using bi-objective function", *2016 International Conference on Advanced Communication Control and Computing Technologies (ICACCCT)*, Ramanathapuram, 2016, pp. 650-654.

[15] L. Hanh, N. Binh and K. Tung, "A Novel Fitness function of metaheuristic algorithms for test data generation for simulink models based on mutation analysis", *Journal of Systems and Software*, vol. 120, pp. 17-30, 2016.

[16] M. Singh, V. M. Srivastava, K. Gaurav and P. K. Gupta, "Automatic test data generation based on multi-objective ant lion optimization algorithm", *2017 Pattern Recognition Association of South Africa and Robotics and Mechatronics (PRASA-RobMech)*, Bloemfontein, 2017, pp. 168-174.

[17] B.H. Liskov and J.M. WING, "Behavioral subtyping using invariants and constraints", *Technical Report CMU CS-99-156, School of Computer Science*, Carnegie Mellon University, July 1999.