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

KHALID BENLHACHMI 1, KHADIJA LOUZAOU1 2

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

E-mail: 1khalid.benlhachmi@uit.ac.ma , 2khadija.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 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 \((P, Q, \text{Inv})\) (Fig.1). The specification \((P, Q, \text{Inv})\) 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]… .

![Figure 1. Constraints and Conformity contract of an OO program](image-url)
In an OO paradigm, the conformity contract is a property \( H \) defined for all elements of the input domain \( E \times I \) of the program under test:

\[
H : E \times I \rightarrow \{true, false\}
\]

\((x,o) \rightarrow H(x,o) : P(x,o) \land \text{Inv}(o_{\text{inv}}) \Rightarrow Q(x,o) \land \text{Inv}(o_{\text{inv}})\)

This conformity constraint of the program under test is satisfied if: “For all invocation of the program, output specifications (\( Q \) and \( \text{Inv} \)) are satisfied if input specifications (\( P \) and \( \text{Inv} \)) 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).

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 safety 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 presents 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.
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 $m(x_1,x_2,...,x_n)$ of a class $C$ is a property of the pair $(x,o)$ (where $x=(x_1,x_2,...,x_n)$ is the vector of input parameters and $o$ is the receiver object) such that:

$$H(x,o)[P(x,o) \land \text{Inv}(o_{(a)})] \rightarrow [Q(x,o) \land \text{Inv}(o_{(b)})], (x,o) \in E \times I_c$$

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

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).

- 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).

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

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)

<table>
<thead>
<tr>
<th>Method</th>
<th>Behavior</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>(m^{(1)})</td>
<td>(C_1)</td>
<td>(m^{(1)}) is in conformity with the specification (P^{(1)}, Q^{(1)}, Inv^{(1)})</td>
</tr>
<tr>
<td>(m^{(1)})</td>
<td>(N_{C_1})</td>
<td>(m^{(1)}) is not in conformity with the specification (P^{(1)}, Q^{(1)}, Inv^{(1)})</td>
</tr>
<tr>
<td>(m^{(2)})</td>
<td>(C_2)</td>
<td>(m^{(2)}) is in conformity with the specification (P^{(2)}, Q^{(2)}, Inv^{(2)})</td>
</tr>
<tr>
<td>(m^{(2)})</td>
<td>(N_{C_2})</td>
<td>(m^{(2)}) is not in conformity with the specification (P^{(2)}, Q^{(2)}, Inv^{(2)})</td>
</tr>
</tbody>
</table>

- 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):

\[X = \left\{(x, o) \in E \times I_{c_1} : P^{(1)}(x, o) \land \left(\text{Similarity}(x, o) = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}\right)\right\}\]
\[Y = \left\{(x, o) \in E \times I_{c_2} : P^{(1)}(x, o) \land \left(\text{Similarity}(x, o) = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}\right)\right\}\]
\[Z = \left\{(x, o) \in E \times I_{c_1} : P^{(2)}(x, o) \land \left(\text{Similarity}(x, o) = \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}\right)\right\}\]
\[T = \left\{(x, o) \in E \times I_{c_2} : P^{(2)}(x, o) \land \left(\text{Similarity}(x, o) = \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix}\right)\right\}\]

This parts \(X, Y, Z\) and \(T\) represent behaviours of conformity of the methods \((m^{(1)}, m^{(2)})\) : \((C_1, C_2), (N_{C_1}, N_{C_2}), (c_1, c_{2.1}), (N_{C_1}, c_{2.1}), (c_1, c_{2.2}), (N_{C_1}, c_{2.2})\) (Fig.9):
• 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).

1. X:={0, 1}, Y:={0, 1}, Z:={0, 1}, T:={0, 1};
2. do
3. do;
4. for x in parameter(m(1)) do generate(E);
5. x := (x, x, x, x);
6. o := generate_object(E);
7. while (P ⊢ (x, o)) do
8. if (o, m(1)(x)) then copy(o);
9. if (x, m(1)(x)) then copy(x);
10. if (x, m(1)(x)) then copy(x);
11. if (x, m(1)(x)) then copy(x);
12. if (x, m(1)(x)) then copy(x);
13. ... if (x, m(1)(x)) then copy(x);
14. if (x, m(1)(x)) then copy(x);
15. if (x, m(1)(x)) then copy(x);
16. else if (x, m(1)(x)) then copy(x);
17. if (x, m(1)(x)) then copy(x);
18. else if (x, m(1)(x)) then copy(x);
19. if (x, m(1)(x)) then copy(x);
20. else
21. add(o);
22. while (E size() = N && Y in Empty() && Z in Empty() && T in Empty()) do 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.

- 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
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 \(e_3 = (N, 0, \epsilon, \epsilon)\). And then the methods \((m^{(1)}, m^{(2)})\) have the behavior \((Nc_1, Nc_2)\).

5. EVALUATION

In this section we present an example of test data generation of the overridden method \(\text{withdraw}^{(1)}\) and the overriding method \(\text{withdraw}^{(2)}\) 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) {
            if (bal < (bal / InterestRate)) {
                this.bal -= this.bal - (bal / InterestRate) * x1;
                InterestRate = InterestRate / 2;
            }
        }
    }
}
```

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

```plaintext
\[
\begin{align*}
P^{(1)}(x, o) : (0 \leq x_1 \leq (\frac{1}{2})\cdot \text{balance}(o)) \\
P^{(2)}(x, o) : (\frac{1}{2})\cdot \text{balance}(o) \leq x_1 \leq (\frac{3}{4})\cdot \text{balance}(o) \\
Q^{(1)}(x, o) : \text{balance}(o) \leq \text{balance}(o) \\
Q^{(2)}(x, o) : ((\text{balance}(o) = \text{balance}(o) - x_1) \lor \\
(\text{balance}(o) = (\text{balance}(o) - (1 + \text{InterestRate}) \cdot x_1))) \\
Inv^{(1)}(o) : (\text{balance}(o) \geq 0) \\
Inv^{(2)}(o) : (0 \leq \text{InterestRate}(o) \leq 0.3)
\end{align*}
\]```

- \(x = x_1\)
- \(o\) is an instance of classes \(\text{Account1}\) or \(\text{Account2}\).
- \(o_{\text{after}}\) represents the instance \(o\) after the call of \(\text{withdraw}\) methods \((o_{\text{after}})\).
- \(o_{\text{before}}\) represents the instance \(o\) before the call of \(\text{withdraw}\) methods \((o_{\text{before}})\).
- \(\text{balance}\) is the field \(\text{bal}\) of the instance \(o\).
- \(\text{InterestRate}\) is the field of \(\text{Account2}\) instances.
Test Data Generation of Conformity Constraint

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

| 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 \((\text{withdraws}^{(1)}, \text{withdraw}^{(2)})\) is as follows (Fig.14): 

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure14.png}
\caption{Graph of conformity states}
\end{figure}

In this example \((\text{withdraw}^{(1)}, \text{withdraw}^{(2)})\) changes from the input state 0 to the incomplete state \(k=(\varepsilon, 0, 0, 0)\), and we maintain the state \(k\) for 98 iterations. 
In the end we have the output state: \(\text{State 2}=(N, 0, 0, 0)\). 

The sink vertex represents the behaviour \((c_1, c_2)\). 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