A Formal Model of Robustness Testing for an Object-Oriented Specification

Khadija Louzaoui¹ and Khalid Benlhachmi²

¹,²Department of Computer Science, Faculty of Sciences, Ibn Tofail University, Kenitra, Morocco
¹Khadija.louzaoui@gmail.com, ²Benlhachmi11@yahoo.fr

Abstract

The work presented in this paper proposes a formal model of constraints for testing the conformity of an implementation from its specification. The principal idea of our approach is based on an equivalence partitioning of input domains for each method type in an object oriented (OO) paradigm for detecting the different classes of errors. The main contribution of our approach is the use of invalid data which do not satisfy the precondition constraints for testing the robustness of entities in an OO model. Indeed, the first objective of the proposed work is to develop a theoretical model of constraint in order to test the conformity of classes. The second objective of our approach is to detect anomalies in invalid input data which induce valid output constraints. The implementation of this approach is based on a random generation of test data and analysis by formal proof.

Keywords: Formal specification, conformity testing, robustness testing, valid data, invalid data, test data generation, inheritance, constraint resolution

1. Introduction

The guarantee of software qualities beyond those that can be achieved by syntax checking, type checking, and testing, requires an important application of mathematical constraints on computer programs. Formal specifications can be used to precisely describe the desired properties of a program, and verification techniques allow one to prove that a program meets its specification. In this context formal specifications are an important step towards more reliable software because they improve development methodology and abstract from implementation details, which simplifies reasoning about software systems and allows verifiers to rule out coding errors. Specification techniques are most cost-effective in the development of critical systems where safety, reliability and security are particularly important. They may also be used to define contracts between users and programs. Indeed, in an Object Oriented (OO) model a formal specification is the set of input and output constraints used to describe by logical equations properties of programs (Figure 1). Precondition (P), postcondition (Q) and invariant (Inv) are the main used constraints in an OO specification (Figure 2), and can be written by using specific languages as OCL [1] and JML [2].
Design by constraints is introduced by Meyer [3, 4] and defines the contract between users and programs. In this context the conformity of a program in an OO paradigm means that the output constraints are satisfied if the input constraints are satisfied for all invocation of the program under test (Figure 3). In an OO testing, the test data generation is based on the valid input data that satisfy the precondition constraints, and is used to testing the conformity of a class implementation from its specification. The generated data from the values domain with respect to the well defined constraints can be generated randomly or using constraints resolution.

Figure 1. Relationship between Program and its Specification

Figure 2. Constraints of the Method m()

Figure 3. User and Program Contracts

The main contribution of this work is the definition of a complementary test based on invalid input data that do not satisfy the precondition constraint of the method under test.
The idea is to integrate the invalid data in the testing process for strengthening the conformity testing of OO classes. Indeed, the analysis of program behavior towards invalid input data is used to enrich the concept of test and to detect design anomalies.

The test oracles do not define the behavior of implementations towards invalid input data and they are restricted to conformity testing by generating only data meeting the precondition constraints. In our approach we complete the testing process by introducing a test based on the invalid data to measure the robustness of OO programs. In this context, we specify anomalies of constraints which are due to input data that meet the contract of program under test even if input constraints are invalid (Figure 4). These input data induce two problems:

- A problem on the relevance of specifications: incomplete precondition (satisfied contract in outside of the valid domain (Figure 4)).
- A problem on the program robustness (the code can ensure output contracts for an invalid input value (Figure 4)).

This paper is organized as follows: in Section 2 we present related works and similar approaches for generating test data from a formal specification, in sections 3 and 4 we describe theoretical aspects of our test process, and we define our test formal model of constraints, partitions deduced from the formal model. In section 5 we present how the formal model and domain partitioning can be used to generate data for testing the conformity of methods in an OO class, in section 6 we present our approach of robustness testing that strengthens the conformity testing, and we show how the robustness approach uses the domain partition for generating test data, and finally we describe our approach with an example of conformity and robustness testing for an OO model.

2. Related Work

Most works have studied the problem of test data generation and formal specification for OO programs. These works show how the programs conformity can be tested by using white box testing that takes into account the internal mechanism of a system or black box testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions.

In [1], the authors present a method based on the constraints resolution for test cases generation with error anticipation in the methods specification. In [2], the approach presents a model-based framework for the symbolic animation of object-oriented specifications. This technique can be applied to Java Modeling Language (JML) specifications, making it possible to animate Java programs that only contain method interfaces and no code. In [5], the authors propose a generation of test data from formal specifications using the Object Constraint Language (OCL).

In [6] the authors present a test derivation and selection method based on a model of communicating processes with inputs, outputs and data types, which is closer to actual implementations of communication protocols.

In [7], we have developed a basic model for the concept of methods similarity, the test is based only on a random generation of input data. In [8], we have generalized the basic
model of similar behaviors using constraints propagation, equivalence partitioning and formal proof.

In [9], we have based on similar behaviors for testing the conformity of overriding methods in inheritance. This approach shows that the test cases developed for testing an original method can be used for testing its overriding method in a subclass and then the number of test cases can be reduced considerably.

In [10], the authors propose a randomly generation of test data from a JML specification of class objects. They classify the methods and constructors according to their signature (basic and extended constructors, mutator, and observer) and for each type of individual method of class, a generation of test data is proposed. In [11], the authors use the constraints resolution principle to reduce the values of testing data for limited domain types and use a random generation for other data types. In [12] the authors propose a theoretical framework for model based robustness testing together with an implementation within the If validation environment.

In [13], the paper describes specially the features for specifying methods, related to inheritance specification; it shows how the specification of inheritance in JML forces behavioral sub-typing. In order to study the effectiveness and performance of random test techniques, the works presented in [14] propose to test the conformity of methods using experimental proofs. In [15], the authors developed a testing tool called JCrasher for random testing of Java classes. Tool oriented, the authors propose in [16] a configurable unit testing tool of Java classes specified in JML.

In [17] the paper gives a description of testing methods based on algebraic specifications, and a brief presentation of some tools and case studies, and presents some applications to other formal methods involving data types. In [18] the authors have studied the multi-objective test data generation problem. The authors in [19] present a robustness modeling methodology that allows modeling robustness behavior as aspects. The goal is to have a complete and practical methodology that covers all features of state machines and aspect concepts necessary for model-based robustness testing. In [20] the authors present a survey of some of the most prominent techniques of automated test data generation, including symbolic execution, model-based, combinatorial, adaptive random and search-based testing.

Test oracles and current standards of verification do not define implementations behaviors towards invalid input data and they are restricted to conformity testing by generating only data meeting precondition constraints. In our approach we complete the testing process by introducing a test based on the invalid data to measure the robustness of OO programs.

3. Formal Model of Constraint

This section presents the definition of a new kind of constraint for modeling the specification of methods in OO classes. In this context, we propose a constraint model that contains the precondition $P$, postcondition $Q$, and the invariant $Inv$ (Figure 2) into a single logical formula. This model should translate algebraically the contract between users and programs.

Let $C$ be a class, and $m$ be a method of $n$ arguments $x = (x_1, x_2, ..., x_n)$. We define for each argument $x_i$ its domain of values $E_i$. We put $E = (E_1 \times E_2 \times \ldots \times E_n)$ where $E$ is the domain of input vectors of the method $m$.

The main idea of this relationship is based on the following interpretation:

If the user respects his own part of the contract by invoking a method with arguments satisfying the precondition $P$, then the method must necessarily satisfy the postcondition $Q$ after the call. Concerning the invariant, it must be satisfied before and after the method call.
We consider that $att_1, att_2, \ldots, att_m$ the attributes of the class $C$, and state the state of an object $o$ of $C$: $state(o)$ is defined by the set of values of its attributes:

$$state(o) = (value(att_1), value(att_2), \ldots, value(att_m))$$

The object is an entity that may change its state if a value of its attributes is modified. We concentrate mainly on state of the object $o$ before (State before) and after (State after) the calling of the method (Figure 5).

**Definition 1**

We define the constraint $H$ of a method $m$ of class $C$ as a property of the pair $(x, o)$ ($x$ is the input vector and $o$ is the receiver object) such that:

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

Where:

- $I_c$ is the set of instances of the class $C$.
- $o_{(bef)}$ is the class object $o$ in the state before the calling of the method $m()$.
- $o_{(aft)}$ is the class object $o$ in the state after the calling of the method $m()$.

The logical implication in the proposed formula means: each call of method with $(x, o)$ satisfying the precondition $P$ and the invariant $Inv$ before the call, $(x, o)$ must necessarily satisfy the post-condition $Q$ and the invariant after the call (Figure 5). In the context of OO modeling, this constraint can be reduced if we consider that all invocations of method $m$ are done with a valid object $o$ satisfying the invariant (instantiated by a valid constructor). We deduce that the invariant of object $o$ in the State-Before is satisfied:

$$Inv(o_{(bef)}) = 1.$$ 

Therefore:

$$H(x, o): P(x, o) \implies \left[ Q(x, o) \land Inv(o_{(aft)}) \right], (x, o) \in E \times I_c \quad (1)$$

The evaluation of the constraint $H$ (for $(x, o) \in E \times I_c$) is done in two steps:

- In the input of the method, the evaluation of $P(x, o)$.
- In the output of the method, the evaluation of $Q(x, o)$ and $Inv(o)$ (Figure 6)

**Figure 5. Input-Output Constraints of a Method $m()$**

**Figure 6. Simplified Specification of a Method $m()$**
4. Partition Analysis

The analysis of the input domain of a method is a crucial step in the implementation of tests. Indeed, this analysis allows dividing the domains to locate the potential data which can affect the test problem in order to identify the anomalies origin. In this section, the constraint \( H \) defined above is used in the generation of domain partitions for each type of methods according to the classification proposed in [10]. Indeed, this classification includes four types of methods (\( C_b \): basic constructor, \( C_e \): extended constructor, \( M \): mutator and \( O \): observer)(Figure 10).

The partitions analysis is a technique that can be used to reduce the number of test cases that need to be developed by covering all classes of errors for the module under test. Indeed, the generalized constraint defined above is used in the partition process to deduce the set of domains representing all possible testing values satisfying or not the constraint \( H \). The proposed equivalence partitioning divides the input domain of a program into classes. For each of these equivalence classes, the set of data should be treated the same by the module under test and should produce the same answer. Test cases should be designed so the inputs lie within these equivalence classes.

In this context, we divide firstly the input domain \( E \times I_c \) of the method under test into two sets \( A \) and \( B \) (Figure 7):

\[
A = \{(x, o) \in E \times I_c / H(x, o) = 1\} \quad \text{and} \quad B = \{(x, o) \in E \times I_c / H(x, o) = 0\}
\]

Then, \( A \) can be divided into two subsets \( A_1 \) and \( A_2 \):

\[
A_1 = \{(x, o) \in E \times I_c / (P(x, o), Q(x, o), Inv(o)) = (1,1,1)\} : \text{the domain whose elements (x,o) satisfy } P, Q, \text{ and Inv}.
\]

\[
A_2 = \{(x, o) \in E \times I_c / (P(x, o), Q(x, o), Inv(o)) = (0,?,?)\} , \text{ this domain represents the pairs (x,o) of E×Ic such as the precondition } P \text{ of the method is not satisfied.}
\]

We divide \( A_2 \) (for a false precondition) to 4 domains \( A_{2,1}, A_{2,2}, A_{2,3}, \text{et } A_{2,4} \) such as:

- \( A_{2,1} = \{(x, o) \in E \times I_c / (P(x, o), Q(x, o), Inv(o)) = (0,0,0)\} \)
- \( A_{2,2} = \{(x, o) \in E \times I_c / (P(x, o), Q(x, o), Inv(o)) = (0,0,1)\} \)
- \( A_{2,3} = \{(x, o) \in E \times I_c / (P(x, o), Q(x, o), Inv(o)) = (0,1,0)\} \)
- \( A_{2,4} = \{(x, o) \in E \times I_c / (P(x, o), Q(x, o), Inv(o)) = (0,1,1)\} \)

\( B \) can be divided into 3 subsets \( B_1, B_2 \) and \( B_3 \) (Figure 7):

- \( B_1 = \{(x, o) \in E \times I_c / (P(x, o), Q(x, o), Inv(o)) = (1,1,0)\} \)
- \( B_2 = \{(x, o) \in E \times I_c / (P(x, o), Q(x, o), Inv(o)) = (1,0,1)\} \)
- \( B_3 = \{(x, o) \in E \times I_c / (P(x, o), Q(x, o), Inv(o)) = (1,0,0)\} \)
(A₁, B₁, B₂, B₃) is the partition whose elements satisfy the precondition constraint (i.e., the partition of the valid domain) and (A₂₁, A₂₂, A₂₃, A₂₄) is the partition whose elements do not satisfy the precondition constraint (i.e., the partition of the invalid domain) (Figure 8).

5. Conformity Testing

In this section we use the model of constraint H and partitioning of input domains to describe the concept of conformity of a method in a class of objects. Indeed, the purpose of this section is to model the conformity of a class by a logical equation and implement algorithms for generating test data.

In conformity test, the input values must satisfy the precondition of the method under test. In this sense, we are particularly interested in the valid input values (i.e., the input data (x,o) that satisfy $P$). We assume that the user respects its part of the contract, by giving valid input values (Figure 3). In this context, the proposed test oracle rejects the method call with an invalid input value.

5.1. Formal model of conformity testing

The conformity of a method $m$ in an OO paradigm means that the output constraints are satisfied if the input constraints are satisfied for all invocation of the method under test.

We consider a method $m$ of class $C$ where $o$ is the receiver object and $x$ the parameters vector ($x, o \in E \times I$).
Definition 2

A method \( m \) is valid or conforms to its specification if for each \((x,o)\), the constraint \( H \) is satisfied:

\[
\forall (x,o) \in E \times I_c : H(x,o) \quad (2)
\]

In other words, for all elements of the input domain: If the precondition \( P \) is satisfied then the postcondition \( Q \) and the invariant \( Inv \) must be satisfied.

In order to check that the method \( m \) does not conform to its specification (\( m \) is invalid:

\[
\exists (x,o) \in E \times I_c : \overline{H(x,o)}
\]

i.e.: \( \exists (x,o) \in E \times I_c : (x,o) \in B_1 \cup B_2 \cup B_3 \).

5.2. Algorithms of Conformity Testing

The main goal of the testing algorithm is the use of the constraint model \( H \): 

\[
P \Rightarrow (Q \wedge \overline{Inv})
\]

and the valid input domain partitioning (A1,B1,B2,B3) for checking if the method under test meets its specification. The algorithm execution of conformity test stops when the constraint \( H \) becomes False (\( H(x,o)=0 \)) or when we reach the threshold of test with \( H \) True (Figure 9).

The Figure 9 shows a fragment of the conformity test algorithm of a mutator \( M \) where \( N \) represents the number of times that the test is executed. It indicates the limit value that must be taken sufficiently large. In this algorithm we generate randomly the input values that satisfy the pre-condition constraint.

```java
1. do{
2.   do{
3.     for ( x_i in parameter(M) )
4.       { x_i = generate { E_i }; } 
5.       x= (x_1,x_2,…,x_n) ;
6.       o = generate_object();
7.   }while(!P(x,o));
8.   invoke"o.M(x)"
9.   if(Q(x,o)&&Inv(o))
10.  A_1.add{x_i,o}; // The set A_1 does not contain the same elements
11.  elseif ( Q(x,o)&&!Inv(o) )
12.  B_1.add{x_i,o};
13.  elseif( !Q(x,o)&&Inv(o) )
14.  B_2.add{x_i,o};
15.  else
16.  B_3.add{x_i,o}
17.  }while(A_1.size()< N && B_1.isEmpty()&& B_2.isEmpty() && B_3.isEmpty());
```

**Figure 9.** Conformity Test Algorithm of a Mutator \( M \)

The condition analysis of the do...while loop has the potential for providing useful information about the validity of the method \( m \), and finding the constraints affected by the problem:

- **Case 1:** \( B_1 \) is not empty: i.e. \( \exists (x,o) \in E \times I_c : (x,o) \in B_1 \) (i.e. \( H=0 \)): \( m \) is not valid and exactly \( m \) does not satisfy the invariant \( Inv \) of its class.

- **Case 2:** \( B_2 \) is not empty: i.e. \( \exists (x,o) \in E \times I_c : (x,o) \in B_2 \) (i.e. \( H=0 \)): \( m \) is not valid and specifically \( m \) does not satisfy its postcondition.
- **Case 3**: \( B_1 \) is not empty: i.e. \( \exists (x, o) \in E \times I : (x, o) \in B_1 \) (i.e. \( H=0 \)): \( m \) is not valid and specifically \( m \) does not satisfy neither the postcondition \( Q \), nor the invariant \( \text{Inv} \) of its class.

- **Case 4**: The size of \( A_1 \) reaches the threshold of test.

We note the elements of \( A_1 \) by the pairs \( ((x_1, o_1), (x_2, o_2), \ldots, (x_N, o_N)) \) such that:

\[
\forall (x, o) \in \{(x_1, o_1), (x_2, o_2), \ldots, (x_N, o_N)\} : H(x, o) = 1
\]

It is important to note that the limit \( N \) must be sufficiently large \( N \rightarrow \infty \) to reduce the risk of not meeting a potential input which does not conform to the specification. Consequently, we may admit that the method is valid with a rejected error margin. The conformity test for an extended constructor and observer uses the same principle as a basic constructor and mutator.

### 5.3. Evaluation

We consider an example for evaluating our approach by testing the conformity of the methods transfer and deposit of the class Account (Figure 10). As is indicated above, the conformity test of mutator requires passing through a test of a basic constructor in order to use valid objects in the testing process.

```
1. class Account
2. {
3.     private int bal; // bal is the account balance
4.     // cb_Account() is a basic Constructor C_b
5.     public static Account cb_Account (int x){
6.         Account o = new Account();o.bal=x;return o ;}
7.     // ce_Account() is an extended Constructor C_e
8.     public static Account ce_Account (Account x){
9.         Account o = new Account (){o.bal=x.bal;return o ;}
10.    public void setBal(int x){this.bal=x;}
11.    // getBal() is an observer O
12.    public int getBal (){return this.bal;}
13.    /* In the mutator transfer(x1,x2), x1 to transfer from
14.       the current account o to the account x2 */
15.    public void transfer (int x1,Account x2)
16.      (this.bal=this.bal - x1;
17.      x2.bal=x2.bal + x1;)
18.    public void withdraw (int x1){this.bal=this.bal - x1;}
19.    public void deposit (int x1){
20.        if(x1% 5 == 0) this.bal=this.bal + x1*75/100;
21.        else
22.            this.bal=this.bal + x1;}
23. }
```

**Figure 10. Java Implementation of the Account class**

- **Conformity testing for the method transfer**:

  The constraints \( H \) of the method transfer (Figure 10) in an algebraic specification is shown in the Figure 11 (\( x=(x_1,x_2) \), \( o(a) \) and \( o(b) \) are respectively the object \( o \) after and before the call of the transfer method):
We generate randomly \( x_i \) values in the interval \([-200,200]\) and \( N=100 \) (Table 1).

**Table 1. Result of a Conformity Test of the Transfer Method**

<table>
<thead>
<tr>
<th>Iteration number</th>
<th>((x_1,x_2))</th>
<th>(O)</th>
<th>(P(x,o))</th>
<th>((x,o)) (\in) (H(x,o))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(21, Account(34))</td>
<td>Account(74)</td>
<td>1</td>
<td>(A_1) 1</td>
</tr>
<tr>
<td>2</td>
<td>(59, Account(98))</td>
<td>Account(164)</td>
<td>1</td>
<td>(A_1) 1</td>
</tr>
<tr>
<td>3</td>
<td>(48, Account(109))</td>
<td>Account(182)</td>
<td>1</td>
<td>(A_1) 1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>98</td>
<td>(36, Account(7))</td>
<td>Account(107)</td>
<td>1</td>
<td>(A_1) 1</td>
</tr>
<tr>
<td>99</td>
<td>(73, Account(30))</td>
<td>Account(199)</td>
<td>1</td>
<td>(A_1) 1</td>
</tr>
<tr>
<td>100</td>
<td>(19, Account(77))</td>
<td>Account(185)</td>
<td>1</td>
<td>(A_1) 1</td>
</tr>
</tbody>
</table>

The test result shows that for 100 iterations the constraint \( H \) is always True (\( H = 1 \)), this leads to the conclusion that the transfer method meets its specification (Table 1).

- **Conformity testing for the method deposit:**

For another example of conformity testing, we consider the method deposit of the class Account (Figure 10). The constraint \( H \) of the deposit method in an algebraic form is given in the Figure 12:

**Figure 11. Constraints of the Method Transfer**

**Figure 12. Constraints of the Method Deposit**

In the same context we test the deposit method by generating randomly \( x_i \) values in the interval \([-200,200]\) and \( N=100 \) (Table 2).
Table 2. Result of a Conformity Test of the Deposit Method

<table>
<thead>
<tr>
<th>Iteration number</th>
<th>( x_i )</th>
<th>( O )</th>
<th>( P(x,o) )</th>
<th>( (x,o) \in H(x,o) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>193</td>
<td>Account(71)</td>
<td>1</td>
<td>A_1</td>
</tr>
<tr>
<td>2</td>
<td>56</td>
<td>Account(112)</td>
<td>1</td>
<td>A_1</td>
</tr>
<tr>
<td>3</td>
<td>188</td>
<td>Account(95)</td>
<td>1</td>
<td>A_1</td>
</tr>
<tr>
<td>4</td>
<td>168</td>
<td>Account(143)</td>
<td>1</td>
<td>A_1</td>
</tr>
<tr>
<td>5</td>
<td>62</td>
<td>Account(79)</td>
<td>1</td>
<td>A_1</td>
</tr>
<tr>
<td>6</td>
<td>149</td>
<td>Account(88)</td>
<td>1</td>
<td>A_1</td>
</tr>
<tr>
<td>7</td>
<td>170</td>
<td>Account(151)</td>
<td>1</td>
<td>B_2</td>
</tr>
</tbody>
</table>

As is shown in Table 2, the deposit method does not meet its specification, the input data \((170, \text{Account}(151))\) in the iteration 7 makes \( H \) false \((H(x,o)=0)\). We deduce that the implementation of this method is not in conformity with its specification. Normally the deposit method must add to the current account the amount \( x_i \) but it adds to the current account only 75% of the amount \( x_i \) when this amount is a multiple of 5 (Figure 10).

6. Robustness Testing

The robustness testing is an important step in the verification process for an OO specification and can detect anomalies in invalid input data (the data do not satisfy the precondition constraint) that induce valid output constraints (the data that satisfy postcondition and invariant). Most of test oracles do not integrate the invalid data in the test process. In this section we present a constraint model and algorithms for testing the robustness of OO programs.

6.1. Formal Model of Robustness Testing

In our approach, the robustness verification of a method \( m( ) \) in an OO paradigm means that the output constraints are satisfied if and only if the input constraints are satisfied for all invocation of the method under test. In this way, the invalid input data must induce only invalid output constraints (Figure 13).

In this work, an invalid data is not an undefined data: an invalid data is a data for which the constraints precondition, postcondition and invariant are well defined (a data that induces a division by 0 is an undefined data and is not accepted…).

Consider a method \( m \) of class \( C \) such that: \( o \) the receiver object and \( x \) the vector of parameters: \( (x,o) \in E \times I_c \).

Definition 3 (Robust method)

A method \( m \) is robust according to its specification if it satisfies the following conditions:

- It conforms to its specification.
- For each invalid input data \((x,o)\) does not satisfy the precondition: \( P(x,o) \), the postcondition \( Q \) and the invariant \( Inv \) should not be both valid in output \( (Q(x,o) \lor Inv(o)) \).
On the theoretical level, we are looking for strengthening the current constraint $H$ in order to integrate this type of test. As is shown in Table 3, the constraint $H$ defined above for conformity test takes in the robustness testing the following form:

$$H_{robustness}((x, o) : [P(x, o) \iff (Q(x, o) \land \text{Inv}(o))]), (x, o) \in E \times I_c$$

<table>
<thead>
<tr>
<th>$P$</th>
<th>$Q$</th>
<th>$\text{Inv}$</th>
<th>$H : P \Rightarrow (Q \land \text{Inv})$</th>
<th>$H_{robustness} : P \Leftrightarrow (Q \land \text{Inv})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B_1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B_2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>B_3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A_{2,1}</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>A_{2,2}</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>A_{2,3}</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>A_{2,4}</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 13 . Principle of Robustness Testing

In this test, we assume that with an invalid input, we can expect only invalid output constraints: any valid result coming from an invalid input indicates the presence of a robustness problem into the method implementation. As a deduced result, we will particularly be focusing on invalid input elements of a method conforms to its specification, i.e. the elements of the family ($A_{2,1}, A_{2,2}, A_{2,3}, A_{2,4}$) (Table 3).

The testing approach consists of two steps of testing:

- Main testing or conformity testing.
- Complementary testing or robustness testing concerns only the methods where the conformity testing is validated.

Theorem

The method $m$ is robust according to its specification if:

$$\forall (x, o) \in E \times I_c : H_{robustness}(x, o)$$

Corollary

A method $m$ that conforms to its specification, is not robust relatively to this specification $\exists (x, o) \in E \times I_c : H_{robustness}((x, o))$ only if: $\exists (x, o) \in E \times I_c : (x, o) \in A_{2,4}$. 
This means that a method $m$ that conforms to its specification, is not robust relatively to this specification if there exist an input data $(x, o)$ which does not satisfy the pre-condition $P$, but it satisfies the postcondition $Q$ and the invariant $Inv$ at the output of the method under test.

### 6.2 Algorithms of Robustness Testing

The main goal of robustness algorithms is the use of the constraint model $H_{robustness}$: $P \iff (Q \land Inv)$ and the invalid input domain partitioning ($A_{2.1}$, $A_{2.2}$, $A_{2.3}$, $A_{2.4}$) for checking if the method under test is robust relatively to its specification. The algorithm execution of robustness test stops when the constraint $H_{robustness}$ becomes False ($H_{robustness}(x,o)=0$) or when we reach the threshold of test with $H_{robustness}$ True (Figure 14). We assume that the method under test is in conformity with its specification.

$N$ represents the number of times that the test is executed ($N$ is an input constant of the algorithm). In this algorithm we generate randomly input values that do not satisfy the pre-condition constraint.

```
1. do{
2.   do{
3.     for ( x_i in parameter(M))
4.       {x_i = generate ( E_i) ;}
5.     x= (x_1,x_2,...,x_n);
6.   o = generate_object ( );
7.   }while(P(x,o));
8.   invoke"o.M(x)"
9.   if( !(Q(x,o))&& !Inv(o))
10.      A_{2.1}.add(x,o);
11.     elseif( !(Q(x,o))&&Inv(o))
12.        A_{2.2}.add(x,o);
13.     elseif( Q(x,o))&& !Inv(o))
14.        A_{2.3}.add(x,o);
15.     else
16.        A_{2.4}.add(x,o);
17.     }while(A_{2.1}.size()<N && A_{2.2}.size()<N &&
18.        A_{2.3}.size()<N && A_{2.4}.isNotEmpty());
```

**Figure 14 . Robustness Test Algorithm of a mutator M**

The analysis of the output condition of robustness algorithm for a valid method will be used to conclude if this method is robust and otherwise providing useful information:

- **Case 1:** The size of $A_{2.1}$ reaches the threshold $N$ of test. This means that for a sufficiently large number $N$ of input values $(x,o)$ which do not satisfy the pre-condition $P$, the invariant $Inv$ and the postcondition $Q$ are not satisfied: $m$ is robust relatively to its specification $(\forall(x,o):H_{robustness}=1)$.

- **Case 2:** The size of $A_{2.2}$ reaches the threshold $N$ of test. This means that for a sufficiently large number $N$ of input values $(x,o)$ which do not satisfy the pre-condition $P$, the postcondition $Q$ is not satisfied i.e. $(\forall(x,o):H_{robustness}(x,o)=1)$. The method $m$ is robust relatively to its specification.

- **Case 3:** The size of $A_{2.3}$ reaches the threshold $N$ of test. This means that for a sufficiently large number $N$ of input values $(x,o)$ which do not satisfy the pre-condition $P$, the invariant $Inv$ is not satisfied i.e. $(\forall(x,o):H_{robustness}(x,o)=1)$. The method $m$ is robust relatively to its specification.

- **Case 4:** $A_{2.4}$ is not empty $(\exists(x,o) \in E \times I, (x,o) \in A_{2.4})$. This means that there exist an input value $(x,o)$ which does not satisfy the pre-condition $P$ and it satisfies at the output both the postcondition $Q$ and the invariant $Inv$ i.e.
\((\exists(x,o) \in E \times I, H_{\text{robustness}}(x,o)=0)\) (Table 3). As a result, the method \(m\) is not robust according to its specification.

### 6.3. Evaluation

For evaluating our robustness approach, we consider the transfer method of the Account class (Figure 10). The conformity test presented in the last section shows that this method is conforming to its specification. An example of a robustness test of the transfer method where \(N=100\) and \(x_1\) into the interval \([-200,200]\) gives the following results :

<table>
<thead>
<tr>
<th>Iteration number</th>
<th>((x_1, x_2))</th>
<th>(O)</th>
<th>(P(x,o))</th>
<th>(H_{\text{robustness}}(x,o))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(136, Account(35))</td>
<td>Account(32)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>(100, Account(67))</td>
<td>Account(78)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>(46, Account(120))</td>
<td>Account(41)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>(87, Account(74))</td>
<td>Account(140)</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

For the first three iterations, we have: \(x_1 > \text{balance}(o) > \text{balance}(o)/2\) (i.e. \(P(x,o)=0\)) and it induces to a false invariant \((\text{balance}(o)<0)\) at the output (i.e. \(H_{\text{robustness}}(x,o)=1\)). In the iteration 4, we have \((x,o)=((87, \text{Account(74)}), \text{Account(140)})\):

- balance \((o)/ 2 < x_1 < \text{balance}(o)\) (i.e., \(P(x,o)=0\)), and this induces \(Q(x,o)=1\) and \(\text{Inv}(o)=1\) (i.e., \(H_{\text{robustness}}(x,o)=0\)).

Indeed, our implementation cannot reject this situation and consequently the method transfer under test which is conforming to its specification, is considered not robust relatively to the same specification.

### 7. Conclusions and Future Work

This paper introduces an approach to generate test data from formal specifications in OO software testing. The key idea of our approach is the definition of a complementary test based on invalid input data that do not satisfy the precondition constraint of the method under test. The robustness testing proposed can integrate the invalid data in the testing process for strengthening the conformity testing and detecting design anomalies of OO classes.

The first result of our work is a constraint model for testing the conformity of OO methods. The second result of our approach is the use of an equivalence partitioning technique that can be used to reduce the number of test cases by uncovering various classes of errors. The third and main result of this work is a negative testing that completes the testing process by introducing the invalid data to measure the robustness of OO programs.

The limitation of this approach is that it is applied only to OO classes without considering the connections between these classes. Indeed, the equivalence partitioning of complex system where there are classes that are linked by some inheritance, association and composition relations is not treated. This the reason why our future works are now oriented in the first instance to generalize our formal model for testing the robustness of overriding methods in sub-classes from the test result of overridden methods in super-classes and in the second instance to develop an overall formal framework for testing the robustness of an object oriented model integrating a rigorous specification language as JML tools and formally designed by UML modeling language.
References


