A Robustness Testing Approach for an Object Oriented Model

Khadija Louzaoui*, Khalid Benlhachmi
Department of Computer Science, Faculty of Sciences, Ibn Tofail University, Kenitra, Morocco.

Corresponding author. Email: Khadija.louzaoui@gmail.com
Manuscript submitted February 1, 2016; accepted April 18, 2016.
doi: 10.17706/jcp.12.4.335-353

Abstract: In this paper we present a new test model of object oriented (OO) programs for testing conformity and robustness from formal specifications. The main contribution of this work is a robustness approach based on invalid input data that do not satisfy the precondition constraint of the program under test. This additional test is used to strengthen the conformity test of programs and to enrich the concept of test by detecting other contract anomalies between user and programs. The approach of this work shows that the test cases developed for testing an original method can be used for testing its overriding method in derived classes by inheritance operation and then the number of test cases can be reduced considerably. In this context we can use a single generator of test data to verify both conformity and robustness, thus making it possible to increase the level of automation during the whole testing process.

Key words: Specification, conformity testing, robustness testing, valid data, invalid data, test data generation, inheritance, constraint resolution.

1. Introduction

Formal specifications are part of a more general collection of techniques that are known as 'formal methods' and can be used to discover problems in system requirements. These are all based on mathematical representation and analysis of software. Their principal benefits are in reducing the number of errors in systems so their main area of applicability is critical systems: air traffic control information systems, railway signaling systems, spacecraft systems, and medical control systems. In this area, the use of formal methods is most likely to be cost-effective. Formal specification involves investing more effort in the early phases of software development. This reduces requirements errors as it forces a detailed analysis of requirements: incompleteness and inconsistencies can be early discovered and resolved. Hence, savings as made as the amount of rework due to requirements problems is reduced (Fig. 1).

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 (Fig. 2). Precondition (P), postcondition (Q) and invariant (Inv) are the main used constraints in an OO specification (Fig. 3), and can be written by using specific languages as Object Constraint Language (OCL) [1] and Java Modeling Language (JML) [2].

Design by constraints is introduced by Meyer [3], [4] and defines the contract between users and programs (Fig. 4). In this context the conformity of a program in an OO paradigm means that output constraints are satisfied if input constraints are satisfied for all invocation of the program under test (Fig. 5). In an OO testing, the test data generation is based on the valid input data that satisfy precondition constraints, and is used for 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.

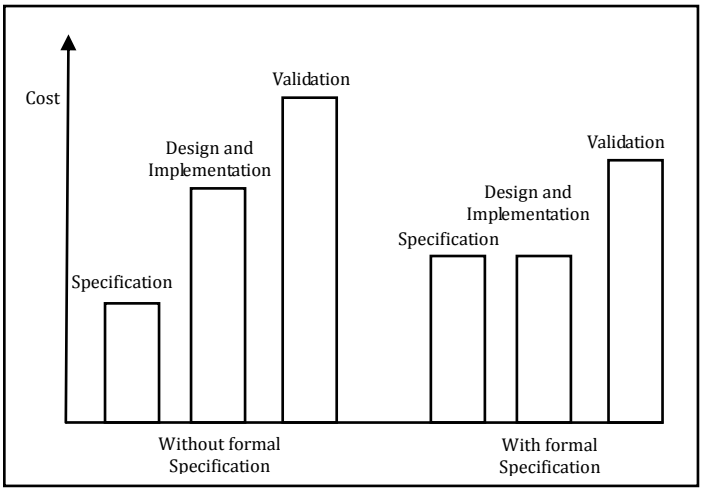

Fig. 1. Development costs with formal specification.

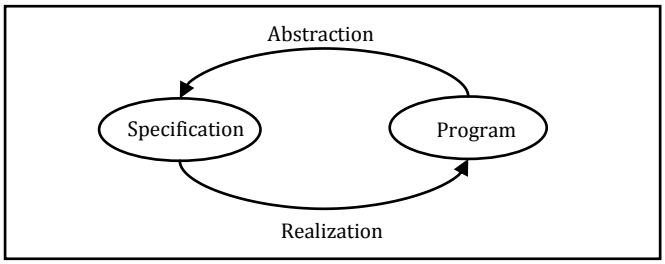

Fig. 2. Relationship between program and its specification.

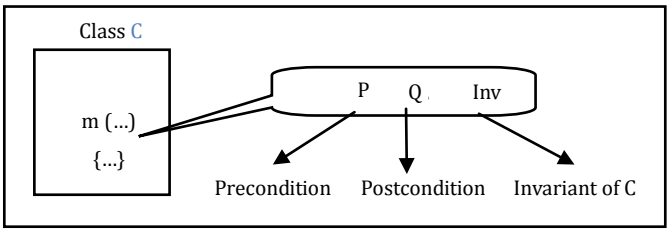

Fig. 3. Constraints of the method *m( ).*

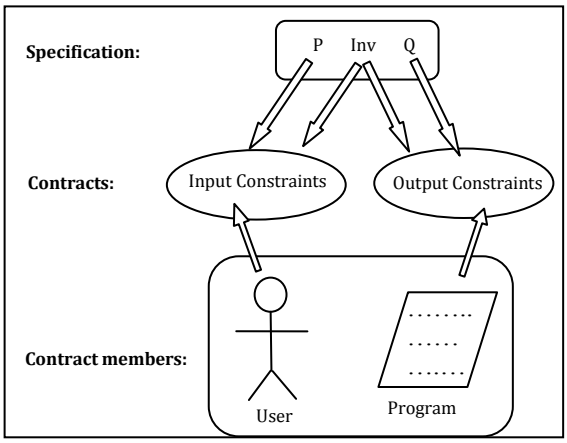

Fig. 4. User and program contracts.

The main contribution of this work is a robustness approach based on invalid input data that do not satisfy

the precondition constraint of the program under test. This additional test is used to strengthen the conformity test of programs and to enrich the concept of test by detecting other contract anomalies between user and programs. The idea is to integrate the invalid data in the testing process for strengthening the contract between users and programs in an OO paradigm. In this context, the invalid input data for an OO program are selected so that all input output constraints (logical predicates of the program (Fig. 4)) are well defined. These input data must induce a normal execution of the program under test (the input data that induce an immediate interruption of the execution or those taken into account by the exception mechanism are ignored by the generator of test data).

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 constraint (Fig. 5). In our approach we complete the testing process by introducing a test based on the invalid data to measure the robustness of OO programs (Fig. 6). In this context, we specify 4 behaviors of a program if the user does not respect his contract:

- Abnormal program termination: immediate interruption of the program execution (Behavior1).
- Abnormal program termination: redirection to the exception handling block (Behavior2)
- Normal program termination: output constraints are satisfied (Behavior3)
- Normal program termination: output constraints are not satisfied (Behavior4)

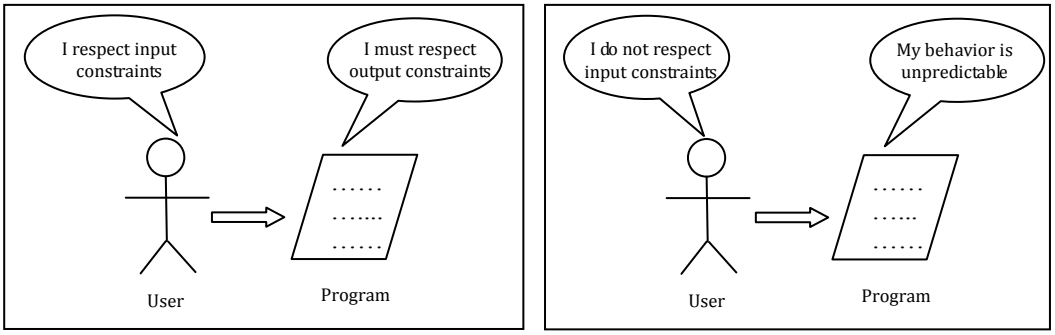

Fig. 5. Conformity contract between user and program.

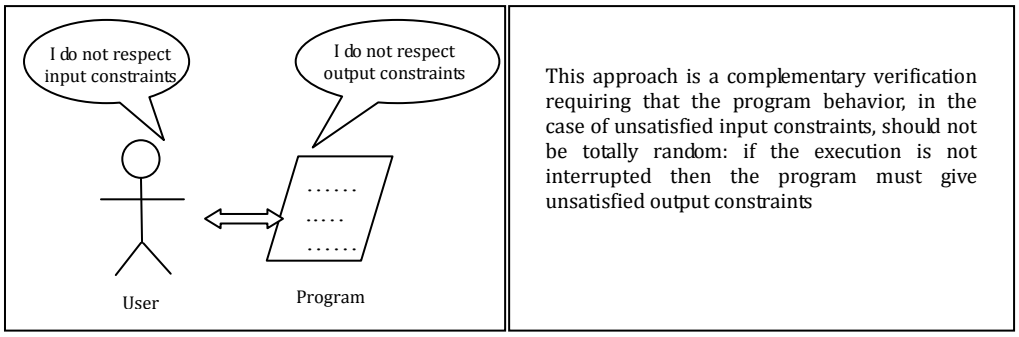

Fig. 6. Robustness contract between user and program.

The key idea of this robustness approach is to add -as far as possible- an obligation for the program to give a single behavior (behavior 4) if the input contract is not satisfied (conformity contract is already satisfied). Indeed the program developer must take into account two contracts instead of one contract. This will increase slightly the developer's responsibility and reduce the user's responsibility. In this approach, a robust software is characterized by the following properties:

- Less requirements for the user.
- No external unit (configurable software package) to verify the responsibility of the user to each use.

This robustness approach can help to:

- Reducing the overall cost of realization.
- Reducing the execution time.
- Facilitating the use by freeing the user from liability.
- Ensuring value for money: a single generator of test data to verify both conformity and robustness.

This work presents in the first instance the model of conformity testing of OO classes described in [5], [7] and how the conformity of overriding methods in inheritance operation can be deduced from the testing result of their overridden methods in basic classes (Sections 3, 4, 5). Secondly we define the robustness approach and model of robustness testing based on invalid input data (Section 6) and we show how this robustness approach can be extended for derived classes by inheritance (Section 7). Finally we describe our approach with an example of conformity and robustness testing for an OO model (Section 8).

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], 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 [6], we have generalized the basic model of similar behaviors using constraints propagation, equivalence partitioning and formal proof.

In [7], 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 [8], the authors propose a generation of test data from formal specifications using the Object Constraint Language (OCL). In [9] 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 [10], the authors propose a randomly generation of test data from a JML specification of class objects. They classify 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] 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 for Basic Classes

3.1. 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 (Fig. 3) into a single logical formula. This model should translate algebraically the contract between users and programs (Fig. 4): 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 (Fig. 5). Concerning the invariant, it must be satisfied before and after the method call.

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 (Fig. 7).

Definition 1

We define the constraint H of a method $m(x_1, x_2,..., x_n)$ of a class C as a property of the pair (x, o) ($x=(x_1, x_2, ..., x_n)$ is the vector of input parameters and o is the receiver object) such that:

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

where:

- $E=(E_1\times E_2\times\ldots\times E_n)$ is the domain of input vectors of the method m.
- 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 $m()$.

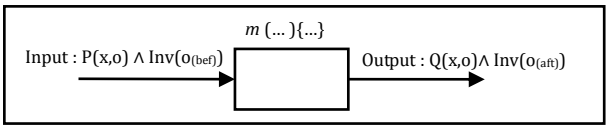

Fig. 7. Input-Output constraints of a method $m()$.

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 postcondition Q and the invariant after the call (Fig.7). 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:

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

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 $Inv(o)$(Fig. 8)

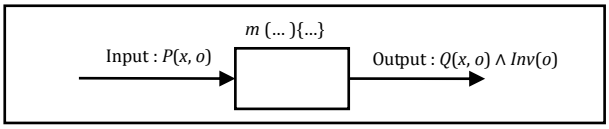

Fig. 8. Simplified specification of a method *m()*.

3.2. 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 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).

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 (Fig. 9):

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

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

- $A_1 = \{(x,o) \in E \times I_C / (P(x,o), Q(x,o), Inv(o)) = (1,1,1)\}$: the domain whose elements (x, o) satisfy P,Q, and Inv .
- $A_2 = \{(x,o) \in E \times I_C / (P(x,o), Q(x,o), Inv(o)) = (0,?,?)\}$, this domain represents the pairs (x, o) of $E \times I_c$ such as the precondition P of the method is not satisfied. We divide A_2 (for a false precondition) to 4 domains $A_{2.1}$, $A_{2.2}$, $A_{2.3}$ and $A_{2.4}$ (Fig. 9).

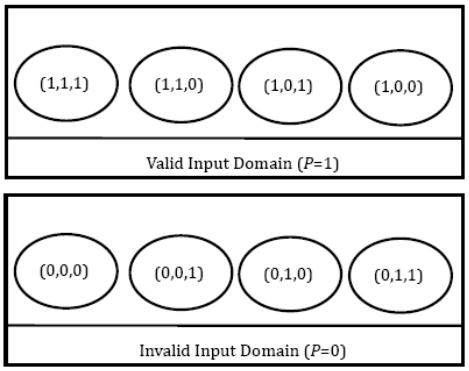

Fig. 9. Input domain partitioning.

B can be divided into 3 subsets B_1, B_2 and B_3 (Fig. 9):

- $B_1 = \{(x,o) \in E \times I_c / (P(x,o), \mathrm{Q(x,o)}, \mathrm{Inv(o)}) = (1,1,0)\}$
- $B_2 = \{(x,o) \in E \times I_c / (P(x,o), \mathrm{Q(x,o)}, \mathrm{Inv(o)}) = (1,0,1)\}$
- $B_3 = \{(x,o) \in E \times I_c / (P(x,o), \mathrm{Q(x,o)}, \mathrm{Inv(o)}) = (1,0,0)\}$

(A_1 , B_1 , B_2 , B_3) is the partition whose elements satisfy the precondition constraint (*i.e.* the partition of the valid domain) and ($A_{2.1}$, $A_{2.2}$, $A_{2.3}$, $A_{2.4}$) is the partition whose elements do not satisfy the precondition constraint (*i.e.* the partition of the invalid domain) (Fig. 9).

4. Formal Model of Constraint in Inheritance

The work of [6], [7] is a way for defining the constraint model of an overriding method in a derived class during inheritance operation by using constraint model of basic classes and constraint propagation. Indeed, we establish a series of theoretical concepts in order to create a solid basis for testing the conformity of overriding methods in subclasses using the test result of original methods in superclasses.

4.1. Constraint Propagation in Inheritance

The results of this paragraph are based on the works of Liskov, Wing [21] and Meyer [22] who have studied the problem relating to types and subtypes with behavioral specification in an object oriented (OO) paradigm. Indeed, a derived class C_2 from a basic class C_1 (Fig. 10) obeys the Liskov Substitution Principle (LSP) if for each system of methods (overridden method: $m^{(1)}$, overriding method: $m^{(2)}$):

- The precondition $P^{(2)}$ of $m^{(2)}$ must be equal to or weaker than the precondition $P^{(1)}$ of $m^{(1)}$: $P^{(1)} \Rightarrow P^{(2)}$.
- The postcondition $Q^{(2)}$ of $m^{(2)}$ must be equal to or stronger than the postcondition $Q^{(1)}$of $m^{(1)} : Q^{(2)} \Rightarrow Q^{(1)}$.
- The class invariant $Inv^{(2)}$ of the subclass C_2 must be equal to or stronger than the class invariant $Inv^{(1)}$ of the C_1 :

$$Inv^{(2)} \Rightarrow Inv^{(1)}$$

As a result of LSP, we have (Fig. 10):

$$P^{(2)} \Leftrightarrow (P^{(1)} \vee P_2^{'}) \quad (1)$$

$$Q^{(2)} \Leftrightarrow (Q^{(1)} \wedge Q_2^{'}) \quad (2)$$

$$Inv^{(2)} \Leftrightarrow (Inv^{(1)} \wedge Inv_2^{'}) \quad (3)$$

where $P_2^{'}, Q_2^{'}$ denote respectively the specific precondition, postcondition of the overriding method $m^{(2)}$, and $Inv_2^{'}$ the specific invariant of the class C_2

Based on the definition of the generalized constraint (Definition 1) (Fig. 8), we have:

$$H^{(1)}(x,o) : P^{(1)}(x,o) \Rightarrow [Q^{(1)}(x,o) \wedge Inv^{(1)}(o)], (x,o) \in E \times I_{C1} : \text{The constraint of } m^{(1)}.$$

$$H^{(2)}(x,o) : P^{(2)}(x,o) \Rightarrow [Q^{(2)}(x,o) \wedge Inv^{(2)}(o)], (x,o) \in E \times I_{C2} : \text{The constraint of } m^{(2)}.$$

Using (1), (2), (3) the constraint of $m^{(2)}$ will have the following form:

$$H^{(2)} : [P^{(1)} \vee P_2^{'}] \Rightarrow [Q^{(1)} \wedge Inv^{(1)} \wedge Q_2^{'} \wedge Inv_2^{'}]$$

In our approach, the specification ($P^{(2)}$,$Q^{(2)}$, $Inv^{(2)}$) of $m^{(2)}$ is constituted by :

Two inputs: the Basic Input (BI=$P^{(1)}$) and the Specific Input (SI= $P_2^{'}$) of $m^{(2)}$.

Two outputs: Basic Output (BO=($Q^{(1)}$, $Inv^{(1)}$)) and the Specific Output (SO= ($Q_2^{'}$, $Inv_2^{'}$)) of $m^{(2)}$. This induces 4 possible combinations of I-O: *(BI, BO)* (Fig. 11), *(BI, SO), (SI, BO), (SI, SO)* (Fig. 12).

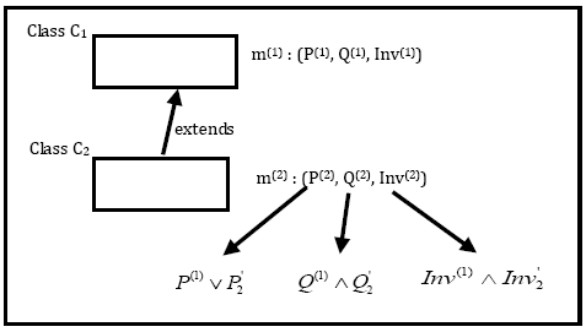

Fig. 10. Constraints of $m^{(1)}$ and $m^{(2)}$.

4.2. Constraint Model of Overriding Methods

The aim of this paragraph is to construct a formal model of an overriding method $m^{(2)}$ by generalizing the constraint model of the original method $m^{(1)}$. In achieving this goal, we should define a specific constraint of inheritance and also study the compatibility between overriding methods and original methods in the parent class.

In this sense, we had proposed in [5], [6] the definition of similarity concept for assuring if the overriding method $m^{(2)}$ has the same behavior as its original version $m^{(1)}$ in the super-class relatively to their common specification ($P^{(1)}$, $Q^{(1)}$, $Inv^{(1)}$). Indeed, the implementation in the subclass must override the implementation in the super class by providing a method $m^{(2)}$ that is similar to $m^{(1)}$.

We propose firstly the definition of a constraint $H_{(SI, SO)}$ allowing a partial view of the overriding method $m^{(2)}$ in the class C_2. Secondly, we determine the relationship between constraints $H^{(2)}$, $H^{(1)}$ and $H_{(SI, SO)}$.

The constraint $H_{(SI, SO)}$ specifies the logical relationship between the input specific predicate $P_2^{'}$ of $m^{(2)}$ and output predicates ($Q_2^{'}$, $Inv_2^{'}$) specific to $m^{(2)}$:

Definition 2: Constraint $H_{(SI,SO)}$

We define the constraint $H_{(SI,SO)}$ of an overriding method $m^{(2)}$ of a sub-class C_2 as a logical property of the pair $(x, o) \in E \times I_{C2}$ such that:

$$H_{(SI,SO)}(x,o) : P_2^{'}(x,o) \Rightarrow [Q_2^{'}(x,o) \wedge Inv_2^{'}(o)]$$

where I_{C2} is the set of instances of C_2, and E is the set of parameters vector of $m^{(2)}$.

The logical implication in the proposed formula means that: each call of method with (x, o) satisfying the specific precondition $P_2^{'}$ before the call, (x, o) must necessarily satisfy the specific postcondition $Q_2^{'}$ and the specific invariant $Inv_2^{'}$ after the call.

The relationship between $H^{(1)}$ and $H^{(2)}$ for two similar methods [5], [6] $m^{(1)}$ and $m^{(2)}$ in classes C_1 and C_2 is shown in the following result:

$$[H^{(2)} \Leftrightarrow (H^{(1)} \wedge H_{(SI, SO)})] \qquad (R)$$

Indeed the overriding method $m^{(2)}$ is defined by 2 constraints *BI-BO* (Fig. 11) and *SI-SO* (Fig. 12).

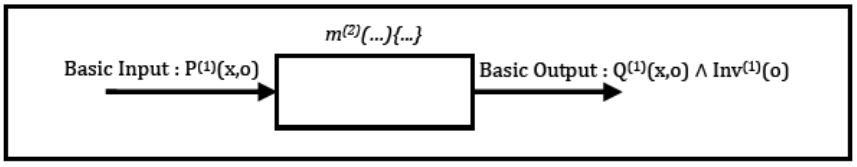

Fig. 11. Constraint *BI-BO* of an overriding method.

The constraints $H^{(1)}$, $H_{(SI,SO)}$, and the relation *(R)* form the theoretical basis that will be used to test the conformity of overriding methods in subclasses from the test result of their original methods in parent

classes.

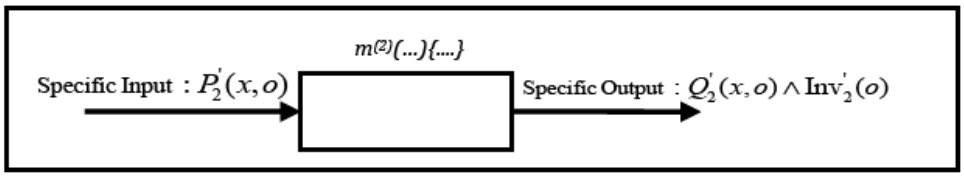

Fig. 12. Constraint *SI-SO* of an overriding method.

5. Conformity Testing

In [6], [7] we have used 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 logical equations and to implement algorithms for generating test data. In conformity test (Fig. 13), 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 (Fig. 4), by giving valid input values. In this context, the proposed test oracle rejects the method call with an invalid input value.

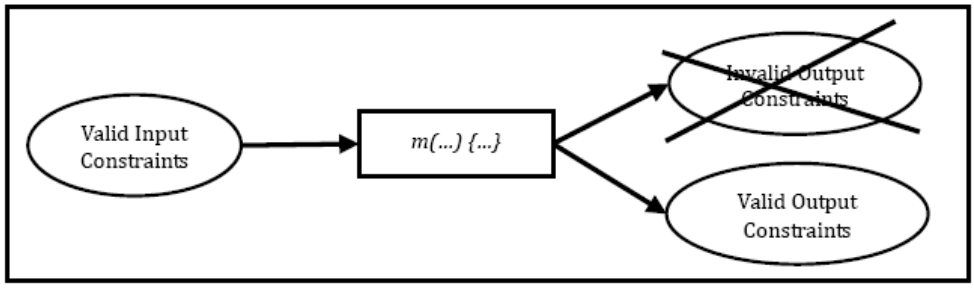

Fig. 13. Principle of conformity testing.

5.1. Constraint of Conformity Testing

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 (Fig. 5), (Fig. 13).

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

Definition 3

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

$$\forall (x,o) \in E \times I_c : H(x,o)$$

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: $\exists (x,o) \in E \times I_c : \overline{H(x,o)}$), it is sufficient to find an input value that satisfies the precondition but does not satisfy output constraints:

$$\exists (x,o) \in E \times I_C : P(x,o) \wedge \left(\overline{Q(x,o)} \vee \overline{Inv(o)} \right)$$

i.e.: $\exists (x,o) \in E \times I_c : (x,o) \in B_1 \cup B_2 \cup B_3$ (Fig. 9).

5.2. Algorithm of Conformity Testing

The main goal of the testing algorithm is the use of the constraint model $H: P \Rightarrow (Q \wedge Inv)$ and the valid input domain partitioning (A_1, B_1, B_2, B_3) (Fig. 9) for checking if the method under test meets its specification. The execution of conformity test algorithm stops when the constraint H becomes False ($H(x, o)=0$) or when we reach the threshold N of test with H True (N represents the number of times that the test is executed) (Fig. 14).

```
1.  do{
2.    do{
3.    for ( xi in parameter(M))
4.     {xi = generate ( Ei) ; }
5.      x= (x1,x2,...,xn) ;
6.      o = generate_object();
7.      }while(!P(x,o));
8.    invoke"o.M(x)"
9.    if(Q(x,o)&&Inv(o))
10.  A1.add(x,o);// The set A1 does not contain the same elements
11. elseif( Q(x,o)&&!Inv(o))
12.  B1.add(x,o);
13. elseif(!Q(x,o)&&Inv(o))
14.  B2.add(x,o);
15. else
16.  B3.add(x,o)
17. }while(A1.size()< N && B1.isEmpty()&& B2.isEmpty() && B3.isEmpty());
```

Fig. 14. Conformity test algorithm of a mutator *M.*

5.3. Conformity Testing of Overriding Methods

The conformity test of $m^{(2)}$ requires the following steps:

- *Step 1:* a conformity test of a basic constructor of the class C_2 (Fig. 10).This step is necessary for using valid objects at the input of the method under test.
- *Step 2:* a similarity test of $m^{(1)}$ and $m^{(2)}$ relatively to ($P^{(1)}$, $Q^{(1)}$, $Inv^{(1)}$).

We assume that the test of step 2 showed that $m^{(1)}$ and $m^{(2)}$ are similar [5], [6].

In [7] we have shown how the conformity test process of the overriding method $m^{(2)}$ relatively to $H^{(2)}$ can be based on the test result of $m^{(1)}$ relatively to $H^{(1)}$(Fig. 15).

```
If (m(1) is in conformity relatively to (P(1),Q(1),Inv(1)))
  If (m(2) is in conformity relatively to (P2',Q2',Inv2'))
       - m(2) is in conformity relatively to (P(2),Q(2),Inv(2))
  Else
       - m(2) is not in conformity relatively to
(P(2),Q(2),Inv(2))
  End If
Else
       - m(2) is not in conformity relatively to
```

Fig. 15. Conformity test cases for an overriding method.

The purpose of next sections is in the first instance to generalize similarity and conformity models [5]-[7] in order to test the robustness of methods in basic classes. Secondly we show how the robustness testing of an overriding method $m^{(2)}$ in a derived class can be deduced from robustness results of its overridden method $m^{(1)}$ in the superclass.

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. Constraint Model of Robustness Testing

In our approach, the robustness verification of a method *m( )* in an OO paradigm means that output constraints are satisfied **if and only if** input constraints are satisfied for all invocation of the method under test (Fig. 5, Fig. 6). In this way, the invalid input data must induce only invalid output constraints (Fig. 16).

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 (for example 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 4: 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: $\overline{P(x,o)}$, the postcondition *Q* and the invariant *Inv* should not be both valid in output $\left(\overline{Q(x,o)} \vee \overline{Inv(o)}\right)$.

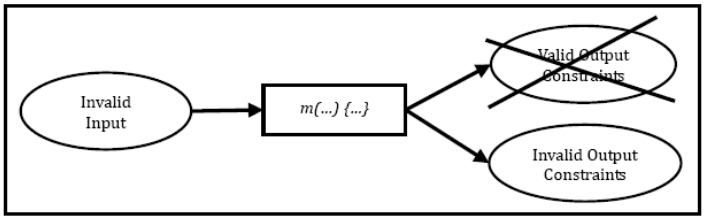

Fig. 16. Principle of robustness testing.

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 1, the constraint *H* defined above for conformity test takes in the robustness testing the following form:

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

Table 1. Truth cases of H and $H_{robustness}$

	P	*Q*	Inv	$H: P \wedge (Q \wedge$ Inv)	$H_{robustness}: P \Leftrightarrow (Q \wedge$ Inv)
A_1	1	1	1	1	1
B_1	1	1	0	0	0
B_2	1	0	1	0	0
B_3	1	0	0	0	0
$A_{2.1}$	0	0	0	1	1
$A_{2.2}$	0	0	1	1	1
$A_{2.3}$	0	1	0	1	1
$A_{2.4}$	0	1	1	1	0

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 1).

The testing approach consists of two steps of testing:

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

Theorem 1

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 $(\exists(x,o) \in E \times I_C : H_{robustness}(x,o) = 0)$ only if : $\exists(x,o) \in E \times I_C : (x,o) \in A_{2.4}$ (Table 1).

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 precondition *P*, but it satisfies the postcondition *Q* and the invariant *Inv* at the output of the method under test.

6.2. Algorithm of Robustness Testing

The main goal of robustness algorithms is the use of the constraint model $H_{robustness}$: $P \Leftrightarrow (Q \wedge Inv)$ 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 execution of robustness test algorithm stops when the constraint $H_{robustness}$ becomes False or when we reach the threshold of test with $H_{robustness}$ True (Fig. 17). We assume that the method under test is in conformity with its specification, and *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 precondition constraint.

```
1.  do{
2.     do{
3.       for ( xi in parameter(M))
4.       {xi = generate ( Ei) ; }
5.         x= (x1,x2,...,xn) ;
6.       o = generate_object ( );
7.     }while(P(x,o));
8.    invoke"o.M(x)"
9.    if( !Q(x,o)&& !Inv(o))
10.     A21.add(x,o);
11. elseif( !Q(x,o)&&Inv(o))
12.     A22.add(x,o);
13. elseif( Q(x,o)&& !Inv(o))
14.     A23.add(x,o);
15. else
16.     A24.add(x,o)
17. }while(A21.size()<N && A22. size()<N && A23.size()<N && A24.isEmpty());
```

Fig. 17. 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 precondition *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 precondition *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 precondition *P*, the invariant *Inv* is not

satisfied *i.e.* $\left(\forall(x,o): H_{robustness}(x,o)=1\right)$. The method *m* is robust relatively to its specification.

- *Case 4*: $A_{2.4}$ is not empty $\left(\exists(x,o)\in E\times I_c,(x,o)\in A_{2.4}\right)$.This means that there exist an input value (*x, o*) which does not satisfy the precondition *P* and it satisfies at the output both the postcondition *Q* and the invariant *Inv i.e.* $\left(\exists(x,o)\in E\times I_c, H_{robustness}(x,o)=0\right)$ (Table 1). As a result, the method *m* is not robust according to its specification.

7. Robustness Testing in Inheritance

In this section we define the robustness approach in a subclass C_2 during inheritance operation and we specify all cases of robustness testing for an overriding method $m^{(2)}$ relatively to its specification $(P^{(2)}, Q^{(2)}, Inv^{(2)})$ from testing result of its overridden method $m^{(1)}$ relatively to its specification $(P^{(1)}, Q^{(1)}, Inv^{(1)})$ in the superclass C_1.

7.1. Robustness Approach of an Overriding Method

In this paragraph we define the robustness approach for overriding methods in a subclass by generalizing the robustness model of overridden methods in superclasses.

Definition 5: Robustness of an overriding method $m^{(2)}$ relatively to $(P^{(1)}, Q^{(1)}, Inv^{(1)})$

An overriding method $m^{(2)}$ is robust relatively to its inherited specification$(P^{(1)},Q^{(1)},Inv^{(1)})$ if:

- $m^{(2)}$ conforms to its inherited specification $(P^{(1)},Q^{(1)}, Inv^{(1)})$.
- For each couple (*x, o*) in the invalid input domain of $m^{(2)}$ $((P^{(1)}\vee P_2^{'})(x, o)=0)$, the postcondition $Q^{(1)}$and the invariant $Inv^{(1)}$ should not be both valid in output of $m^{(2)}$ $(Q^{(1)}(x, o)\wedge Inv^{(1)}(o)=0)$.

Definition 6 : Robustness of an overriding method $m^{(2)}$ relatively to $(P_2^{'}, Q_2^{'}, Inv_2^{'})$

An overriding method $m^{(2)}$ is robust relatively to its own specification $(P_2^{'}, Q_2^{'}, Inv_2^{'})$ if:

- $m^{(2)}$ conforms to its own specification $(P_2^{'}, Q_2^{'}, Inv_2^{'})$.
- For each couple (*x, o*) in the invalid input domain of $m^{(2)}$ $((P^{(1)}\vee P_2^{'})(x, o)=0)$, the postcondition $Q_2^{'}$ and the invariant $Inv_2^{'}$ should not be both valid in output of $m^{(2)}$ $(Q_2^{'}(x, o)\wedge Inv_2^{'}(o)=0)$.

Definition 7 : Robustness of an overriding method $m^{(2)}$ relatively to its specification $(P^{(2)}, Q^{(2)}, Inv^{(2)})$

An overriding method $m^{(2)}$ is robust relatively to its specification$(P^{(2)},Q^{(2)},Inv^{(2)})$ if:

- $m^{(2)}$ conforms relatively to its specification $(P^{(2)},Q^{(2)},Inv^{(2)})$.
- For each invalid couple (*x, o*) of input (does not satisfy the precondition $P^{(2)}(x, o)=0$), the postcondition $Q^{(2)}$ and the invariant $Inv^{(2)}$ should not be both valid in output of $m^{(2)}$ $(Q^{(2)}(x ,o)\wedge Inv^{(2)}(o)=0)$.

Theorem 2

An overriding method $m^{(2)}$ is robust relatively to its specification$(P^{(2)}, Q^{(2)}, Inv^{(2)})$ if and only if:

- $m^{(2)}$ is robust relatively to its inherited specification$(P^{(1)}, Q^{(1)}, Inv^{(1)})$.
- $m^{(2)}$ is robust relatively to its own specification $(P_2^{'}, Q_2^{'}, Inv_2^{'})$.

Theorem 3

An overriding method $m^{(2)}$ (that conforms to its specification $(P^{(2)}, Q^{(2)}, Inv^{(2)})$) is not robust relatively to this specification if: $\exists(x, o)\in E\times I_{C2}$:

$[(P^{(1)}\vee P_2^{'})(x, o)=0$ in input of $m^{(2)}]$ And $[((Q^{(1)}\wedge Inv^{(1)})(x, o)=1)\vee((Q_2^{'}\wedge Inv_2^{'})(x, o)=1)$ in output of $m^{(2)}]$.

7.2. Cases of Robustness Testing for an Overriding Method

We specify in this paragraph all cases of robustness testing for an overriding method $m^{(2)}$ from testing result of its overridden method $m^{(1)}$.

The robustness test of $m^{(2)}$ relatively to its specification ($P^{(2)}$, $Q^{(2)}$, $Inv^{(2)}$) requires the following steps:

- *Step 1:* a conformity test of $m^{(2)}$ relatively to its specification ($P^{(2)}$, $Q^{(2)}$, $Inv^{(2)}$).
- *Step 2:* a robustness test of $m^{(1)}$ relatively to its specification ($P^{(1)}$, $Q^{(1)}$, $Inv^{(1)}$).

The robustness test of $m^{(2)}$ relatively to its specification ($P^{(2)}$, $Q^{(2)}$, $Inv^{(2)}$) is performed only if the conformity test of $m^{(2)}$ relatively to ($P^{(2)}$, $Q^{(2)}$, $Inv^{(2)}$) is validated (step1).We consider that $m^{(1)}$ and $m^{(2)}$ are in conformity relatively to their specifications, and we assume that the robustness test of $m^{(2)}$ starts when the robustness testing of $m^{(1)}$ is completed. Indeed, the robustness test of $m^{(1)}$ induces two cases: $m^{(1)}$ is robust relatively to its specification or $m^{(1)}$ is not robust relatively to its specification:

- Case 1: $m^{(1)}$ is not robust relatively to its specification ($P^{(1)}$, $Q^{(1)}$, $Inv^{(1)}$)

We assume that the method $m^{(1)}$ is not robust relatively to its specification. This means (*Definition 4 and Theorem 1*) that: $\exists (x_0, o_0) \in E \times I_{C1} : H_{robustness}^{(1)}(x_0, o_0)=0$

The object o_0 is an instance of the class C_1. We consider an object o_0' of the subclass C_2 that has the same values as the object o_0 for common attributes of C_1 and C_2. And therefore, the object $o_0^{'}$ has the same behavior as o_0 in a context of C_1 and consequently:

$$\exists\ (x_0, o_0^{'}) \in E \times I_{C2} : H_{robustness}^{(1)}(x_0, o_0^{'})=0$$

i.e. [[$P^{(1)}(x_0, o_0^{'})=0$ in input of $m^{(1)}$] and [$(Q^{(1)} \wedge Inv^{(1)})(x_0, o_0^{'})=1$ in output of $m^{(1)}$]].

$m^{(1)}$ and $m^{(2)}$ are similar [5], [6], then we have:

[[$P^{(1)}(x_0, o_0^{'})=0$ in input of $m^{(2)}$]and [$(Q^{(1)} \wedge Inv^{(1)})(x_0, o_0^{'})$=1 in output of $m^{(2)}$]].

We have two cases: $P_2^{'}(x_0, o_0^{'})=0$ or $P_2^{'}(x_0, o_0^{'})=1$

$$P_2^{'}\ (x_0,\ o_0^{'})=0$$

In this case, we have:

[[$(P^{(1)} \vee P_2^{'})(x_0,\ o_0^{'})$=0 in input of $m^{(2)}$] and [$(Q^{(1)} \wedge Inv^{(1)})(x_0, o_0^{'})$=1 in output of $m^{(2)}$]].

Consequently the method $m^{(2)}$ is not robust relatively to ($P^{(1)}$, $Q^{(1)}$, $Inv^{(1)}$)(*Definition 5*).

Finally, this shows (*Theorem 3*) that the method $m^{(2)}$ is not robust relatively to its specification ($P^{(2)}$,$Q^{(2)}$,$Inv^{(2)}$).

$$P_2^{'}\ (x_0,\ o_0^{'})=1$$

In this case, we have: $(P^{(1)} \vee P_2^{'})(x_0, o_0^{'})$=1

Consequently the couple $(x_0, o_0^{'})$ is in the valid input domain: the robustness of $m^{(2)}$ cannot be deduced from the robustness test of $m^{(1)}$ by using the input value $(x_0, o_0^{'})$ (Fig. 18).

- Case 2 : $m^{(1)}$ is robust relatively to its specification ($P^{(1)}$, $Q^{(1)}$, $Inv^{(1)}$)

In this case, we have to test the method $m^{(2)}$ relatively to its own specification ($P_2^{'}$, $Q_2^{'}$, $Inv_2^{'}$) (Fig. 18).

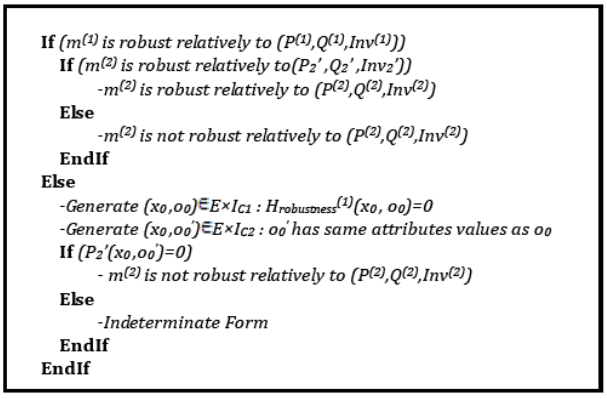

Fig. 18. Robustness test cases for an overriding method.

8. Evaluation

We evaluate the correctness of our approach by implementing the algorithm of conformity and robustness testing for inheritance.

8.1. Specification and Implementation of Methods (*Withdraw*(1), *Withdraw*(2))

We consider for example of conformity and robustness testing methods *withdraw*(1) and *withdraw*(2) of classes *Account1* and *Account2* (Fig. 19).

```
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;}
}
```

Fig. 19. Java implementation of *Account1* and *Account2* classes.

Constraints $H^{(1)}$ and $H^{(2)}$ of *withdraw*(1) and *withdraw*(2) in an algebraic specification are shown in the Fig. 20 ($x=x_1$, $o_{(a)}$ and $o_{(b)}$ are respectively the object o after and before the call of the method):

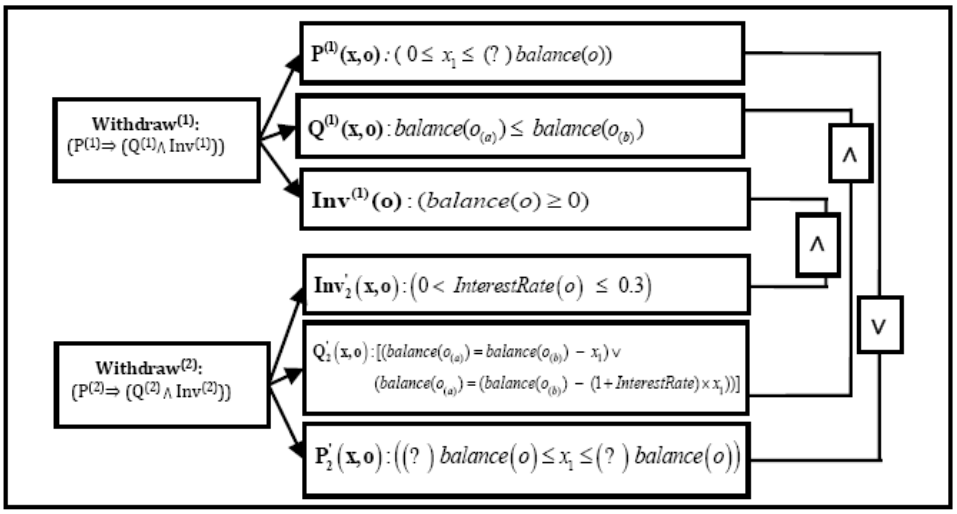

Fig. 20. Constraints of *withdraw*(1) and *withdraw*(2).

8.2. Conformity Testing for Methods (*Withdraw*(1), *Withdraw*(2))

We test firstly the similarity of *withdraw*(2) and *withdraw*(1) on the common valid domain *CVD* (*CVD*={(x, o)∈ E×I_{C2}/($P^{(1)}$(x, o)=1}): for each (x, o) that satisfy the common precondition of *withdraw*(1) and *withdraw*(2) ($P^{(1)}$(x,o)=1), the condition of the block *if{...}*(method *withdraw*(2) in Fig. 19) is not satisfied, and thus the block *if{...}* is not executed, this means that the method *withdraw*(2) does exactly the same thing as

the method *withdraw*(1)(Fig. 19). As a result thereof, *withdraw*(2) and *withdraw*(1) are similar on the valid domain *CVD.*

The second test concerns the conformity testing of *withdraw*(2) that is based on the conformity testing of *withdraw*(1)(Fig. 15):

- Conformity testing for *withdraw* (1)

In order to test the conformity of *withdraw* (1) in the class *Account1*, we generate randomly x_1 and the balance values (for example in the interval *]-200,200[* with *N =100*) (Table 2):

Table 2. Result of a conformity test of the *withdraw*(1) method

Iteration number	x_1	o	$P^{(1)}(x, o)$	$H^{(1)}(x, o)$
1	29	Account1(70)	1	1
2	42	Account1(93)	1	1
3	79	Account1(187)	1	1
…	…	…	…	…
….	….	….	….	….
…..	…..	…..	…..	…..
98	31	Account1(104)	1	1
99	18	Account1(86)	1	1
100	68	Account1(151)	1	1

The test result shows that for 100 iterations the constraint $H^{(1)}$ is always true ($H^{(1)}$=1), we can deduce that the *withdraw*(1) method is valid (Table 2). In this case it is necessary to test the method *withdraw*(2) relatively to its own constraint $H_{(SI, SO)}$ (Fig. 15).

- Conformity testing for withdraw(2) relatively to $H_{(SI, SO)}$

For testing the method *withdraw*(2) relatively to the constraint $H_{(SI, SO)}$ (*Definition 2*), we use an analysis with proof. The testing by proof of the method *withdraw*(2) relatively to the constraint $H_{(SI, SO)}$ is used to strengthen the randomly testing. Indeed, we must have for satisfying the specific output *(SO)*:

The specific postcondition $Q_2^{'}$ must be satisfied.

The specific invariant $Inv_2^{'}$ must be satisfied.

The constraint $Q_2^{'}$ is always satisfied (Fig.19 and Fig.20), however we must proof that $Inv_2^{'}$ is satisfied.

For each created object o_0 ,we have: ($0 \leq InterestRate_0 \leq 0.3$) where $InterestRate_0$ is the initial value assigned to *InterestRate* when creating the object o_0 ,and $InterestRate_{(n)}$ is the value of *InterestRate* after *n* operations of type *withdraw*(2) in an execution sequence.

We have: $[InterestRate_{(n)} = InterestRate_{(n-1)} /2], n \geq 1$ where *n* is number of withdrawals (Fig. 19) .

The geometric series proposed is written in the general case as follows:

$[InterestRate_{(n)} = InterestRate_{(0)}/2^n]$ with $n \geq 0$ and$(0 \leq InterestRate_0 \leq 0.3)$

We deduce that: *[*$\forall n$*:* ($0 \leq InterestRate_{(n)} \leq 0.3$)] And consequently, the specific invariant is always satisfied (Fig. 20). This leads to the conclusion that the method *withdraw*(2) is in conformity to $H_{(SI,SO)}$, and we can deduce that *withdraw*(2) is in conformity with its specification ($P^{(2)}$, $Q^{(2)}$, $Inv^{(2)}$) (Fig. 15).

8.3. Robustness Testing for Methods (*withdraw*(1), *withdraw*(2))

For robustness testing, we test firstly the similarity of *withdraw* methods on the common invalid domain *CID* ($CID=\{(x, o) \in E \times I_{C2} / (P^{(1)} \vee P_2^{'})(x, o)=0\}$), for this we generate randomly x_1 and balance values (for example in the interval *]-200,200[* with the threshold limit *N =100*) (Table 3).

Table 3. Result of a Similarity Test of the Withdraw Methods on CID

Iteration number	x_1	O	$(P^{(1)} \vee P_2^{'})(x,o)$	(x,o) ∈
1	117	Account2(96,0.24)	0	Sim
2	94	Account2(83,0.18)	0	Sim
3	173	Account2(147,0.01)	0	Sim
...	...	...	...	...
....	....	....	...	....
.....	.....	...	...	.....
98	102	Account2(120,0.17)	0	Sim
99	88	Account2(72,0.1)	0	Sim
100	131	Account2(159,0.22)	0	Sim

The test result shows that for 100 iterations the size of the similarity set *Sim* is exactly the threshold limit of the test. We can conclude that the methods *withdraw*(2) and *withdraw*(1) are similar on the domain *CID* relatively to their common specification ($P^{(1)}$, $Q^{(1)}$, Inv_1) (Table 3).

In the last paragraph, we have showed that *withdraw*(2) and *withdraw*(1) are in conformity with their specifications and are similar on the common invalid domain. For testing the robustness of the overriding method *withdraw*(2), we must in the first instance testing the robustness of the overridden method *withdraw*(1) (Fig. 18):

- *Robustness testing for withdraw*(1) *relatively to* ($P^{(1)}$, $Q^{(1)}$, $Inv^{(1)}$)

We generate in the Table 4 robustness test cases for the overridden method *withdraw*(1) relatively to ($P^{(1)}$, $Q^{(1)}$, $Inv^{(1)}$):

Table 4. Result of a robustness test of *withdraw*(1) relatively to ($P^{(1)}$, $Q^{(1)}$, $Inv^{(1)}$)

Iteration number	x_1	*O*	$P^{(1)}(x, o)$	$H_{robustness}^{(1)}(x, o)$
1	113	Account1(87)	0	1
2	176	Account1(138)	0	1
3	91	Account1(42)	0	1
4	101	Account1(73)	0	1
5	137	Account1(180)	0	0

For the first four iterations, we have:

x_1>balance(*o*)>balance(*o*)/2 *(i.e.* $P^{(1)}(x, o)$=0) (Fig. 20) and it induces to a false invariant (*balance*(*o*)<0) at the output (*i.e.* $H_{robustness}^{(1)}(x, o)$=1). In the iteration 5, we have (*for* (*x, o*)=(137, Account1*(180)*) :

balance(*o*)/2<x_1<balance(*o*) (*i.e.* $P^{(1)}(x, o)$=0) (Fig. 20), and this induces $Q^{(1)}(x, o)$=1 and $Inv^{(1)}(o)$=1 (*i.e.* $H_{robustness}^{(1)}(x, o)$=0).

Indeed, our implementation cannot reject this situation and consequently the overridden method *withdraw*(1) under test which is conforming to its specification, is considered not robust relatively to the same specification.

- *Robustness testing for withdraw*(2) *relatively to* ($P^{(2)}$, $Q^{(2)}$, $Inv^{(2)}$)

According to the Fig. 18,we have for *(x,o)=(137, Account1(180))* the method *withdraw*(1) is not robust, we consider for example the object $o^{'}$ of the class *Account2* that has the same balance value ($o^{'}$ *=Account2(180,0.19))* and we must determinate the truth value of $P_2^{'}(x,o^{'})$ (Fig. 18).

We have: $P_2^{'}(x,o^{'})=0$ (because: *(½).180≤137≤ (¾).180* is false (Fig. 20)).

Finally, we can deduce that the overriding method *withdraw*(2) is not robust relatively to its specification

($P^{(2)}$, $Q^{(2)}$, $Inv^{(2)}$)(Fig. 18).

9. Conclusion

This paper presents an approach to generate test data from formal specifications in OO software testing. The key idea of our approach is the definition of an additional 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, reducing the number of test cases by uncovering various classes of errors, and detecting design anomalies of OO classes.

The first result of our approach is a constraint model based on an equivalence partitioning technique for testing the conformity of OO programs. The principal 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. Finally we show how it is possible to exploit the robustness test result of overridden methods for testing overriding methods in derived classes during inheritance operation.

References

[1] Aichernig, B. K., & Salas, P. A. P. (September 19-20, 2005). Test case generation by OCL mutation and constraint solving. *Proceedings of the International Conference on Quality Software* (pp. 64–71). Melbourne, Australia.

[2] Bouquet, F., Dadeau, F., Legeard, B., & Utting, M. (July 2005). Symbolic animation of JML specifications. *Proceedings of International Conference on Formal Methods*: *Vol. 3582* (pp. 75–90). Springer-Verlag.

[3] Bertrand, M. (1992). Applying' design by contract. *Computer*, *25(10)*, 40-51.

[4] Bertrand, M. (2000). *Design by Contract Object and Component Technology Series*, Prentice Hall PTR.

[5] Khalid, B., & Benattou, M. (October 24-26, 2012). A formal model of similarity testing for inheritance in object-oriented software. *Proceedings of the IEEE International Conference* (pp. 38-42). Fez, Morocco.

[6] Khalid, B., & Benattou, M. (2014). Similar behaviors and conformity testing in inheritance for an object oriented model. *IADIS International Journal on Computer Science and Information Systems*, *9(1),* 30-42.

[7] Khalid, B., & Benattou, M. (2013). A formal model of conformity testing of inheritance for object oriented constraint programming. *International Journal of Software Engineering and Its Applications*, *7(5)*, 209-220.

[8] Mohammed, B., Bruel, J.-M., & Nabil, H. (2002). Generating test data from OCL specification. *Proceedings of the ECOOP'2002 Workshop on Integration and Transformation of UML Models*.

[9] Lestiennes, G. et G., & Marie-Claude. (2002). Testing processes from formal specifications with inputs, outputs and data types. *Proceedings of 13th International Symposium on Software Reliability Engineering* (pp. 3-14).

[10] Yoonsik, C., & Carlos, E. R.-M. (June 25-28, 2007). Random test data generation for Java classes annotated with JML specifications. *Proceedings of the 2007 International Conference on Software Engineering Research and Practice*: *Vol. II* (pp. 385-392). Las Vegas, Nevada.

[11] Yoonsik, C., Antonio, C., Martine, C., & Gary, T. L. (July 1-3, 2008). Integrating random testing with constraints for improved efficiency and diversity. *Proceedings of the 20th International Conference on Software Engineering and Knowledge Engineering* (pp. 861-866). San Francisco, CA.

[12] Fernandez, J.-C., Mounier, L., Cyril, et P. (2005). A model-based approach for robustness testing. *Testing of Communicating Systems*, 333-348, Springer Berlin Heidelberg.

[13] Leavens, G. T. (August 11, 2006). JML's rich inherited specification for behavioral subtypes. Department of Computer Science Iowa State University.

[14] Ciupa, I., Leitner, A., Oriol, M., & Meyer, B. (2007). Experimental assessment of random testing for

object-oriented software. *Proceedings of International Symposium on Software Testing and Analysis* (pp. 84–94).

[15] Csallner, C., & Smaragdakis, Y. (Sept. 2004). JCrasher: An automatic robustness tester for Java. *Software Practice and Experience*, *34(11)*, 1025–1050.

[16] Oriat, C. (Sept. 2005). Jartege: A tool for random generation of unit tests for Java classes. *Proceedings of International Conference on the Quality of Software Architectures*: *Vol. 3712* (pp. 242–256). Springer-Verlag,.

[17] Gaudel, M.-C. et L. G., *et al.* (2008). Testing data types implementations from algebraic specifications. *Formal Methods and Testing,* 209-239, Springer Berlin Heidelberg.

[18] Ferrer, J., Chicano, F., Enrique, et A. (2012). Evolutionary algorithms for the multi-objective test data generation problem. *Software: Practice and Experience*, *42(11),* 1331-1362.

[19] Ali, S., *et al.* (2012). Modeling robustness behavior using aspect-oriented modeling to support robustness testing of industrial systems. *Software & Systems Modeling*, *11(4)*, 633-670.

[20] Anand, S., Burke, E. K., *et al.* (2013). An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, *86(8)*, 1978-2001.

[21] Liskov, B. H., & Wing, J. (July 1999). Behavioral subtyping using invariants and constraints. Technical Report CMU CS-99-156, School of Computer Science, Carnegie Mellon University.

[22] Meyer, B. (1988). *Object-Oriented Software Construction*, Prentice Hall.

Khadija Louzaoui currently is a PhD student at the Faculty of Sciences, Ibn Tofail University, Kenitra, Morocco since 2014. She received a master's degree in network and system administration from the same faculty in 2010. She received her bachelor's degree in Java and C++ Development at the Faculty of Sciences, Ibn Tofail University. Her research interest includes software engineering and software testing …

Khalid Benlhachmi received his diploma of state engineer in computer science in 2005 from the School of engineers (ENIM) at Rabat (Morocoo) and obtained his PhD degree in computer science in 2013 from the Faculty of Sciences, University Ibn Tofail. In 2006 he joined the Faculty of Sciences Ibn Tofail University, as an engineer where he is currently a professor in the Department of Computer Science since 2014. His research interests relate to software engineering, software testing, Verification, performance and security software, graph theory …