SIMILARITY TESTING BY PROOF AND ANALYSIS OF PARTITION FOR OBJECT ORIENTED SPECIFICATIONS

[1]KHALID BENLHACHMI, [2]MOHAMMED BENATTOU
RLCST Research Laboratory in Computer Science and Telecommunications, University Ibn Tofail,Kénitra, Morocco
E-mail: [1]benlhachmi11@yahoo.fr, [2]mbenattou@yahoo.fr

ABSTRACT

The work presented in this paper is devoted to establish the theoretical model for the evaluation of the behavior of redefined methods in a subclass with the behavior of the original methods in the super-class using the inheritance mechanism. We analyze firstly, how a redefined method can use the specification of its corresponding method in the super-class. Secondly, we present the relationship between the test model of a redefined method in a subclass and the original method in a super-class. Our approach proposes a new concept which compares the behavior of methods, and gives the conditions where this comparison can induce a similar behavior.

Keywords–*Inheritance, Formal Specification, Conformity Test, Constraints Resolution, Valid Data, Invalid Data, Test Data Generation.*

1. INTRODUCTION

Formal methods have the potential to improve both quality and productivity in software development, and to circumvent expensive problems in traditional development practices, to enhance early error detection, to enable formal modeling and analysis, and to facilitate verifiability of implementation. Indeed, in object oriented modeling, a formal specification defines operations by collections of equivalence relations and is often used to constrain class and type, to define the constraints on the system states (invariant), to describe the pre- and post-conditions on operations and methods, and to give constraints of navigation in a class diagram. OCL [1] and JML [2] seem to be now the main used languages to formulate the constraints for an object-oriented model.

Software testing can be formalized and quantified when a solid basis for test generation can be defined [3]. In this context a number of researches focus on extract test data from a formal specification. The test data generation uses the constraints that are defined as restriction on one or more values in object-oriented model. Most testing methods attempt to test the conformity of class implementation from its specification using the values domain defined by the constraint rules on the individual operations and methods. The generated data from the values domain with respect to the well defined constraints can be generated randomly or using constraints resolution.

In [4], we present the definition of a formal model of constraint, illustrating the relationship between pre-condition and post-condition of individual methods of a class, and then we have shown how the invariant of a class can be used as a necessary condition for the truth of this constraint. We have formalized a generic constraint of a given individual method of class that contains the pre-condition, post-condition, and the invariant into a single logical formula. The given model translates algebraically the contract between the user (the calling program) and the called method.

The work presented in this paper allows to extend the constraint defined in [4] for modeling the specification of a redefined method in subclass using inheritance principle. This work extends our basic model [5] for similarity testing of inheritance by constructing the domain partitions for specifying all cases of methods similarity and by proving formally the correctness of the model. The main objective is to establish theoretical model allowing the evaluation of the behavior of redefined methods in a subclass with the behavior of the original methods in the super-class.

We analyze firstly, how a redefined method can use the specification of its corresponding method in the super-class. Secondly, we present the relationship between the test model of a redefined method in a subclass and the original method in a super-class. Our approach proposes a new concept which compares the behavior of methods, and gives the conditions where this comparison can induce a similar behavior.

This paper is organized as follows: in section 2 we present a related works and similar approaches for generating test data from a formal specification, in section 3 we describe theoretical aspects of our test process, and we define a formal model of similarity constraints, in section 4 we define the matrix partitions deduced from the formal model and show how these partitions can be used to test the similarity of the redefined methods, in section 5 we present how the testing formal model can be used to generate data of the similarity test, and finally we describe our approach with an example of similarity testing for an object oriented model.

2. RELATED WORKS

Most works have studied the problem of relating types and subtypes with behavioral specification in an object-oriented paradigm. These proposed works show how the contracts are inherited when a method is redefined in a sub-class and how the testing process can use the formal specification.

In [5] we develop a basic model for the concept of methods similarity, the test is based only on a random generation of input data.

In [6], the authors propose to generate randomly test data from a JML specification of objects class. They classify the methods and constructors according to their signature (basic, extended constructors, mutator, and observer) and for each type of individual method of class, a generation of test data is proposed. In [7], 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.

The work presented in [8] shows how to enforce contracts if components are manufactured from class and interface hierarchies in the context of Java. It also overcomes the problems related to adding behavioral contracts to Object-Oriented Languages, in particular, the contracts on overriding methods that are improperly synthesized from the original contracts of programmer in all of the existing contract monitoring systems.

The work is based on the notion of behavioral sub-typing; it demonstrates how to integrate contracts properly, according to the notion of behavioral sub-typing into a contract monitoring tool for java.

In [9], the authors treat the problem of types and subtypes with behavioral specifications in object-oriented world. They present a way of specification types that makes it convenient to define the subtype relation. They also define a new notion of the subtype relation based on the semantic properties of the subtype and super-type. In [10], they examine various notions of behavioral sub-typing proposed in the literature for objects in component-based systems, where reasoning about the events that a component can raise is important.

All the proposed works concerning the generation of test data from formal specification or to test the conformity of a given method implementation, are focused only on constraint propagation from super to subclass related to subtype principle. The work presented in this paper proposes a new approach for testing the redefined methods in subclasses using the basic specifications in super-classes, and allows specifying system behavior by constructing a model in terms of mathematical constructs.

3. FORMAL MODEL OF CONSTRAINT

This section presents a formal model of the generalized constraint defined in [4] which provides a way of modeling the specification of a redefined method by inheritance from a super-class. Indeed, the object of this section is to establish a series of theoretical concepts in order to create a solid basis for comparing the behavior of redefined methods in a subclass and the behavior of the original methods in the super-class.

3.1. Formal Model Of Constraint For A Basic Class

We present in [4] the definition of a formal model of constraint, illustrating the relationship between the pre-condition, the post-condition of a method and the invariant of the class. We use the same notation used in [4]: P is the precondition of the method m , Q is the post-condition of the same method and Inv is the invariant of the class C..

Let C be a class, and m be a method of n arguments $x=(x_1 , x_2 ,\ldots, x_n)$. We define for each argument x_i its domain of values $E_{i.}$.
We denote $E=E_1\times E_2\times\ldots E_n$ the domain of input vector of the method m. We have defined in [4] the generalized constraint H of a method m of class C as a logical property of the pair (x,o) with x the vector of parameters and o the receiver object such that:

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

Where I_c is the set of instances of the class C.

Indeed, the invocation of a method m is generally done by reference to an object o and consequently, m is identified by the couple (x,o) The logical implication in the proposed formula means that: each call of method with (x,o) satisfying the precondition P and the invariant before the call, (x,o) must necessarily satisfy the post-condition Q and the invariant Inv after the call. In the context of this work, we assume that the object which invokes the method under test is valid (satisfying the invariant of its class), therefore, the objects used at the input of the method are generated from a valid constructor.

This justifies the absence of predicate Inv of the object o before the call to m in the formula H (Figure 1).

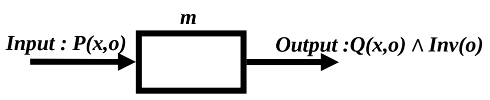

*Figure1. Specification Of A Method **M***

The evaluation of the constraint H (for all (x,o)) is done in two steps:
- In the Input of m, the evaluation of P(x,o).
- In the Output of m, the evaluation of Q(x,o) and Inv (o).

3.2. Formal Model Of Constraint For Inheritance

The purpose of this paragraph is to establish a series of theoretical rules and definitions in order to evolve the constraint H of a method m of a super-class during the operation of simple inheritance. We consider a method m of a class C_2 which inherits from a super-class C_1 such that m is redefinition of a method of C_1.

The method m and its redefinition in the subclass C_2 will be denoted respectively by $m^{(C1)}$, $m^{(C2)}$.
$(P^{(C1)},Q^{(C1)} ,Inv_{C1})$ denote respectively the pre-condition, the post-condition of the basic method $m^{(C1)}$, and the invariant of C_1 .
$(P^{(C2)} ,Q^{(C2)},Inv_{C2})$ denote respectively the pre-condition, the post-condition of the redefined method $m^{(C2)}$, and the invariant of the class C_2 .

The results of this paragraph are based on the works of Liskov (Principle of Substitution) [11,12] and Meyer [13] who have studied the problem relating to types and subtypes with behavioral specification in an object-oriented paradigm. These works show that the contracts are inherited when a method is redefined in a sub-class C_2 . Indeed, a subclass method $m^{(C2)}$ must:

- Accept all valid input to the super-class method $m^{(C1)}$.
- Meet all guarantees of the super-class method $m^{(C1)}$.

Consequently, the precondition of redefined operation $m^{(C2)}$ in subclass C_2 must be equal to or weaker than the precondition of corresponding operation $m^{(C1)}$ in super-class C_1.Therefore, we have: $P^{(C2)} \Leftrightarrow (P^{(C1)} \vee \alpha)$ (1)

Where α is the predicate added by the redefined method m in C_2 .Thus the precondition of m in C_2 is formed by two constraints: $P^{(C1)}$ inherited from the basic class C_1 and α the specific precondition of $m^{(C2)}$.

The post-condition of a redefined operation $m^{(C2)}$ must be equal to or stronger than the post-condition of corresponding operation $m^{(C1)}$ in super-class C_1 .

Thus, we have: $Q^{(C2)} \Leftrightarrow (Q^{(C1)} \wedge \beta)$ (2)
Where β is the predicate added by the redefined method $m^{(C2)}$.

Thus the post-condition of m in C_2 is formed by two constraints: $Q^{(C1)}$ inherited from the super-class C_1 and β the specific post-condition of $m^{(C2)}$.
The class invariant Inv_{C2} of the sub-class C_2 must be equal to or stronger than the class invariant Inv_{C1} of the C_1.

Thus, we have: $Inv_{C2} \Leftrightarrow (Inv_{C1} \wedge \lambda)$ (3)
Where λ is the predicate added by the sub-class C_2 .

Thus the invariant of C_2 is formed by two constraints: Inv_{C1} inherited from C_1 and λ the specific invariant of C_2.

Based on the definition of the generalized constraint, we have for $(x,o) \in E \times I_{C1}$:
$H^{(C1)}(x,o)$: $P^{(C1)}(x,o) \Rightarrow [Q^{(C1)}(x, o) \wedge Inv_{C1}(o)]$: The generalized constraint of the method $m^{(C1)}$.
And for $(x,o) \in E \times I_{C2}$:
$H^{(C2)}(x,o)$: $P^{(C2)}(x,o) \Rightarrow [Q^{(C2)}(x, o) \wedge Inv_{C2}(o)]$: The generalized constraint of the method $m^{(C2)}$.
Using (1),(2),(3) the constraint of $m^{(C2)}$ will have the following form :
$H^{(C2)}$: $(P^{(C1)} \vee \alpha) \Rightarrow [(Q^{(C1)} \wedge \beta) \wedge (Inv_{C1} \wedge \lambda)]$

3.3. Formal Model Of Constraint For Similar Methods

We propose in this section the definition of a new concept to study the compatibility between the redefined methods and original methods in the super-class.
The objective is to find a relationship between the constraint $H^{(C2)}$ of $m^{(C2)}$ and the constraint $H^{(C1)}$ of $m^{(C1)}$.

The two constraints : $H^{(C1)}:(P^{(C1)} \Rightarrow (Q^{(C1)} \wedge Inv_{C1}))$ and $H^{(C2)}:(P^{(C1)} \vee \alpha) \Rightarrow (Q^{(C1)} \wedge Inv_{C1} \wedge \beta \wedge \lambda)$ have common predicates $:(P^{(C1)},Q^{(C1)},Inv_{C1})$ (Figure2).

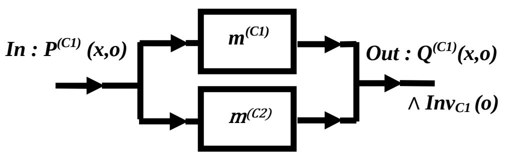

Figure 2. Common specification of $\boldsymbol{m^{(C1)}}$ *and* $\boldsymbol{m^{(C2)}}$

To compare two formulas $H^{(C1)}$ and $H^{(C2)}$, we analyze at first , the behavior of the two methods $m^{(C1)}$ and $m^{(C2)}$ relatively to the common specification (Figure 2).

In the particular case of figure 3, the fixed pair (x_0,o_0) of $E \times I_{C2}$ satisfies the basic precondition at the input of $m^{(C1)}$ and $m^{(C2)}$ $(P^{(C1)}(x_0,o_0)=1)$, however the basic post-condition $Q^{(C1)}(x_0,o_0)$ takes a true truth-value at the output of $m^{(C1)}$ and a false truth-value at the output of $m^{(C2)}$: this means that the same logical predicate for the same value of (x,o) $((x,o)=(x_0,o_0))$ can be True and False at the same time.

This type of situation makes impossible to establish a logical relationship between the theoretical model of the original method $m^{(C1)}$ and the model of $m^{(C2)}$ in the subclass (Figure 3).

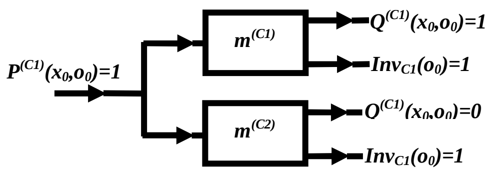

Figure 3. Example Of Non-Similarity

Our approach will rectify these types of problems by defining a new concept which compares the behavior of the methods $m^{(C1)}$ and $m^{(C2)}$ relatively to the common specification. Indeed, the original method in the basic class must evolve in subclasses with the condition that its behavior relatively to the basic specification is similar to the original version.

This comparison can define a certain degree of compatibility that will make it possible to establish a logical relationship between the mathematical model of the basic method and model of the redefined method in a subclass. Consequently, in figure 4 we must have for each pair (x,o) of input: a'=a ,b'=b where a',a ,b',b are Boolean values.

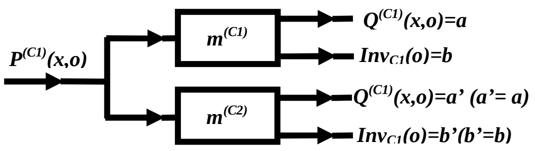

Figure 4. Condition Of Similarity

Definition 1: (Similarity of methods)

A method $m^{(C2)}$ of a class C_2 inheriting from the class C_1 is similar to the method $m^{(C1)}$ relatively to the basic specification $(P^{(C1)}, Q^{(C1)}, Inv_{C1})$ if :

- $m^{(C1)}$ is redefined by $m^{(C2)}$
- For each pair $(x,o) \in E \times I_{C2}$

$Q^{(C1)}(x,o)$ (respectively $Inv_{C1}(o)$) has the same truth-value in output of $m^{(C1)}$ and in output of $m^{(C2)}$.

The problem raised above by the logical mathematical will have a solution if the studied methods are similar relatively to their common specification. Indeed, we consider two similar methods $m^{(C1)}$ and $m^{(C2)}$ in classes C_1 and C_2 : We determine the logical equation between $H^{(C2)}$ and $H^{(C1)}$.
Using the following result:

$$[(a \vee b) \Rightarrow (c \wedge d)] \Rightarrow [(a \Rightarrow c)]$$

Such that a,b,c,d are logical predicates for find a relationship between $H^{(C2)}$ and $H^{(C1)}$.

We apply this result on the four predicates:

$P^{(C1)}, \alpha, (Q^{(C1)} \wedge Inv_{C1}), (\beta \wedge \lambda)$:

Therefore, we have the following result (R):

$$[H^{(C2)} \Rightarrow H^{(C1)}] \quad (R)$$

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 and allows to make a specific reasoning for each partition in order to identify the anomalies origin. In [4], we show how the constraint H may be used in the generation of domain partitions for each type of methods according to the classification proposed in [6]. This analysis of partition mainly concerns constructors and methods defined in a class without taking into account the inheritance relationships between classes.

The methods in subclass, during the operation of inheritance, are tested by a matrix partition: partition of similarity. This is a way to divide the input common domain of the two methods and to generate input data aimed at comparing the compatibility of a redefined method with the original version in a basic class relatively to their common specification.

4.1. General Analysis Of Partition

The partition analysis of a class method aims to construct domains of input values and to generate test data, this analysis can simplify the testing methodology. Indeed, the generalized constraint H is used in the partitioning process for extracting the possible cases of test values . In [4], we propose a partition (A,B) of the input domain $E \times I_C$ of a method in a basic class C. This partition is based on the truth cases of the constraint H:

$$A = \{(x,o) \in E \times I_C \,/\, H(x,o)=1\}$$

And $B = \{(x,o) \in E \times I_C \,/\, H(x,o)=0\}$

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

$A_1=\{(x,o) \in E \times I_C / (P(x,o),Q(x,o),Inv(o))=(1,1,1)\}$

And the partition A_2 of the invalid domain:

$A_2= \{(x,o) \in E \times I_C / (P(x,o),Q(x,o),Inv(o))=(0,?,?)\}$

For the domain B , we deduce the following three partitions :

$B_1=\{(x,o) \in E \times I_C / (P(x,o),Q(x,o),Inv(o)) =(1,1,0)\}$.

$B_2=\{(x,o) \in E \times I_C / (P(x,o),Q(x,o),Inv(o)) =(1,0,1)\}$.

$B_3=\{(x,o) \in E \times I_C / (P(x,o),Q(x,o),Inv(o)) =(1,0,0)\}$.

The partition (A_1, B_1, B_2, B_3) of input domain $E \times I_C$ for a method defines the elements which satisfy the precondition (i.e. the valid domain).

These domains of partition can be reduced if we consider the classification of methods (C_b , C_e , M , O) defined in [6]. The results of this paragraph are applied essentially to constructors and methods of a basic class.

4.2. Similarity Partition

We propose in this paragraph a matrix partition of the input domain for a method $m^{(C2)}$ in a subclass, the aim of this matrix partition is to compare the compatibility between $m^{(C2)}$ and $m^{(C1)}$ relatively to the basic specification.

For each input data (x,o) of the method $m^{(C2)}$, we associate a square matrix of size 2 × 2 with Boolean values (0 or1) which represents the four possible values for the quadruplet : (a ,b, a', b') (Figure 4).

Definition 2: (Similarity Application)

We define the application Similarity that associates each pair (x,o) of $E \times I_{C2}$ to its similarity matrix :

Similarity: $E \times I_{C2} \longrightarrow \mathcal{M}_2(\{0,1\})$

$(x,o) \rightarrow Similarity(x,o)=\begin{pmatrix} s_{11} & s_{12} \\ s_{21} & s_{22} \end{pmatrix}$ with $s_{ij} \in \{0,1\}$.

s_{11} = The truth-value of $Q^{(C1)}(x,o)$ after the call of $m^{(C1)}$

s_{12} = The truth-value of $Inv_{C1}(o)$ after the call of $m^{(C1)}$

s_{21}= The truth-value of $Q^{(C1)}(x,o)$ after the call of $m^{(C2)}$

s_{22}=The truth-value of $Inv_{C1}(o)$ after the call of $m^{(C2)}$

The first row (s_{11} , s_{12}) of the matrix corresponds to the original method, and the second row (s_{21} , s_{22}) corresponds to the redefined method .

We deduce that $m^{(C1)}$ and $m^{(C2)}$ are similar only if: for each input data (x,o) the two rows of the matrix Similarity (x,o) are identical :

$$(s_{11}, s_{12}) = (s_{21}, s_{22})$$

We divide $E \times I_{C2}$ on two subset Sim , NotSim :

The elements of Sim satisfy the constraint of similarity.

The elements of NotSim do not satisfy this constraint.

We divide Sim on four domains (Figure 5):

Sim_1 ,Sim_2 , Sim_3 , Sim_4 :

Sim_1 ={(x,o) ∈E × I_{C2} /Similarity(x,o) = $\begin{pmatrix}1 & 1\\1 & 1\end{pmatrix}$}

Sim_2 ={(x,o) ∈E × I_{C2} /Similarity(x,o) = $\begin{pmatrix}1 & 0\\1 & 0\end{pmatrix}$ }

Sim_3 ={(x,o) ∈E × I_{C2} / Similarity(x,o) = $\begin{pmatrix}0 & 1\\0 & 1\end{pmatrix}$ }

Sim_4 ={(x,o) ∈E × I_{C2} / Similarity(x,o) =$\begin{pmatrix}0 & 0\\0 & 0\end{pmatrix}$ }

Then, we divide NotSim on $NotSim_1$ and $NotSim_2$: The elements (x,o) of $NotSim_1$ do not satisfy the constraint of similarity and verify :

[($Q^{(C1)}$(x,o) = 1 and Inv_{C1}(o) = 1) after the call of $m^{(C2)}$] .

This induces three possible cases corresponding to the following three parts of $NotSim_1$ (Figure 5):

$NotSim_{11}$={(x,o)∈E×I_{C2}/Similarity(x,o)=$\begin{pmatrix}0 & 0\\1 & 1\end{pmatrix}$}

$NotSim_{12}$={(x,o)∈E×I_{C2}/Similarity(x,o)=$\begin{pmatrix}1 & 0\\1 & 1\end{pmatrix}$}

$NotSim_{13}$={(x,o)∈E×I_{C2}/Similarity(x,o)=$\begin{pmatrix}0 & 1\\1 & 1\end{pmatrix}$}

We deduce that the elements of $NotSim_1$ satisfy the necessary condition:

$$\forall(x,o) \in E \times I_{C2} : [(x,o) \in NotSim_1] \Rightarrow [(Q^{(C1)}(x,o)= 0 \text{ or } Inv_{C1}(o) = 0) \text{ after the call of } m^{(C1)}]$$

The elements (x,o) of $NotSim_2$ do not satisfy the constraint of similarity and constitute the rest cases. This induces nine cases distributed on subsets of $NotSim_2$ (Figure 5):

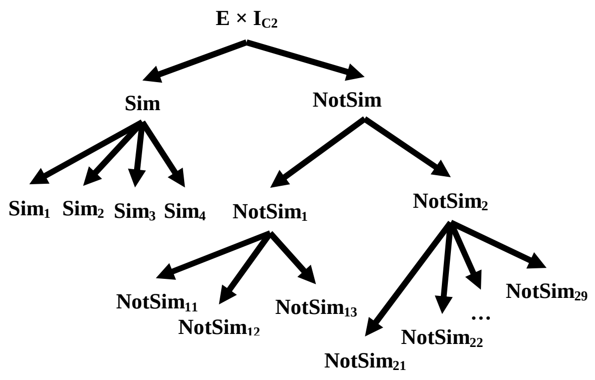

Figure 5. Similarity Tree

$NotSim_{21}$={(x,o)∈E×I_{C2}/Similarity(x,o)=$\begin{pmatrix}1 & 1\\0 & 0\end{pmatrix}$}

$NotSim_{22}$={(x,o)∈E×I_{C2} /Similarity(x,o)=$\begin{pmatrix}1 & 0\\0 & 1\end{pmatrix}$}

$NotSim_{23}$={(x,o)∈E×I_{C2}/Similarity(x,o)=$\begin{pmatrix}0 & 1\\1 & 0\end{pmatrix}$}

$NotSim_{24}$ ={(x,o)∈E×I_{C2}/Similarity(x,o)‘=$\begin{pmatrix}1 & 1\\1 & 0\end{pmatrix}$}

$NotSim_{25}$ ={(x,o) ∈E×I_{C2}/Similarity(x,o)=$\begin{pmatrix}1 & 1\\0 & 1\end{pmatrix}$}

$NotSim_{26}$={(x,o)∈E×I_{C2}/Similarity(x,o)=$\begin{pmatrix}0 & 1\\0 & 0\end{pmatrix}$}

$NotSim_{27}$={(x,o) ∈E×I_{C2}/Similarity(x,o)=$\begin{pmatrix}0 & 0\\0 & 1\end{pmatrix}$}

$NotSim_{28}$={(x,o) ∈E×I_{C2}/Similarity(x,o)=$\begin{pmatrix}1 & 0\\0 & 0\end{pmatrix}$}

$NotSim_{29}$ ={(x,o) ∈E×I_{C2}/Similarity(x,o)=$\begin{pmatrix}0 & 0\\1 & 0\end{pmatrix}$}

We deduce that the elements of $NotSim_2$ satisfy the necessary condition:

$$\forall(x,o) \in E \times I_{C2} : [(x,o) \in NotSim_2] \Rightarrow [(Q^{(C1)}(x,o) = 0 \text{ or } Inv_{C1}(o) = 0) \text{ after the call of } m^{(C2)}]$$

5. SIMILARITY TESTING

The formal model of test proposed in [4] defines the notion of method validity in a basic class. This model is a way to generate test data for conformity. The testing process proposed in this section compares the compatibility between a redefined method in a subclass and its original version in the super-class. Our approach will be evaluated by implementing the algorithm of similarity testing for inheritance.

5.1. Formal Model Of Test For A Basic Class

In [4], we test the conformity of methods in a basic class without taking into account the inheritance relationship: the model of test generates random data at the input of a method using elements of the valid domain which satisfy the precondition of the method under test.

The test algorithm is based on the partition :(A_1 ,B_1 ,B_2 ,B_3) , this test stops when the constraint H becomes False (H(x,o)=0) or when the maximum threshold of the test is reached with H satisfied.

Definition 3: (Valid method)

A method m of class C is valid or conforms to its specification if for each (x,o) ∈ E×I_C , the constraint H is satisfied :

$$\forall (x,o) \in E \times I_C : H(x,o) = 1$$

With *o* the receiver object and *x* the parameters vector.

Using the definition 3, a method m is invalid if:

$$\exists (x,o) \in E \times I_c : H(x,o) = 0$$

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

5.2. Similarity Test

The results of the last paragraph are applied to the methods without considering the inheritance relationship. In this section we test the similarity of a redefined method with its original version. This operation compares the behavior of the two methods relatively to their common specification.

The similarity test generates random input data which satisfy the basic precondition of both methods ($P^{(C1)}$(x,o)=1) and compares at the output of each method the behavior relatively to the common specification (Figure 6).

In the algorithm of figure 6 , we choose a threshold of test N and we generate randomly the pairs (x,o) ∈ E×I_{C2} which satisfy the basic precondition $P^{(C1)}$, and for each (x,o) we compare the truth-value of the basic post-condition $Q^{(C1)}$ for this (x,o) at the output of $m^{(C1)}$ and $m^{(C2)}$,at the same time we compare the truth-value of the basic invariant $Inv_{(C1)}$ at the output of $m^{(C1)}$ and $m^{(C2)}$.

We assume that the objects used are generated from a valid constructor of the subclass, and the three sets Sim , $NotSim_1$, $NotSim_2$ are initialized to ∅ for each call of the algorithm.

```
do{
   do{
      for ( $x_i$ in $m^{(C2)}$ parameter)
         {$x_i$ = generate ( $E_i$);}
      x=($x_1$,$x_2$,…,$x_n$);
      o = generate_object($C_2$);
   }while(!$P^{(C1)}$(x,o));
(x',o')=copy(x,o);
invoke"o. $m^{(C1)}$(x)";
$S_{11}$=$Q^{(C1)}$(x,o);$S_{12}$= $Inv_{C1}$(o);
(x,o)=copy(x',o');
invoke"o. $m^{(C2)}$(x)";
$S_{21}$=$Q^{(C1)}$(x,o);$S_{22}$=$Inv_{C1}$(o);
(x,o)=copy(x',o');
if(($s_{11}$,$s_{12}$)=($s_{21}$,$s_{22}$ ) )
   Sim.add(x,o);

elseif(($s_{21}$,$s_{22}$)∈{(0,0),(0,1),(1,0 )})
   NotSim$_2$.add(x,o);
else
   NotSim$_1$.add(x,o);
     }while(Sim.size()<N
     && NotSim$_1$.isEmpty()
     && NotSim$_2$.isEmpty());
```

Figure 6. Similarity Test Algorithm of a Redefined Method

The test stops when we find a pair (x,o) for which the two rows of the matrix Similarity(x,o) are not identical. In this case, the methods $m^{(C1)}$ and $m^{(C2)}$ are not similar relatively to the inherited specification.

If we reach the threshold N of test without identifying a difference between the two rows for every matrix Similarity, we may admit with a rejected error margin (the limit N must be sufficiently large ($N \longrightarrow \infty$)) that the methods $m^{(C1)}$ and $m^{(C2)}$ are similar.

5.3. Evaluation

We evaluate the correctness of our approach by implementing the algorithm of similarity testing for inheritance.

As is indicated above, the similarity test of a redefined method in sub class requires passing through a test of a basic constructor in order to use the valid objects in testing process.

We consider for example of the similarity test for inheritance the methods deposit and withdraw of the class Account1 and Account2 described in figure 7 and figure 8:

```
class Account1
{
protected double bal;/* bal is the
account balance*/
public Account1(double x1){
this.bal=x1;}
public Account1 (Account1 x1){
this.bal=x1.bal;}
public void transfer(int x1,Account1
x2){…}
public void withdraw (int x1){
       this.bal=this.bal - x1;}
public void deposit (int x1){
       this.bal=this.bal + x1;}
}
```

Figure 7. Account1 class

```
class Account2 extends Account1
{
private double InterestRate;
…
public Account2(double x1, double x2)
{super(x1);this.InterestRate=x2;…}
public  Account2  (Account1  x1,double
x2,…)
{ super(x1); this.InterestRate=x2;…}
public void deposit (int x1)
{super.deposit(x1);
if (x1 >=(this.bal/2))
this.bal=this.bal+(this.InterestRate)*
x1;
InterestRate=InterestRate*(1.25);}
public void withdraw (int x1)
    {super.withdraw(x1);
if (x1 > bal)
this.bal=this.bal-
(this.InterestRate)*x1;
```

Figure 8. Account2 class

- **Similarity test of deposit$^{(1)}$ and deposit$^{(2)}$**

The constraint H$^{(1)}$ of the deposit method in the class Account1 in an algebraic specification with x= x_1 :

$P^{(1)}(x,o)$: ($x_1 \geq 10$)
$Q^{(1)}(x,o)$: ($balance(o_{(a)}) \geq balance(o_{(b)})$)
$Inv_1(o)$: $balance(o) \geq 0$

The constraint H$^{(2)}$ of the deposit method in class Account2 in an algebraic specification with x= x_1 :

$P^{(2)}(x,o)$: ($P^{(1)}(x,o) \vee 0 \leq x_1 < 10$)
$Q^{(2)}(x,o)$: $Q^{(1)}(x,o) \wedge [\, balance(o_{(a)}) \in \{(balance(o_{(b)}) + x_1) , (balance(o_{(b)}) + (1+InterestRate) \times x_1)\}\,]$
$Inv_2(o)$: ($balance(o) \geq 0$) $\wedge$ ($0 < InterestRate\ (o) \leq 0{,}3$)
Where $o_{(a)}$ and $o_{(b)}$ are respectively the object o after and before the call of the method.

In order to test the similarity of deposit methods in class Account1 and Account2, we generate randomly x_1 and the balance values in the interval]-200,200[with the threshold limit N=100(Table 1).

Table 1. Result of a similarity test of the deposit methods

Iteration number:	***x***	***O***	***P$^{(1)}$(x,o)***	***(x,o) ∈***
1	*21*	*Account2(74,0.2)*	*1*	*Sim*
2	*45*	*Account2(130,0.1)*	*1*	*Sim*
3	*183*	*Account2(167,0.23)*	*1*	*Sim*
…	…	…	…	…
….	….	….	….	….
…..	…..	…..	…..	…..
98	*79*	*Account2(112,0.14)*	*1*	*Sim*
99	*101*	*Account2(87,0.11)*	*1*	*Sim*
100	*157*	*Account2(142,0.28)*	*1*	*Sim*

The test result shows that for 100 iterations the size of the set Sim is exactly the threshold limit of the test, this leads to the conclusion that the deposit$^{(1)}$ and deposit$^{(2)}$ are similar relatively to the basic specification (Table 1).

- **Similarity test of withdraw$^{(1)}$ and withdraw $^{(2)}$**

The constraint H$^{(1)}$ of the withdraw$^{(1)}$ in an algebraic specification with x = x_1 :

$P^{(1)}(x,o)$: ($0 \leq x_1 \leq balance(o)$)
$Q^{(1)}(x,o)$: ($balance(o_{(a)}) \leq balance(o_{(b)})$)
$Inv_1(o)$: $balance(o) \geq 0$

The constraint H$^{(2)}$ of the withdraw $^{(2)}$:

$P^{(2)}(x,o)$: ($P^{(1)}(x,o)$)
$Q^{(2)}(x,o)$:($Q^{(1)}(x,o) \wedge (balance(o_{(a)}) \in \{(balance(o_{(b)}) - x_1),(balance(o_{(b)}) - (1+InterestRate) \times x_1)\}$)
$Inv_2(o)$:
($balance(o) \geq 0$) $\wedge$ ($0 < InterestRate(o) \leq 0.3$)

The test of similarity for withdraw methods in class Account1 and Account2 with the same conditions as deposit gives the following results (Table 2)

Table 2. Result Of A Similarity Test Of The Withdraw Methods

Iteration number:	***x = x$_1$***	***O***	***P$^{(1)}$(x,o)***	***(x,o) ∈***
1	*105*	*Account2(146,0.02)*	*1*	*Sim*
2	*37*	*Account2(87,0.13)*	*1*	*Sim*
3	*73*	*Account2(93,0.21)*	*1*	*Sim*
4	*141*	*Account2(188,0.17)*	*1*	*Sim*
5	*52*	*Account2(61,0.25)*	*1*	*NotSim$_2$*

As is shown in table 2,the methods withdraw $^{(1)}$ and withdraw $^{(2)}$ are not similar, the pair (x,o)=(52 , Account2(61 ,0.25)) in the iteration 5 does not satisfy the constraint of similarity :

$$\text{Similarity}(52\,,\text{Account2}\;(61\,,0.25)) = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

- **Analysis with proof**

We use an analysis with proof to demonstrate the non- similarity of Withdraw $^{(2)}$ and Withdraw $^{(1)}$.
For a value of (x,o) that satisfy the basic precondition ($P^{(1)}$(x,o)=1) of the method Withdraw$^{(2)}$ we have :

$$[0 \le x_1 \le \text{balance}(o)]\,,\, x=x_1$$

We study for example the truth-value of the basic invariant after the call of methods Withdraw$^{(2)}$ and Withdraw$^{(1)}$, for this we look for a necessary condition satisfied by the basic invariant after the call of Withdraw$^{(2)}$.

Using the relation $[0 \le x_1 \le \text{balance}(o)]$, we have two cases to be treated:

Case1: $[0 \le x_1 \le (1/2).\text{balance}\;(o_{(b)})]$

The behaviors of Withdraw$^{(2)}$ and Withdraw$^{(1)}$ are identical relatively to the basic specification, Indeed, analysis with proof of this methods shows that the condition of the block " if " is not satisfied ,and consequently we have (Figure 8):

$$\text{balance}(o_{(a)}) = \text{balance}(o_{(b)}) - x_1$$

Case2:$[(1/2).\text{balance}\;(o_{(b)}) \le x_1 \le \text{balance}\;(o_{(b)})]$

We have in the context of the case 2 (Figure 8):

$$\text{balance}(o_{(a)}) = \text{balance}\;(o_{(b)}) - x_1 \\ -\text{InterestRate}(o_{(b)}) \times x_1$$

The basic invariant is satisfied if and only if :

$$(\,\forall o)\,:\;\text{balance}(o_{(a)})\;\ge 0$$

Consequently, the necessary condition for satisfy the basic invariant in the case 2 is:

$$(\,\forall o)\,:\;\;x_1\;\le \frac{\text{balance}(o_{(b)})}{1+\,\text{InterestRate}(o_{(b)})}$$

The element (x,o) = (52 , Account2(61 ,0.25)) does not satisfy this condition (i.e. 52 > 61/(1+0.25)) (Table 2).

Consequently, the pair (x,o) = (52 , Account (61 ,0.25)) satisfy the basic invariant after the call of Withdraw$^{(1)}$, but the same pair does not satisfy the basic invariant after the call of Withdraw$^{(2)}$.
We can conclude that the methods withdraw$^{(2)}$ and Withdraw$^{(1)}$ are not similar relatively to the basic specification ($P^{(1)}$, $Q^{(1)}$, Inv_1) .

6. CONCLUSION

The approach of this paper proposes a new concept of test which represents a way to compare the behaviors of methods in sub-classes and their original versions in the super-classes for an object oriented specification. The test process gives the conditions where the comparison can induce a similar behavior. The result of this test constitutes a solid basis to reuse the inherited specifications in the sub-classes.

We analyze firstly, how a redefined method can use the specification of its corresponding method in the super-class. Secondly, we present the relationship between the test model of a redefined method in a subclass and the original method in a super-class. The principal idea of the proposed work is based on the matrix partitions for similarity testing of inheritance by constructing the domain partitions for specifying all cases of methods similarity and by proving formally the correctness of the model.

Our future works are oriented to develop and generalize the formal model of conformity defined for a basic class, and to apply the similarity process to test the conformity in subclasses.

REFERENCES

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

[2] F. Bouquet, F. Dadeau, B. Legeard, and M. Utting. Symbolic animation of JML specifications. *In International Conference on Formal Methods*, volume 3582 of LNCS, pages 75– 90. Springer-Verlag, July 2005.

[3] Mohammed Benattou, Jean-Michel Bruel, and Nabil Hameurlain. "Generating Test Data from OCL Specification". *In Proceedings of the ECOOP'2002 Work-shop on Integration and Transformation of UML models* (WITUML'2002), 2002.

[4] Khalid Benlhachmi, and M.Benattou, and Jean-louis Lanet. "Génération de Données de Test Sécurisé à partir d'une Spécification Formelle par Analyse des Partitions et Classification". *In Proceedings of the 6 th International Conference on Network Architectures and Information Systems Security* (SAR-SSI 2011) ,May 18-21,2011,La rochelle ,France.

[5] Khalid Benlhachmi, and M.Benattou."A Formal Model of Similarity Testing for Inheritance in Object-Oriented Software".*In Proceedings of the 2012 edition of the IEEE International Conference* (CIST'2012) ,October 24-26,2012, Fez, Morocco.

[6] Yoonsik Cheon and Carlos E. Rubio-Medrano. "Random Test Data Generation for Java Classes Annotated with JML Specifications". *In Proceedings of the 2007 International Conference on Software Engineering Research and Practice*, Volume II, June 25-28, 2007, Las Vegas, Nevada, pages 385-392

[7] Gary T.Leavens , "JML's Rich,Inherited Specification for Behavioral Subtypes", *Department of Computer Science Iowa State University* , August 11,2006.

[8] Robet Bruce Findler , and Mario Latendresse , and Matthias Felleisen ,"Behavioral Contracts and Behavioral Subtyping", *Foundations of Software Engineering* , Rice University , FSE 2001.

[9] Barbara H. Liskov and J. M. Wing , "Abehavioral notion of subtyping" , MIT Laboratory for Computer Science, Carnegie *Mellon University, ACM Transactions on Programming Languages and Systems*,Vol 16. No 6, November 1994, Pages 1811-1841.

[10] Gary T.Leavens , Krishna Kishore Dhara , "Concepts of Behavioral Subtyping and a Sketch of their Extension to Component-Based Systems" , *In G. T. Leavens and M. Sitaraman, editors, Foundations of Component-Based Systems* ,2000.

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

[12] Liskov, B. H. and J. M. Wing."A behavioral notion of subtyping ". *ACM Transactions on Programming Languages and Systems*, November 1994.

[13] Meyer, B. "Object-oriented Software Construction". *Prentice Hall*, 1988.