Journal of Theoretical and Applied Information Technology
15th December 2021. Vol.99. No 23
© 2021 Little Lion Scientific

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

AN OPTIMAL APPROACH OF CONFORMITY ASSESSMENT AND ROBUSTNESS TESTING FOR OBJECT ORIENTED CONSTRAINTS

KHADIJA LOUZAOUI [1], KHALID BENLHACHMI [2]

[1,2]Laboratory For Computer Science Research, Faculty Of Science, Ibn Tofail University, Kenitra, Morocco
[1] khadija.louzaoui1@uit.ac.ma, [2]khalid.benlhachmi@uit.ac.ma

ABSTRACT

In this work we propose a formal modeling of optimal constraints for testing the conformity contract and robustness behaviors of object oriented (OO) programs. Our approach is an important way to generate test data of overriding methods of the inheritance process in the general case where behaviors of OO classes are not necessarily similar. The key idea of this work is to use mathematical entities for developing some algorithms of test data generation to simplify conformity and robustness verification process.

Our model of constraints is based on set theory and logical axioms, and can represent in an unambiguous form all properties and behaviors of OO robustness contracts. The second model of this paper is an equivalence partitioning of input data of the program under test, this partitioning technique can be used to reduce the number of test cases that must be developed for classes and subclasses.

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

1. INTRODUCTION

Formal modelling is an important method of discovering system anomalies and presenting program properties in an unambiguous form. The techniques of formal specification and verification in computing science are used since 1940s: Turing showed how the logical properties about programs at input and output states can simplify and facilitate their conformity assessment [1]. Floyd, Hoare and Naur used axiomatic methods for verifying the consistency and the conformity contract between programs and their specification [2,3,4]. Dijkstra showed how to derive nondeterministic programs from formal calculus, properties, and the specification equation of the system under test [5].

In this paper we develop a constraint model that includes various abstraction levels and corresponding methods for synthesis and verification of conformity and robustness properties: the first method is a behavioural equivalence partitioning of input domains of derived classes of inheritance. The second method is an optimal model of constraints for describing all states of conformity and robustness of overriding methods. Our approaches are based on formal specifications and design by contracts (DBC) [6, 7, 8].

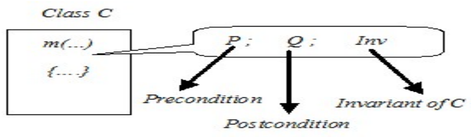


Figure1: Specification Of An OO Class

For object oriented (OO) programs, designs by contract represent a powerful technique for robust and reliable software. DBC is based on three Boolean constraints: precondition, postcondition and invariant (*P, Q, Inv*) (Fig.1).The specification (*P, Q, Inv*) must be satisfied in input and output of programs under test, and can be used by different languages of constraints: OCL[9] and JML[10]… .

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

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

This conformity constraint of the program under test is satisfied if: “For all invocation of the program, output specifications (*Q* and *Inv*) are satisfied **if** input specifications (*P* and *Inv*) are satisfied” (Fig.2).

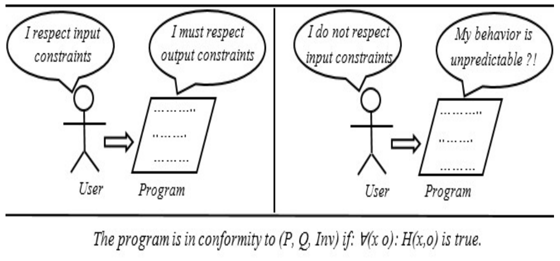

I respect input constraints
I must respect output constraints
User
Program
I do not respect input constraints
My behavior is unpredictable ?!
User
Program
The program is in conformity to (P, Q, Inv) if: ∀(x o): H(x,o) is true.

Figure 2: Conformity Contract Of An OO Program

Our previous approaches of conformity testing [11,12] and robustness testing [13] in inheritance are based on similarity of behaviours[14] between overridden and overriding methods. In our basic approaches we have indeed tested the conformity and robustness of overriding methods in derived classes from test results of overridden methods in the super class. This reusability of test sequences of the super class is only possible if overriding and overridden methods have same basic behaviour [14].

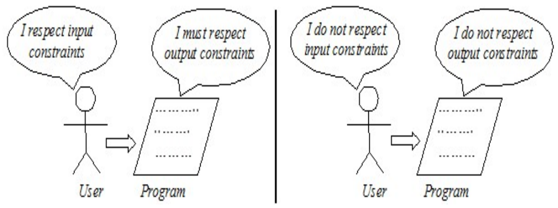

I respect input constraints
I must respect output constraints
User
Program
I do not respect input constraints
I do not respect output constraints
User
Program

Figure 3: Robustness Contract Of An OO Program

In this paper we present an approach for testing the conformity and robustness of overriding methods in derived classes in the general case where overriding and overridden methods are not necessarily similar. In this work we complete our basic robustness approach by measuring the robustness of programs in inheritance even if derived classes and super classes are dissimilar (Fig.3). The principle of this approach is based on an optimal model of constraints and an equivalence classes partitioning to generate test data of conformity and robustness. In this context this approach can be used to verify conformity and robustness contracts of subclasses of inheritance according to precondition, postcondition and invariant specifications.

We organize our paper as follows: section 2 and 3 present similar approaches of software testing and our previous works of conformity constraints in derived classes. In section 4 we propose our approach of conformity testing of inheritance by using an equivalence classes partitioning to generate test data. We present in section 5 our optimal model of robustness testing. Finally, our approach is evaluated by an OO example of conformity and robustness testing.

2. RELATED WORKS

Most works have studied the problem of test data generation and formal specifications 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 [9], the authors present a method based on the constraints resolution for test cases generation with error anticipation in the methods specification. In [10], they propose to use Java 8 streams for writing more concise and cleaner assertions on a collection. The use of streams in JML can be minimal and non-invasive in the conventional style of writing assertions. It can also be holistic to write all assertions in the abstract state defined by streams. In [11,12], we have proposed a formal model of constraints for testing conformity contracts. This approach is used only for testing conformity of similar classes. In [13] we have used an optimal constraint for testing robustness contracts of derived classes of inheritance.

The approach of [14] can be used to test the robustness of overriding methods in derived classes from test results of overridden methods in the super class. This reusability of test sequences of the super class is only possible if overriding and overridden methods have same basic behavior. In [15], 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 [16], the paper focuses on the basic ideas of formal EventML programming illustrated by implementing a fault-tolerant consensus protocol and showing how proving its safety properties with the Nuprl proof assistant. In [17], 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 [18], 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 [19], the authors present a new test model written in SysML and an associated black box test suite for the Ceiling Speed Monitor (CSM) of the European Train Control System (ETCS). The model is publicly available and intended to serve as a novel benchmark for investigating new testing theories and comparing the capabilities of model-based test automation tools. They apply a novel method for equivalence class testing that-despite the conceptually infinite cardinality of the input domains-is capable to produce finite test suites that are complete for a given fault model.

In [20], the authors present some of the key issues involved in model transformation specification and testing, and introduce the concept of Tract, a generalization of model transformation contracts. They show how Tracts can be used for model transformation specification and black-box testing, and the kinds of analyses they allow.

In [21], the paper presents an approach to define contracts of methods and their refinements in Feature-oriented programming (FOP) that is an extension of object-oriented programming to support software variability by refining existing classes and methods. In order to increase the reliability of all implemented program variants, the authors integrate design by contract (DbC) with FOP.

In [22], the authors propose a formal process to specify, verify and correct the security policy using the decision tree formalism, which consists of four steps. First, they define the security policy specifications and write it in a natural language. Second, the security policy will be translated into a formal language. Third, they verify the security policy correctness. If this latter is plugged with anomalies, they correct it in the last step. To achieve these goals, they present a decision tree based formalism for security policy verification and propose a correction algorithm to guarantee the security policy correctness. In [23], 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 [24], authors have studied the multi-objective test data generation problem. The authors in [25] 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 [26], the authors propose a theoretical framework for model based robustness testing together with an implementation within the If validation environment. In [27], the authors present a survey of some of the most prominent techniques of automated test data generation. The techniques presented include: structural testing using symbolic execution, model-based testing, combinatorial testing, random testing and its variant of adaptive random testing, and search-based testing.

In [28], the basic ACO algorithm is reformed into discrete version so as to generate test data for structural testing. First, the technical roadmap of combining the adapted ACO algorithm and test process together is introduced. In order to improve algorithm's searching ability and generate more diverse test inputs, some strategies such as local transfer, global transfer and pheromone update are defined and applied. In [29], the proposed approach generates the test data using Bi-Objective function based on genetic algorithm. The objective function includes space dispersity and path disparity which will produce better spatial distribution of input space. Furthermore, Clustering technique is applied to the generated test data to reduce the time of error finding ability.

The study of [30] aims to propose a novel fitness function of metaheuristic algorithms to generate test data based on the mutation technique for the Simulink models (Simulink is an environment widely used in industry to design and simulate critical systems). The fitness function is designed by analyzing each mutation operator and the features of blocks in the Simulink environment in order to guide the search process to reach the test data killing mutants more easily. Then, this fitness function is used in the multi-parent crossover genetic algorithm to generate test sets. The obtained results indicated that the mutation score has been significantly improved for all models when using the novel fitness function. In [31], they have focused on resolving the multi-objective optimization of coverage based test data by proposing Multi-Objective Ant Lion Optimization (MOALO) algorithm. Further, they have discussed that how the proposed algorithm enhance the path coverage with reduced number of tests. To validate the proposed algorithm, they have compared the obtained experimental results

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

with random resting and conventional genetic algorithm's data.

3. MODEL OF CONSTRAINTS FOR CONFORMITY TESTING

The work of [11] can be used for testing the conformity of an overriding method in derived classes during the inheritance operation by using constraint models of basic classes and constraints propagations.

3.1 Model of constraint for basic classes

The conformity contract (Fig.2) can be represented by the constraint model *H*.

Definition

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

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

- Where $o_{(bef)}$ is the class object o in the state before the calling of the method m( ) and $o_{(aft)}$ is the class object o in the state after the calling of the method m( ) (Fig.4).

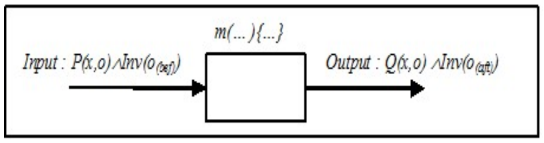


Figure 4: Input-Output Constraints Of M( )

3.2 Model of conformity testing in inheritance

In [11] we have used the model of constraint *H* for testing the conformity of methods in derived classes (Fig.5).

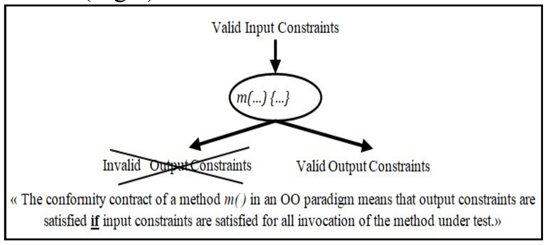

Valid Input Constraints
m(...) {...}
~~Invalid Output Constraints~~
Valid Output Constraints
« The conformity contract of a method m( ) in an OO paradigm means that output constraints are satisfied if input constraints are satisfied for all invocation of the method under test.»

Figure 5: Principle Of Conformity Testing

- *Constraints propagation in inheritance*

We consider a method *m* of a class C_2 which inherits from the class C_1 such that *m* overrides a method of C_1. The original method and its overriding method in the subclass C_2 will be denoted respectively by $m^{(1)}$, $m^{(2)}$ (Fig.6).

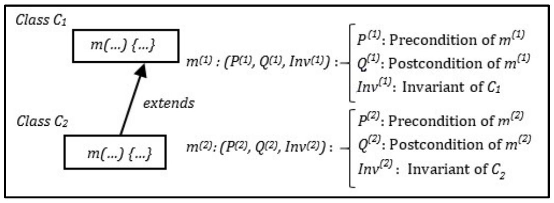

Class C1
m(...) {...}
m(1) : (P(1), Q(1), Inv(1)) :
P(1): Precondition of m(1)
Q(1): Postcondition of m(1)
Inv(1): Invariant of C1
extends
Class C2
m(...) {...}
m(2): (P(2), Q(2), Inv(2)) :
P(2): Precondition of m(2)
Q(2): Postcondition of m(2)
Inv(2): Invariant of C2

Figure 6: Constraints Of ($m^{(1)}$,$m^{(2)}$)

The problem of behavioural constraints of types (classes) and subtypes (subclasses) of object oriented programs is resolved by Meyer [6,7,8] and Liskov, Wing [32] (Fig.7).

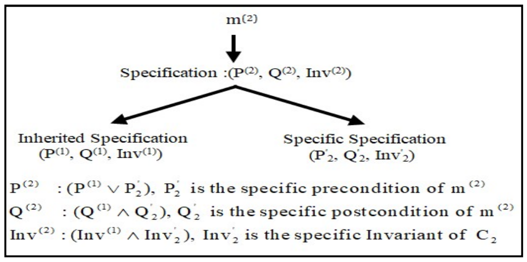

m(2)
Specification :(P(2), Q(2), Inv(2))
Inherited Specification (P(1), Q(1), Inv(1))
Specific Specification (P'2, Q'2, Inv'2)
P(2) : (P(1) ∨ P'2), P'2 is the specific precondition of m(2)
Q(2) : (Q(1) ∧ Q'2), Q'2 is the specific postcondition of m(2)
Inv(2) : (Inv(1) ∧ Inv'2), Inv'2 is the specific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

Figure 7: Specification Of $m^{(2)}$

In this approach, the specification ($P^{(2)}$,$Q^{(2)}$,$Inv^{(2)}$) of the overriding method $m^{(2)}$ is constituted by two specifications (Fig.7).

- *Constraint of conformity testing*

  - Conformity testing of overridden methods:

    -The overridden method $m^{(1)}$is in conformity with its specification if:

    $$\forall(x,o)\in E\times I_{C1}: H^{(1)}(x,o)$$

    -The overridden method $m^{(1)}$ is not in conformity with its specification if:

    $$\exists(x,o)\in E\times I_{C1}: \overline{H^{(1)}(x,o)}$$

  - Conformity testing of overriding methods:

    -The overriding method $m^{(2)}$ is in conformity with its specification if:

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

    -The overriding method $m^{(2)}$ is not in conformity with its specification if :

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

3.3 Similarity model

The similarity approach of our previous works [14] is used for assuring if the overriding method $m^{(2)}$ has the same behaviour as its original version $m^{(1)}$ in the superclass according to the inherited specification *($P^{(1)}$,$Q^{(1)}$,$Inv^{(1)}$)*. For each input value *(x,o)* of the overriding method $m^{(2)}$, we associate the matrix:

Similarity(x,o)=(a,b,a',b'). The matrix represents the 16 values of the quadruplet (*a ,b, a', b'*) (Fig.8).

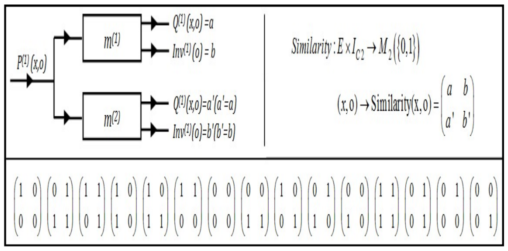

$P^{(1)}(x,o)$
$m^{(1)}$
$m^{(2)}$
$Q^{(1)}(x,o)=a$
$Inv^{(1)}(o)=b$
$Q^{(1)}(x,o)=a'(a'=a)$
$Inv^{(1)}(o)=b'(b'=b)$
$Similarity: E \times I_{C2} \rightarrow M_2(\{0,1\})$
$(x,o) \rightarrow Similarity(x,o) = \begin{pmatrix} a & b \\ a' & b' \end{pmatrix}$
$\begin{pmatrix}1&0\\0&0\end{pmatrix}\begin{pmatrix}0&1\\1&1\end{pmatrix}\begin{pmatrix}1&1\\0&1\end{pmatri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

Figure 8: Condition And Equivalence Partitioning Of Similarity

The methods $m^{(1)}$ and $m^{(2)}$ are similar to the specification $(P^{(1)},Q^{(1)}, Inv^{(1)})$ if and only if :
$(a,b)=(a',b')$ and $(a,b,a',b') \in \{0,1\}^4$ (Fig.8).

In the approach of this paper we show that the similarity of behaviour is not obligatory for verifying conformity in sub classes. So the conformity of dissimilar methods can be tested. The purpose of next sections is to generalize the model of [14] in order to test the conformity of an overridden and overriding methods ($m^{(1)}$ and $m^{(2)}$) even if methods are not necessarily similar.

4. OPTIMAL MODEL OF CONSTRAINTS AND INPUT DATA PARTITIONING

We have shown in [12] that there is no requirement to use the constraint model $H^{(2)}$ in subclasses for testing conformity contracts of overriding methods. In our approach, we propose an optimal model of constraints H_{op}. The optimal model H_{op} is a refinement of the current model H by eliminating unnecessary constraints and reducing some responsibilities of test data.

4.1 Optimal model of constraints

The Conjunctive Normal Form (CNF) can be used to reduce all invalid and unnecessary constraints.
The constraint model $H^{(2)}$in the CNF :

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

$$[P^{(1)} \Rightarrow (Q^{(1)} \wedge Inv^{(1)})] \wedge [P_2^{'} \wedge \overline{P^{(1)}} \Rightarrow (Q_2^{'} \wedge Inv_2^{'})] \wedge$$

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

In conformity testing, input constraints must be compatible with output constraints, and therefore the two constraints $[P^{(1)} \Rightarrow (Q_2^{'} \wedge Inv_2^{'})]$ and $[P_2^{'} \Rightarrow (Q^{(1)} \wedge Inv^{(1)})]$ should be eliminated from the constraint model $H^{(2)}$.

In principle, the overriding method $m^{(2)}$ must satisfy only two contracts of conformity:
– The contract of the superclass (Fig.9).
– The contract of its class (Fig.10).

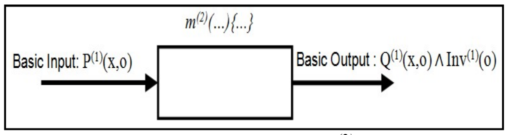

$m^{(2)}(...)\{...\}$
Basic Input: $P^{(1)}(x,o)$
Basic Output : $Q^{(1)}(x,o) \wedge Inv^{(1)}(o)$

Figure 9: Conformity Contract Of $m^{(2)}$ To Its Superclass

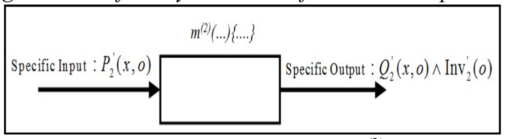

$m^{(2)}(...)\{....\}$
Specific Input : $P_2^{'}(x,o)$
Specific Output : $Q_2^{'}(x,o) \wedge Inv_2^{'}(o)$

Figure 10: Conformity Contract Of $m^{(2)}$ To Its Class

The second anomaly of the constraint model H is that some test data (Category 2 (Tab.1)) can be have simultaneously two responsibilities in the conformity assessment of overriding methods of inheritance process.

In the table 1, the test data (x,o) of the category 2 has two responsibilities in the conformity testing process : this (x,o) is used for conformity assessments of an overriding method $m^{(2)}$ according to $(P^{(1)},Q^{(1)},Inv^{(1)})$ and $(P_2^{'}, Q_2^{'}, Inv_2^{'})$.

Table 1: Responsibilities Of A Test Data In The Current Model H

(x,o)	*P(1)(x, o)*	*P'2(x,o)*	*Responsibility of (x,o)*
Category 1	*1*	*0*	$\longrightarrow$ *Conformity testing of $m^{(2)}$ to $(P^{(1)},Q^{(1)},Inv^{(1)})$*
Category 2	*1*	*1*	
Category 3	*0*	*1*	$\longrightarrow$ *Conformity testing of $m^{(2)}$ to (P'_2,Q'_2,Inv'_2)*
Category 4	*0*	*0*	$\longrightarrow$ *No responsibility*

In order to improve the conformity testing quality it is essetial that each test data (x,o) assumes only one responsibility. This means that all input data of the category 2 must be used for testing the conformity contract only according to the inherited specification $(P^{(1)},Q^{(1)},Inv^{(1)})$.

To satisfy this criteria, the constraint $[P_2^{'} \wedge P^{(1)} \Rightarrow Q_2^{'} \wedge Inv_2^{'}]$ should be eliminated from the constraint model H.

Definition (Constraint Model H_{op})
The optimal model of constraint H_{op} of $m^{(2)}$ is defined as follow :

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

- *$(P^{(1)},Q^{(1)}, Inv^{(1)})$ is the inherited specification of $m^{(2)}$.*
- *$(P_2^{'}, Q_2^{'}, Inv_2^{'})$ is the specific specification of $m^{(2)}$.*

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

4.2 Input values partitioning and test data generation

The equivalence classes partitionnig is an important technique for testing conformity contracts. In this technique, the input data domain of the program under test is divided into different equivalence classes.

• *Equivalence classes and conformity behaviors*

In our work, we use the optimal model of constraints H_{op} as a criterion to select input values, and to identify conformity behaviors (B_1,B_2,...,B_n): each equivalence class of the quotient set is used to represent a specific conformity behavior of overriding methods (Fig.11).

The equivalence classes partitioning can also be used to reduce the number of test data that must be generated in the conformity testing process of overriding methods.

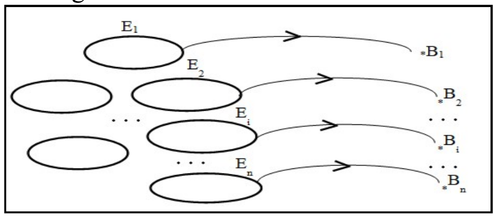


Figure 11: Equivalence Classes And Conformity Behaviors

The input domain $E{\times}I_{C2}$ of an overriding method $m^{(2)}$ is divided into 9 sets (Fig.12)

$$A=\{(x,o)\in E\times I_{c2}:H_{op}(x,o)\}\text{ and }B=\{(x,o)\in E\times I_{c2}:\overline{H_{op}(x,o)}\}$$

$$A_{op}=\{(x,o)\in E\times I_{c2}:(P^{(1)},P_2^{'},Q^{(1)},Inv^{(1)},Q_2^{'},Inv_2^{'})=(1,?,1,1,?,?)\}$$
$$A_{op}^{'}=\{(x,o)\in E\times I_{c2}:(P^{(1)},P_2^{'},Q^{(1)},Inv^{(1)},Q_2^{'},Inv_2^{'})=(0,1,?,?,1,1)\}$$
$$A_{op}^{''}=\{(x,o)\in E\times I_{c2}:(P^{(1)},P_2^{'},Q^{(1)},Inv^{(1)},Q_2^{'},Inv_2^{'})=(0,0,?,?,?,?)\}$$
$$B_{op}^{1}=\{(x,o)\in E\times I_{c2}:(P^{(1)},P_2^{'},Q^{(1)},Inv^{(1)},Q_2^{'},Inv_2^{'})=(1,?,1,0,?,?)\}$$
$$B_{op}^{2}=\{(x,o)\in E\times I_{c2}:(P^{(1)},P_2^{'},Q^{(1)},Inv^{(1)},Q_2^{'},Inv_2^{'})=(1,?,0,1,?,?)\}$$
$$B_{op}^{3}=\{(x,o)\in E\times I_{c2}:(P^{(1)},P_2^{'},Q^{(1)},Inv^{(1)},Q_2^{'},Inv_2^{'})=(1,?,0,0,?,?)\}$$
$$B_{op}^{'1}=\{(x,o)\in E\times I_{c2}:(P^{(1)},P_2^{'},Q^{(1)},Inv^{(1)},Q_2^{'},Inv_2^{'})=(0,1,?,?,1,0)\}$$
$$B_{op}^{'2}=\{(x,o)\in E\times I_{c2}:(P^{(1)},P_2^{'},Q^{(1)},Inv^{(1)},Q_2^{'},Inv_2^{'})=(0,1,?,?,0,1)\}$$
$$B_{op}^{'3}=\{(x,o)\in E\times I_{c2}:(P^{(1)},P_2^{'},Q^{(1)},Inv^{(1)},Q_2^{'},Inv_2^{'})=(0,1,?,?,0,0)\}$$

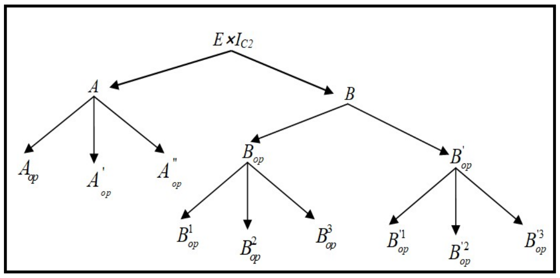


Figure 12: Tree Structure Of The Input Domain Partitioning

• *Algorithm of conformity testing*

The algorithm of conformity testing (Fig.13) is developed for generating input test data of overridden and overriding methods. This algorithm is based on the optimal model of constraints H_{op} and the equivalence partitioning

$(A_{op},A_{op}^{'},B_{op}^{1},B_{op}^{2},B_{op}^{3},B_{op}^{'1},B_{op}^{'2},B_{op}^{'3})$ for testing conformity behaviours of $m^{(2)}$ (The constant N is the test threshold limit) (Fig.12 and Fig.13).

The condition of the *do..while loop* can be exploited to deduce all conformity states of $m^{(2)}$ ($|A_{op}|$ is the cardinal number of the set A_{op}) :

$$\left.\begin{matrix}B_{op}^{1}\neq\varnothing\\ B_{op}^{2}\neq\varnothing\\ B_{op}^{3}\neq\varnothing\end{matrix}\right\}\Rightarrow\left[\exists(x,o):\overline{H_{op}(x,o)}\right]\Rightarrow m^{(2)}\text{ does not staisfy }(\mathrm{P}^{(1)},\mathrm{Q}^{(1)},\mathrm{Inv}^{(1)}).$$

$$\left.\begin{matrix}B_{op}^{'1}\neq\varnothing\\ B_{op}^{'2}\neq\varnothing\\ B_{op}^{'3}\neq\varnothing\end{matrix}\right\}\Rightarrow\left[\exists(x,o):\overline{H_{op}(x,o)}\right]\Rightarrow m^{(2)}\text{ does not staisfy }(\mathrm{P}_2^{'},\mathrm{Q}_2^{'},\mathrm{Inv}_2^{'}).$$

$$\left(\left|\mathrm{A}_{\mathrm{op}}\right|\geq N\wedge\left|\mathrm{A}_{op}^{'}\right|\geq N\right)\Rightarrow\forall(x,o)\in\{(x_1,\mathrm{o}_1),\ldots,(x_{2N},o_{2N}),\ldots\}:H_{op}(x,o)$$
$$\Rightarrow\mathrm{m}^{(2)}\text{ satisfies its conformity contracts.}$$

```
Aop=Ø; A'op=Ø; B1op=Ø; B2op=Ø; B3op=Ø; B'1op=Ø; B'2op=Ø; B'3op=Ø;
do{
  do{
  for ( xi in m(2) parameter's)
   {xi = generate(Ei);}
    x= (x1,x2,…,xn) ;
    o = generate_object();
    }while(!P(1)(x,o)&& !P2'(x,o));
  invoke"o.m(2)(x)"
  if(P(1)(x,o)){
     if(Q(1)(x,o)&& Inv(1)(o))
         Aop.add(x,o);
     elseif( Q(1)(x,o)&& !Inv(1)(o))
         B1op.add(x,o);
     elseif(!Q(1)(x,o)&& Inv(1)(o))
         B2op.add(x,o);
     else
         B3op.add(x,o);}
   else {
     if(Q'2 (x,o)&& Inv'2 (o))
         A'op.add(x,o);
     elseif( Q2'(x,o)&& !Inv'2 (o))
         B'1op.add(x,o);
     elseif(!Q2'(x,o)&& Inv'2 (o))
         B'2op.add(x,o);
     else
         B'3op.add(x,o);}
}while((Aop.size()<N || A'op.size()<N) && B1op.isEmpty()&& B2op.isEmpty()
&& B3op.isEmpty()&& B'1op.isEmpty()&& B'2op.isEmpty()&& B'3op.isEmpty());
```

Figure 13: Algorithm Of Conformity Testing

5. APPROACH OF ROBUSTNESS TESTING BY THE OPTIMAL MODEL OF CONSTRAINTS

The robustness approach of [14] can be used to test robustness contracts of subclasses from test result of super classes. This reusability of test sequences is only possible if overriding and overridden methods are similar (Fig.8). In this section we present an approach of robustness for testing the robustness of overriding methods in subclasses in the general case where $m^{(2)}$ and $m^{(1)}$ are not necessarily similar.

5.1 Principle and constraints of robustness

testing

In [14], we have proposed an approach of robustness testing by similarity of behaviours for OO classes. The robustness testing is based on input data which don't satisfy the precondition constraints (*P(DT)=0*). In our approach, an invalid input data must induce only invalid output constraints (Fig.14), and is not an undefined data: an invalid data is a data for which *P*, *Q*, and *Inv* are well defined (a test data *DT* that induces a division by *0* is an undefined data and must be ignored by the system of testing…).

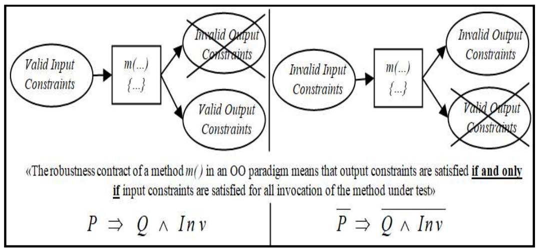

Valid Input Constraints
m(...) {...}
Invalid Output Constraints
Valid Output Constraints
Invalid Input Constraints
m(...) {...}
Invalid Output Constraints
Valid Output Constraints
«The robustness contract of a method m( ) in an OO paradigm means that output constraints are satisfied if and only if input constraints are satisfied for all invocation of the method under test»
$P \Rightarrow Q \wedge Inv$
$\overline{P} \Rightarrow \overline{Q \wedge Inv}$

Figure 14: Principle And Constraints Of Robustness Testing

A method *m(…){…}* of an OO class is robust if the constraint $H_{robustness}$ is satisfied (Fig.15).

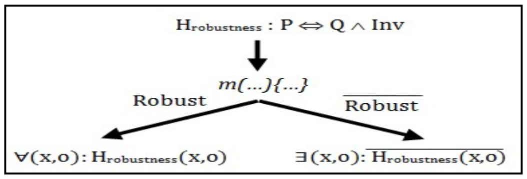

$H_{robustness} : P \Leftrightarrow Q \wedge Inv$
m(...){...}
Robust
$\overline{\text{Robust}}$
$\forall(x,o): H_{robustness}(x,o)$
$\exists(x,o): \overline{H_{robustness}(x,o)}$

Figure 15: Constraint Model Of Robustness Testing

5.2 Optimal model of constraints and robustness verification of inheritance

The optimal model of constraint H_{op} can be used for modeling the conformity contract of overriding methods in subclasses of the inheritance mechanism. In this section, we present an optimal model of robustness constraints to verify not only the conformity property but also the robustness contract of OO programs.

- *Optimal model of constraints* H_{op}^{rob}

The optimal model of robustness constraints H_{op}^{rob} (***optimal robustness***) can be constructed from the constraint model $H_{robustness}$ (Fig.15) by adding specific constraints of subclasses (Fig.16). This model must be also a strengthening of the conformity model H_{op} by using complementary constraints for modeling not only the conformity assessment but also the robustness contract.

Definition

The optimal model of robustness H_{op}^{rob} *of* $m^{(2)}$ *is defined as follows:*

$$H_{op}^{rob} : [P^{(1)} \Rightarrow (Q^{(1)} \wedge Inv^{(1)})] \wedge [\overline{(P^{(1)} \vee P_2')} \Rightarrow \overline{(Q^{(1)} \wedge Inv^{(1)})}] \wedge [(P_2' \wedge \overline{P^{(1)}}) \Rightarrow (Q_2' \wedge Inv_2')] \wedge [\overline{(P^{(1)} \vee P_2')} \Rightarrow \overline{(Q_2' \wedge Inv_2')}]$$

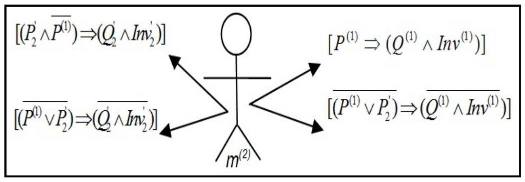

$[(P_2' \wedge \overline{P^{(1)}}) \Rightarrow (Q_2' \wedge Inv_2')]$
$[P^{(1)} \Rightarrow (Q^{(1)} \wedge Inv^{(1)})]$
$[\overline{(P^{(1)} \vee P_2')} \Rightarrow \overline{(Q_2' \wedge Inv_2')}]$
$[\overline{(P^{(1)} \vee P_2')} \Rightarrow \overline{(Q^{(1)} \wedge Inv^{(1)})}]$
$m^{(2)}$

Figure 16: Constraints Of The Robustness Optimal Model

- *Robustness contracts of overriding methods*

The robustness optimal model H_{op}^{rob} is an important mathematical entity for modeling the robustness in subclasses.

Corollary

- *$m^{(2)}$ is robust according to its specification if :*

$$\forall(x,o) \in E \times I_{C2} : H_{op}^{rob}(x,o)$$

- *$m^{(2)}$ is not robust according to its specification if :* $\exists(x,o) \in E \times I_{C2} : \overline{H_{op}^{rob}(x,o)}$

Definition

- *$m^{(2)}$ is robust according to $(P^{(1)}, Q^{(1)}, Inv^{(1)})$ if :*

$$\forall(x,o) \in E \times I_{C2} : [P^{(1)}(x,o) \Rightarrow (Q^{(1)}(x,o) \wedge Inv^{(1)}(o))] \wedge [\overline{(P^{(1)} \vee P_2')(x,o)} \Rightarrow \overline{(Q^{(1)}(x,o) \wedge Inv^{(1)}(o))}]$$

- *$m^{(2)}$ is robust* according *to* (P_2', Q_2', Inv_2') *if :*

$$\forall(x,o) \in E \times I_{C2} : [(P_2'(x,o) \wedge \overline{P^{(1)}(x,o)}) \Rightarrow (Q_2'(x,o) \wedge Inv_2'(o))] \wedge [\overline{(P^{(1)} \vee P_2')(x,o)} \Rightarrow \overline{(Q_2'(x,o) \wedge Inv_2'(o))}]$$

Corollary

- *$m^{(2)}$ is not robust to $(P^{(1)}, Q^{(1)}, Inv^{(1)})$ if :*

$$\exists(x,o) \in E \times I_{C2} : [\overline{(P^{(1)} \vee P_2')(x,o)}] \wedge [(Q^{(1)}(x,o) \wedge Inv^{(1)}(o))]$$

- *$m^{(2)}$ is not robust to (P_2', Q_2', Inv_2') if :*

$$\exists(x,o) \in E \times I_{C2} : [\overline{(P^{(1)} \vee P_2')(x,o)}] \wedge [(Q_2'(x,o) \wedge Inv_2'(o))]$$

5.3 Input data partitioning and robustness behaviours

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

- *Input data partitioning*

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

Our input data partitioning is based on the optimal model H_{op}^{rob} to generate all classes of test data of overriding methods *m*$^{(2)}$. This partitioning where the precondition is in the logical state False gives all possible behaviors of robustness contracts (Fig.17 and Fig.18):

$$A_{op}^{"} = \{(x,o) \in E \times I_{c2} : (P^{(1)}, P_2^{'}, Q^{(1)}, Inv^{(1)}, Q_2^{'}, Inv_2^{'}) = (0,0,?,?,?,?)\}$$

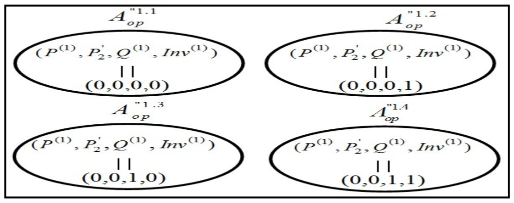

$A_{op}^{"1.1}$
$(P^{(1)}, P_2^{'}, Q^{(1)}, Inv^{(1)})$
$(0,0,0,0)$
$A_{op}^{"1.2}$
$(P^{(1)}, P_2^{'}, Q^{(1)}, Inv^{(1)})$
$(0,0,0,1)$
$A_{op}^{"1.3}$
$(P^{(1)}, P_2^{'}, Q^{(1)}, Inv^{(1)})$
$(0,0,1,0)$
$A_{op}^{"1.4}$
$(P^{(1)}, P_2^{'}, Q^{(1)}, Inv^{(1)})$
$(0,0,1,1)$

Figure 17: Input Data Partitioning Of m$^{(2)}$ According To $(P^{(1)}, Q^{(1)}, Inv^{(1)})$

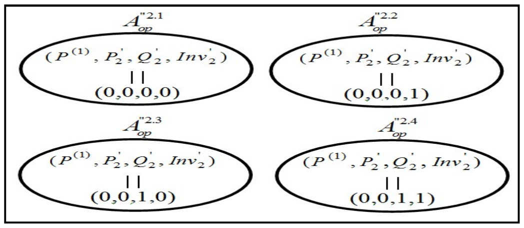
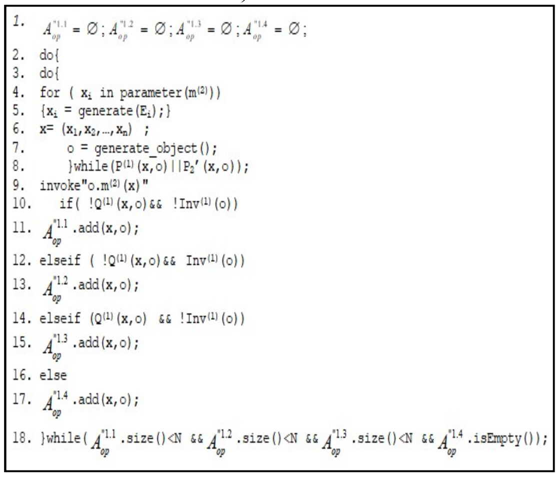

$A_{op}^{"2.1}$
$(P^{(1)}, P_2^{'}, Q_2^{'}, Inv_2^{'})$
$(0,0,0,0)$
$A_{op}^{"2.2}$
$(P^{(1)}, P_2^{'}, Q_2^{'}, Inv_2^{'})$
$(0,0,0,1)$
$A_{op}^{"2.3}$
$(P^{(1)}, P_2^{'}, Q_2^{'}, Inv_2^{'})$
$(0,0,1,0)$
$A_{op}^{"2.4}$
$(P^{(1)}, P_2^{'}, Q_2^{'}, Inv_2^{'})$
$(0,0,1,1)$

Figure 18: Input Data Partitioning Of m$^{(2)}$ According To $(P_2^{'}, Q_2^{'}, Inv_2^{'})$

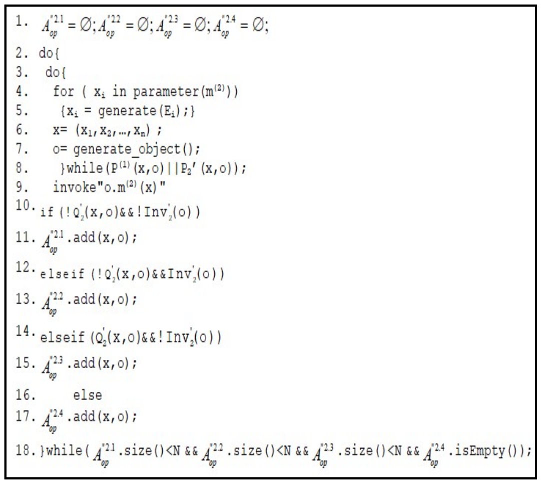
• *Robustness testing of overriding methods*

The algorithm of robustness testing (Fig.19 and Fig.20) is developed for generating input test data of overriding methods. This algorithm is based on the optimal model of constraints and the domain partitioning $(A_{op}^{"1.1}, A_{op}^{"1.2}, A_{op}^{"1.3}, A_{op}^{"1.4}, A_{op}^{"2.1}, A_{op}^{"2.2}, A_{op}^{"2.3}, A_{op}^{"2.4})$ for testing robustness behaviours of *m*$^{(2)}$ (The constant *N* is the test threshold limit).



```
$A_{op}^{"1.1} = \varnothing; A_{op}^{"1.2} = \varnothing; A_{op}^{"1.3} = \varnothing; A_{op}^{"1.4} = \varnothing;$
do{
do{
for ( $x_i$ in parameter($m^{(2)}$))
{$x_i$ = generate($E_i$);}
x= ($x_1, x_2, …, x_n$) ;
    o = generate_object();
    }while($P^{(1)}$(x,o)||$P_2'$(x,o));
invoke"o.$m^{(2)}$(x)"
   if( !$Q^{(1)}$(x,o)&& !$Inv^{(1)}$(o))
$A_{op}^{"1.1}$.add(x,o);
elseif ( !$Q^{(1)}$(x,o)&& $Inv^{(1)}$(o))
$A_{op}^{"1.2}$.add(x,o);
elseif ($Q^{(1)}$(x,o) && !$Inv^{(1)}$(o))
$A_{op}^{"1.3}$.add(x,o);
else
$A_{op}^{"1.4}$.add(x,o);
}while( $A_{op}^{"1.1}$.size()<N && $A_{op}^{"1.2}$.size()<N && $A_{op}^{"1.3}$.size()<N && $A_{op}^{"1.4}$.isEmpty());
```

Figure 19: Algorithm Of Robustness Testing To $(P^{(1)}, Q^{(1)}, Inv^{(1)})$

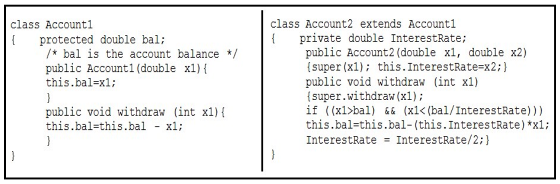


```
$A_{op}^{"2.1} = \varnothing; A_{op}^{"2.2} = \varnothing; A_{op}^{"2.3} = \varnothing; A_{op}^{"2.4} = \varnothing;$
do{
 do{
  for ( $x_i$ in parameter($m^{(2)}$))
   {$x_i$ = generate($E_i$);}
  x= ($x_1, x_2, …, x_n$) ;
  o= generate_object();
   }while($P^{(1)}$(x,o)||$P_2'$(x,o));
  invoke"o.$m^{(2)}$(x)"
if (!$Q_2'$(x,o)&&!$Inv_2'$(o))
$A_{op}^{"2.1}$.add(x,o);
elseif (!$Q_2'$(x,o)&&$Inv_2'$(o))
$A_{op}^{"2.2}$.add(x,o);
elseif ($Q_2'$(x,o)&&!$Inv_2'$(o))
$A_{op}^{"2.3}$.add(x,o);
     else
$A_{op}^{"2.4}$.add(x,o);
}while( $A_{op}^{"2.1}$.size()<N && $A_{op}^{"2.2}$.size()<N && $A_{op}^{"2.3}$.size()<N && $A_{op}^{"2.4}$.isEmpty());
```

Figure 20: Algorithm Of Robustness Testing To $(P_2^{'}, Q_2^{'}, Inv_2^{'})$

6. EVALUATION

In this section we present an example of test data generation of the overridden method *withdraw*$^{(1)}$ and the overriding method *withdraw*$^{(2)}$ for two java classes (Fig.21).

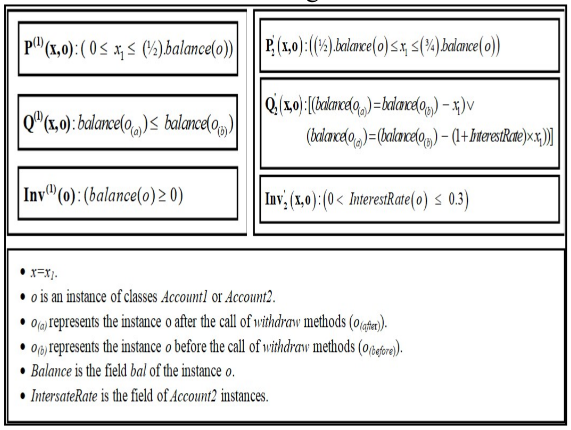


```
class Account1
{    protected double bal;
      /* bal is the account balance */
      public Account1(double x1){
      this.bal=x1;
      }
      public void withdraw (int x1){
      this.bal=this.bal - x1;
      }
}
```

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

Figure 21: Java Implementation Of withdraw Methods

In the figure 22 we present specifications of the overridden and the overriding methods *withdraw*:



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

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

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

$\mathbf{P_2^{'}(x,o)}: ((½).balance(o) \le x_1 \le (¾).balance(o))$

$\mathbf{Q_2^{'}(x,o)}: [(balance(o_{(a)}) = balance(o_{(b)}) - x_1) \vee (balance(o_{(a)}) = (balance(o_{(b)}) - (1+InterestRate) \times x_1))]$

$\mathbf{Inv_2^{'}(x,o)}: (0 < InterestRate(o) \le 0.3)$

- *x=x₁*.
- *o* is an instance of classes *Account1* or *Account2*.
- $o_{(a)}$ represents the instance o after the call of *withdraw* methods ($o_{(after)}$).
- $o_{(b)}$ represents the instance *o* before the call of *withdraw* methods ($o_{(before)}$).
- *Balance* is the field *bal* of the instance *o*.
- *IntersateRate* is the field of *Account2* instances.

Figure 22: Specification (P$^{(1)}$,Q$^{(1)}$,Inv$^{(1)}$) And (P'$_2$,Q'$_2$,Inv'$_2$) Of Methods (withdraw$^{(1)}$,withdraw$^{(2)}$)

6.1 Test data generation of conformity constraints

The table 2 illustrates an example of test data generation of the method *withdraw*$^{(2)}$ (x_1 and *balance(o)* are in *]-200,200[*; The threshold limit *N=100*).

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

Table 2: Result Of A Conformity Test Of Withdraw$^{(2)}$

	x_1	*O*	$P^{(1)}$	P'_2	$\lvert A_{op}\rvert$	$\lvert A'_{op}\rvert$	H_{op}
1	68	Account2(185,0.23)	1	0	1	0	1
2	80	Account2(167,0.18)	1	0	2	0	1
3	40	Account2(114,0.22)	1	0	3	0	1
4	91	Account2(126,0.12)	0	1	3	1	1
…	…	…	…	…	…	…	…
71	130	Account2(197,0.25)	0	1	47	24	1
72	33	Account2(88,0.27)	1	0	48	24	1
73	70	Account2(103,0.13)	0	1	48	25	1
…	…	…	…	…	…	…	…
99	49	Account2(150,0.16)	1	0	69	30	1
100	38	Account2(99,0.12)	1	0	70	30	1
101	58	Account2(142,0.07)	1	0	71	30	1
…	…	…	…	…	…	…	…
136	104	Account2(176,0.13)	0	1	99	37	1
137	19	Account2(47,0.17)	1	0	100	37	1
138	27	Account2(111,0.12)	1	0	101	37	1
…	…	…	…	…	…	…	…
189	14	Account2(77,0.08)	1	0	123	66	1
190	119	Account2(191,0.04)	0	1	123	67	1
191	73	Account2(189,0.19)	1	0	124	67	1
…	…	…	…	…	…	…	…
199	86	Account2(153,0.14)	0	1	126	73	1
200	12	Account2(35,0.28)	1	0	127	73	1
201	52	Account2(78,0.05)	0	1	127	74	1
…	…	…	…	…	…	…	…
240	81	Account2(164,0.21)	1	0	142	98	1
241	90	Account2(130,0.24)	0	1	142	99	1
242	57	Account2(129,0.15)	1	0	143	99	1
243	111	Account2(182,0.10)	0	1	143	100	1

For *(2.N+i)* iterations *(N=100,i=43)* of the *do…while loop* of the algorithm of conformity testing (Fig.13), the optimal model is always satisfied ($H_{op}=1$). Therefore, the method *withdraw*$^{(2)}$ meets its conformity contract.

6.2 Test data generation of robustness constraints

Our approach is used to test the robustness contract even if similarity is not satisfied, the table 3 illustrates an example of test data generation of the overriding method *withdraw*$^{(2)}$(x_1 and *balance(o)* are in *]-200,200[*; The threshold limit *N=100*).

Table 3: Result Of A Robustness Test Of withdraw$^{(2)}$

	x_1	*O*	$(P^{(1)} \vee P'_2)(x_1,o)$	$H^{rob}_{op}(x_1,o)$
1	173	Account2(146,0.18)	0	1
2	118	Account2(99,0.23)	0	1
3	121	Account2(112,0.29)	0	1
…	…	…	…	…
20	110	Account2(103,0.15)	0	1
21	88	Account2(77,0.06)	0	1
22	159	Account2(114,0.22)	0	1
…	…	…	…	…
45	101	Account2(89,0.16)	0	1
46	131	Account2(91,0.25)	0	1
47	176	Account2(187,0.11)	0	0

In the iteration number *47* of the *do…while loop* of the algorithm of robustness testing (Fig.19 and Fig.20), the optimal model is not satisfied ($H^{rob}_{op}=0$). Therefore, the method *withdraw*$^{(2)}$ does not meet its robustness contract.

7. CONCLUSION

We have proposed in this paper an optimal model of constraints to validate conformity and robustness contracts between OO programs and their specifications. Our approach is based on some mathematical entities (set theory and logical axioms) to represent conformity and robustness behaviors, and therefore to generate test data of OO classes. Our work is an important way to verify conformity and robustness behaviours of subclasses of inheritance mechanism.

The first approach of this work is an algorithm of test data generation based on input data partitioning for testing the conformity contract of an OO model. The second approach is a way to generate test data of robustness by using the invalid input data partitioning. This paper shows how the equivalence partitioning can be used to reduce the test data generation and therefore, to improve software testing.

References

[1] B. Randell, *The origins of digital computers*. Berlin: Springer-Verlag, 1973.

[2] R.W. Floyd, “Assigning Meanings to Programs”, *In: Program Verification, Studies in Cognitive Systems*, vol 14. Springer,1993, pp.65-81.

[3] C. Hoare, “An axiomatic basis for computer programming”, *Communications of the ACM*, vol. 12, no. 10, pp. 576-580, 1969.

[4] P. Naur, “Proof of algorithms by general snapshots”, *BIT*, vol. 6, no. 4, pp. 310-316, 1966.

[5] E. Dijkstra, “Guarded commands, nondeterminacy and formal derivation of programs”, *Communications of the ACM*, vol. 18, no. 8, pp. 453-457, 1975.

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

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

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

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

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

[11] K. Louzaoui, K. Benlhachmi, J.A. Chentoufi, "Conformity testing by optimal constraints for object oriented programs" *In Information Science and Technology* (CiSt),IEEE, Tangier, Morocco, 2016. pp. 21-29.

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

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

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

[15] F. Bouquet, F. Dadeau, B. Legeard, M. Utting, "Symbolic Animation of JML Specifications". *In: International Symposium on Formal Methods*, Springer, Berlin, Heidelberg, 2005, pp. 75-90.

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

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

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

[19] W. Huang and J. Peleska, "Complete model-based equivalence class testing", *International Journal on Software Tools for Technology Transfer*, vol. 18, no. 3, pp. 265-283, 2014.

[20] A. Vallecillo, M. Gogolla, L. Burgueño, M. Wimmer, L. Hamann, "Formal Specification and Testing of Model Transformations", *In : International School on Formal Methods for the Design of Computer, Communication and Software Systems*, Springer, Berlin, Heidelberg, 2012, pp. 399-437.

[21] T. Thüm, I. Schaefer, M. Kuhlemann, S. Apel, G. Saake, "Applying Design by Contract to Feature-Oriented Programming", *In : International Conference on Fundamental Approaches to Software Engineering*, Springer, Berlin, Heidelberg, 2012, pp. 255-269.

[22] K. Karoui, , F. B. Ftima, H. B. Ghezala, "Formal specification, verification and correction of security policies based on the decision tree approach". *International Journal of Data & Network Security*, vol. 3, no 3, pp. 92-111, 2013.

[23] M-C. Gaudel and P. L. Gall, "Testing Data Types Implementations from Algebraic Specifications", *In Formal Methods and Testing*, Springer, Berlin, Heidelberg, 2008, pp. 209–239.

[24] J. Ferrer, F. Chicano and E. Alba, "Evolutionary algorithms for the multi-objective test data generation problem", *Software: Practice and Experience*, vol. 42, no. 11, pp. 1331-1362, 2011.

[25] S. Ali, L. Briand and H. Hemmati, "Modeling robustness behavior using aspect-oriented modeling to support robustness testing of industrial systems", *Software & Systems Modeling*, vol. 11, no. 4, pp. 633-670, 2011.

[26] JC. Fernandez, L.Mounier, C. Pachon. "A Model-Based Approach for Robustness Testing". *In : IFIP International Conference on Testing of Communicating Systems.*

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

Springer, Berlin, Heidelberg, 2005, pp. 333-348.

[27] S. Anand et al., "An orchestrated survey of methodologies for automated software test case generation", *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978-2001, 2013.

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

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

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

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

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