Génération de Données de Test de Conformité à partir d'une Spécification Formelle par Analyse des Partitions et Classification

Khalid Benlhachmi, Mohammed Benattou
Laboratoire LARIT
Université Ibn Tofail, Faculté des Sciences
Kénitra, Maroc

Jean-louis Lanet
Laboratoire XLIM/SDD
Université de Limoges
Limoges, France

Résumé —**Les travaux présentés dans cet article proposent un modèle formel de contrainte généralisée permettant de tester la conformité d'une implémentation par rapport à sa spécification. L'idée principale de notre approche est basée sur une classification par analyse de partition des domaines de conformité et de non-conformité pour chaque type de méthode dans le cadre de la programmation objet. L'oracle de test proposé utilise une génération aléatoire de données et permet de déduire l'origine de l'anomalie (invariant, post-condition, pré-condition).**

Mots clés ; Spécification formelle; test de conformité ;données valides ; données invalides ; génération de données de test

I. INTRODUCTION

Les principales approches pour la vérification des logiciels sont la preuve et le test. Les méthodes de vérifications basées sur des preuves mathématiques permettent de garantir l'absence de certaines classes d'erreurs mais elles sont très couteuses et difficiles à mettre en œuvre dans l'industrie. Le test permet de stimuler le comportement d'un logiciel en lui appliquant des entrées et en vérifiant les sorties par rapport aux spécifications attendues. Dans ce contexte, les méthodes de test à partir d'une spécification formelle sont devenues les bases de plusieurs travaux et d'oracles de test. Dans le cadre de la programmation objet, OCL [1] et JML [2] sont devenus des standards pour la spécification formelle permettant de décrire et d'énoncer des contraintes sur les états d'un système (invariants), de décrire l'effet d'une opération par un couple de pré-conditions et post-conditions, et des expressions de navigation dans un diagramme de classe.

La génération de données de test à partir d'une spécification formelle utilise soit des méthodes de génération de données aléatoire pour tester la conformité d'une classe d'objet par rapport à sa spécification, soit des méthodes de résolutions de contraintes pour réduire le domaine de valeurs des données générées.

Notre approche propose un modèle formel généralisé permettant de tester la conformité de l'implémentation d'une méthode par rapport à sa spécification. L'idée principale de notre approche est basée sur une classification par analyse de partition des domaines de conformité et de non-conformité pour chaque type de méthode. L'oracle de test proposé utilise une génération aléatoire de données et permet de déduire l'origine de la non-conformité ou de l'anomalie (invariant, post-condition, pré-condition).

L'idée de base de ces travaux est la proposition d'un modèle formel généralisant les contraintes classiques d'une méthode (Pré-condition, Post-condition et Invariant). Nous utilisons par la suite ce modèle pour l'analyse et la construction des partitions des domaines d'entrée de la méthode à analyser. Ces partitions nous permettent de mettre en œuvre des méthodes de génération des données de test de conformité pour des entrées valides. Ce papier est organisé comme suit : la section 2 présente un état de l'art sur la génération de données de test à partir d'une spécification formelle, la section 3 décrit notre modèle formel de contrainte, la section 4 présente notre méthode de classification par analyse de partition, la section 5 est dédiée à la génération de données de test de conformité.

II. ÉTAT DE L'ART

La génération de données de test à partir d'une spécification formelle utilise soit des méthodes de génération de données aléatoires pour tester la conformité d'une classe d'objets par rapport à sa spécification, soit des méthodes de résolutions de contraintes pour réduire le domaine de valeurs des données générées. Dans ce contexte, plusieurs travaux se sont orientés vers la génération de données de test à partir d'une spécification formelle écrite en OCL [1] et JML [2] qui se présentent comme des standards de la spécification formelle dans le cadre de la programmation objet.

Dans [3], ils proposent de générer des données de test aléatoire à partir d'une spécification formelle JML d'une classe d'objets. Ils classifient les méthodes et les constructeurs suivant leurs signatures (basique, constructeur étendu, mutateur, observateur) et pour chaque type de méthode une génération de donnée de test est proposé. Dans [4], ils réduisent le domaine de génération de données de test pour les types de données dont le domaine est fini et une génération aléatoire pour les autres types comme les objets en utilisant la

résolution de contraintes. Dans [5], ils proposent de tester la conformité des méthodes en se basant sur la génération aléatoire des données de test par utilisation des preuves expérimentales permettant l'étude de l'efficacité et de la performance des techniques de tests aléatoires . Dans [6], ils réalisent un outil appelé JCrasher pour le test aléatoire des classes Java. Dans [7], ils proposent un outil paramétrable Jartege pour la génération des tests unitaires pour des classes Java spécifiées en JML. Dans [1], les auteurs présentent une méthode basée sur la résolution des contraintes pour la génération des cas de test avec anticipation d'erreur dans la spécification des méthodes. Dans [8], les auteurs décrivent une méthode et un outil combinant la résolution de contraintes et l'exécution symbolique de programmes. L'outil permet de trouver des valeurs possibles pour les variables à contraintes en parcourant l'automate fini généré à partir d'un programme et sa spécification formelle. Dans [9], les auteurs présentent un oracle de test automatisé des programmes Java appelé Korat. Ils utilisent la pré-condition de la spécification d'une méthode pour générer automatiquement les cas de test et réduire la taille des domaines d'entrées.

Le travail proposé dans ce papier permet par rapport aux travaux cités ci-dessus de définir un modèle théorique évolutif et paramétrable selon le type de test demandé. Cela induit un oracle de test qui analyse la conformité de l'implémentation d'une méthode par rapport à sa spécification

III. MODELE FORMEL DE CONTRAINTE

Dans ce paragraphe, nous donnons la définition d'une nouvelle contrainte, illustrant, d'une part la relation entre la pré-condition et la post-condition d'une méthode et d'autre part montrant l'importance de l'invariant en tant que condition nécessaire pour la vérité de cette contrainte. Cette contrainte regroupe la pré-condition, la post-condition, et l'invariant en une seule formule logique. Elle devra traduire algébriquement le contrat entre l'utilisateur (le programme appelant) et la méthode appelée.

En effet, nous supposons dans le cadre de ce travail que nous ne disposons pour une méthode que de sa spécification formelle liant ses entrées à la sortie attendue. Ces résultats seront appliqués en premier sur les constructeurs et les méthodes statiques qui créent des instances. Nous analysons par la suite les cas où l'état de l'objet sera modifié à l'intérieur de la méthode, et finalement les cas où cet état demeure stable.

Soit C une classe, m une méthode de n arguments $x=(x_1, x_2,\ldots, x_n)$. On définit pour chaque argument x_i son domaine de valeurs E_i. On pose $E=E_1\times E_2\times\ldots E_n$ avec $E= \{(x_1, x_2,\ldots, x_n) / x_1\in E_1 \text{ et } x_2 \in E_2 \text{ et } \ldots \text{et } x_n \in E_n \}$ le domaine des vecteurs d'entrée de la méthode m.

On note dans ce papier : P la pré-condition de la méthode m, Q la post-condition de la méthode m et Inv l'invariant de la classe C. L'idée principale de cette relation est basée sur l'interprétation suivante :

Si l'utilisateur respecte sa part du contrat en invoquant une méthode par des arguments vérifiant la pré-condition P, alors la méthode doit respecter nécessairement la post-condition Q après l'appel.

Quant à l'invariant, il doit être vérifié avant et après l'appel de la méthode.

Définition1 On définit la contrainte généralisée H d'une méthode m d'une classe C définie sur un domaine d'entrée E comme étant une propriété du vecteur d'entrée x et de l'objet récepteur o tel que :

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

Avec I_c l'ensemble d'instances de la classe C.

En effet, l'appel d'une méthode m se fait généralement par référence à un objet o et par conséquent m est identifiée par le couple (x,o) avec x le vecteur de paramètres et o l'objet appelant. On note $att_1, att_2,\ldots, att_m$ les attributs de la classe C et $state$ l'état d'un objet o de C défini par l'ensemble des valeurs de ses attributs :

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

L'objet est une entité qui peut changer son état suite à une modification d'une valeur de ses attributs. Nous nous intéressons à l'état de l'objet o avant l'appel de la méthode (*State before*) et après l'appel (*State after*).

On note $State\ (before(o)) = b = (b_1,b_2 ,\ldots,b_m)$ et $State\ (after(o)) = a = (a_1 ,a_2 ,\ldots,a_m)$ avec respectivement b_i et a_i les valeurs de chaque attribut att_i avant et après l'appel de la méthode. En effet, on caractérise chaque objet par sa référence o et son état b et a :

$$o_{(b)}= (o, b_1,b_2 ,\ldots,b_m)\ o_{(a)}= (o, a_1 ,a_2 ,\ldots,a_m)$$

Vu le changement de l'état d'un objet de l'état avant vers l'état après l'appel d'une méthode, la contrainte définie ci-dessus doit prendre en compte l'état de l'objet avant et après l'appel. Par conséquent, la contrainte doit indiquer le lien entre l'invariant de l'objet à l'entrée $o_{(b)}$ et l'invariant à la sortie $o_{(a)}$:

$$H(x,o):[P(x,o)\wedge Inv(o_{(b)})]\Rightarrow [Q(x,o)\wedge Inv\ (o_{(a)})],(x,o) \in E \text{ x } I_c$$

L'implication logique dans la formule se traduit par : tout appel à la méthode avec (x, o) vérifiant la pré-condition P et l'invariant Inv avant l'appel, (x,o) doit nécessairement vérifier la post-condition Q et l'invariant après l'appel. Dans le contexte du paradigme de la programmation objet la contrainte peut être réduite suite à la déduction suivante :

La vérification de l'invariant, pour chaque instance o de la classe à la sortie de son constructeur et à la sortie de toutes les méthodes précédant la méthode m étudiée, est suffisante pour le maintenir valide à l'entrée de m : i.e., tout appel d'un objet o à une méthode m ne peut se faire qu'avec un objet valide vérifiant l'invariant (instancié par un constructeur valide).

L'objet récepteur o vérifie toujours l'invariant à l'entrée de la méthode, et par conséquent l'invariant n'est vérifié dans notre contrainte qu'après l'appel.

On en déduit que l'invariant de l'objet o à l'état avant est vérifié : $Inv\ (o_{(b)}) = 1$

Par conséquent :

$$[\ P(x,o) \wedge Inv\ (o_{(b)})] \Rightarrow [Q(x, o) \wedge Inv\ (o_{(a)})] \Leftrightarrow$$
$$[\ P(x,o) \wedge 1\ ] \Rightarrow [Q(x, o) \wedge Inv\ (o_{(a)})]$$

Il en résulte de cette déduction une contrainte simplifiée :

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

$o_{(a)}$ sera noté simplement o où l'invariant est évalué à la sortie de la méthode. L'évaluation de la contrainte H pour un $(x,o) \in E \times I_c$ est effectuée en deux étapes :

- À l'entrée de la méthode évaluation de $P(x,o)$
- À la sortie de la méthode évaluation de $Q(x,o)$ et $Inv\ (o_{(a)})$ (noté $Inv\ (o)$)

IV. ANALYSE DES PARTITIONS

Nous nous intéressons dans ce paragraphe à la partition générale du domaine d'entrée d'une méthode pour la génération aléatoire des données de test. Nous montrons par la suite que ces domaines pourront être réduits si on tient en compte les classifications des méthodes définies dans [3].

A. Analyses générales des partitions

L'objectif de ce sous paragraphe est l'analyse générale de partition du domaine d'entrée d'une méthode d'une classe d'objet. En effet, on divise le domaine d'entrée $E \times I_c$ de la méthode en deux parties A et B :

$A = \{(x,o) \in E \times I_c / H(x,o)=1\}$ et $B = \{(x,o) \in E \times I_c / H(x,o)=0\}$

Par la suite, On divise A en deux sous ensemble A_1 et A_2 :

$A_1 = \{(x,o) \in E \times I_c / (P(x,o),Q(x,o),Inv(o))=(1,1,1)\}$: le domaine dont les éléments (x,o) vérifient P,Q, et Inv .

$A_2 = \{(x,o) \in E \times I_c / (P(x,o),Q(x,o),Inv(o))=(0, ?,?)\}$, ce domaine représente les couples (x,o) de $E \times I_c$ tels que la pré-condition P de la méthode n'est pas vérifiée.

Nous répartissons A_2 suivant les cas où P est fausse, et l'invariant Inv et la post-condition Q sont quelconques. Cela induit quatre domaines A_{21} , A_{22} , A_{23} et A_{24} tels que :

- $A_{21} = \{(x,o) \in E \times I_c / (P(x,o),Q(x,o),Inv(o))=(0,0,0)\}$: Les couples (x,o) de A_{21} ne vérifient ni P ni Q ni Inv.
- $A_{22} = \{(x,o) \in E \times I_c / (P(x,o),Q(x,o),Inv(o))=(0,0,1)\}$: Les couples (x,o) de A_{22} vérifient Inv, et ne vérifient ni P ni Q.
- $A_{23} = \{(x,o) \in E \times I_c / (P(x,o),Q(x,o),Inv(o))=(0,1,0)\}$: Les couples (x,o) de A_{23} vérifient Q et ne vérifient ni P ni Inv.
- $A_{24} = \{(x,o) \in E \times I_c / (P(x,o),Q(x,o),Inv(o))=(0,1,1)\}$: Les couples (x,o) de A_{24} vérifient Inv,Q et ne vérifient pas P.

On a: $A = A_1 \cup A_2$ et $A_2 = A_{21} \cup A_{22} \cup A_{23} \cup A_{24}$

Ce qui donne $A = A_1 \cup A_{21} \cup A_{22} \cup A_{23} \cup A_{24}$

Les couples (x,o) de A_1 , A_{21} , A_{22} , A_{23} et A_{24} vérifient la contrainte H (Fig. 1).

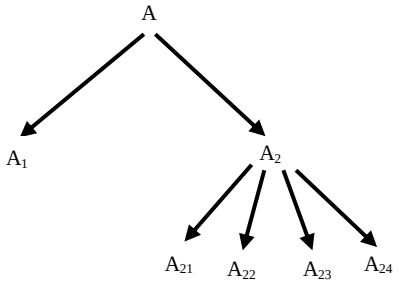

Figure 1. Arborescence de la partition de A

B est définie par $B = \neg A = \{(x,o) \in E \times I_c / H(x,o)=0\}$. On en déduit les trois partitions suivantes de B :

$B_1 = \{(x,o) \in E \times I_c / (P(x,o),Q(x,o),Inv(o))=(1,1,0)\}$. Ce domaine correspond aux couples (x,o) de $E \times I_c$ qui vérifient P et Q et ne vérifient pas Inv .

$B_2 = \{(x,o) \in E \times I_c / (P(x,o),Q(x,o),Inv(o))=(1,0,1)\}$. Ce domaine représente les couples (x,o) de $E \times I_c$ tels que Q n'est pas vérifié ,et P et Inv sont vérifiés.

$B_3 = \{(x,o) \in E \times Ic / (P(x,o),Q(x, o),Inv(o))=(1,0,0)\}$ .Ce domaine correspond aux couples (x,o) de $E \times I_c$ qui vérifient P et ne vérifient ni Q ni Inv .

On a: $B = B_1 \cup B_2 \cup B_3$ (Fig. 2).

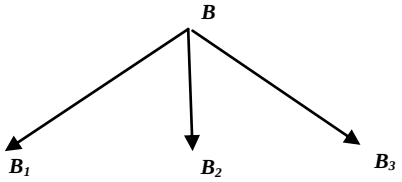

Figure 2. Arborescence de la partition de B

La famille suivante : $(A , A_{21} , A_{22} , A_{23} , A_{24} , B_1 , B_2 , B_3)$ constitue la partition générale du domaine d'entrée $E \times I_c$ pour une méthode d'une classe (Tab. 1).

TABLE I. CAS DE VERITE DE H

	P	Q	Inv	$H : P \Rightarrow (Q \wedge Inv)$
A_1	1	1	1	1
B_1	1	1	0	0
B_2	1	0	1	0
B_3	1	0	0	0
A_{21}	0	0	0	1
A_{22}	0	0	1	1
A_{23}	0	1	0	1
A_{24}	0	1	1	1

B. Analyses des partitions des domaines de définition

Dans le cadre d'un test de conformité, les éléments d'entrée doivent respecter la pré-condition de la méthode à tester. En effet, l'utilisateur (le programme appelant) lui-même se charge de vérifier la pré-condition de la méthode avant d'effectuer l'appel à celle-ci. L'oracle de test proposé refuse dans son état actuel (le cas du test de conformité) d'effectuer l'appel avec un élément d'entrée qui n'est pas valide.

Dans ce sens, nous nous intéressons particulièrement aux éléments d'entrée valides (***i.e.*** les couples (x,o) qui vérifient P). Nous supposons donc que l'utilisateur respecte sa part du contrat, en donnant un vecteur d'entrée (de paramètres) correct. Cela nous invite à préciser le domaine de définition D de m défini par :

$D = \{(x,o) \in E \times I_c / (P(x,o)=1\}$ i.e. les couples (x,o) de $E \times I_c$ qui vérifient la pré-condition P.

La famille (A_1 , B_1 , B_2 , B_3) constitue une partition du domaine D de la méthode m :

$D = A_1 \cup B_1 \cup B_2 \cup B_3$ .On partitionne ce domaine :

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

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

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

$B_3 = \{(x,o) \in E \times I_c / (P(x,o), Q(x,o), Inv(o)) = (1,0,0)\}$
Le domaine restant dans la partition générale $(A_{21}, A_{22}, A_{23}, A_{24})$ correspond à une pré-condition P fausse, ne sera pas implémenté dans le cadre de ce papier.

C. *Partition par classification*

Les domaines étudiés dans les sous paragraphes *A* et *B* pourront être réduits si on tient en compte les classifications de chaque méthode définies dans [3]. En effet ce classement prévoit quatre types de méthodes (C_b : constructeur basique, C_e : constructeur étendu, M : mutateur et O : observateur). La figure 3 illustre des exemples de chaque type de méthode :
-La méthode ***static Account Cb_Account (int)*** de la classe ***Account*** illustre un exemple d'un C_b (Fig. 3).
-La méthode ***static Account Ce_Account (Account)*** de la classe ***Account*** illustre un exemple d'un C_e (Fig. 3).
-La méthode ***void transfer (int, Account)*** de la classe ***Account*** illustre un exemple d'un M (Fig. 3).
-La méthode ***int getbal( )*** de la classe ***Account*** illustre un exemple d'un O (Fig. 3).

Dans ce paragraphe, nous appliquons la partition du domaine à chaque type de méthode.

```
class Account
{
private int bal; // bal is the account balance
        //public Account(int x){ this.bal=x;}
        public static Account cb_Account (int x){
        Account o = new Account();o.bal=x; return o ;}
//public Account (Account x){ this.bal=x.bal;}
        public static Account ce_Account (Account x){
        Account o = new Account ();o.bal=x.bal; return o ;}
        public void setBal(int x){this.bal=x;}
        public int getBal () {return this.bal; }
//x1 to transfer the balance of current account o to the account x2
        public void transfer (int x1,Account x2){
        this.bal=this.bal - x1; x2.bal=x2.bal + x1;}
        public void withdraw (int x1) { this.bal=this.bal - x1;}
// x1 is the balance on deposit in the current account o
        public void deposit (int x1){
        if(x1% 5== 0) this.bal=this.bal + x1*75/100;
        else
        this.bal=this.bal + x1;}
}
```

Figure 3. Implémentation de la classe Account

Constructeur basique C_b : On considère un constructeur basique C_b avec un vecteur de paramètres x variant dans un domaine E. Il ne fait référence à aucun objet (ou instance) lors de son appel, mais, il génère un objet O_x à la sortie. Le domaine d'entrée $E \times I_c$ sera donc réduit au domaine E, et la contrainte H de C_b aura la forme suivante :

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

On n'a pas d'objet récepteur, donc l'invariant dans la formule porte sur le résultat O_x et non sur l'objet récepteur car celui-ci n'existe pas dans le cas des constructeurs (basiques ou étendus). Donc, il y aura une modification légère dans la conception des partitions par rapport à la partition générale d'une méthode.

Dans ce qui suit, nous déterminons la partition du domaine de définition D ($D = \{x \in E / P(x)=1\}$) de C_b

On a $H(x) : [P(x) \Rightarrow (Q(x) \wedge Inv(O_x))], x \in E$
L'objet n'a pas de présence avant l'appel à C_b
On a : $A = \{x \in E / H(x)=1\}$
$A_1 = \{x \in E / (P(x), Q(x), Inv(O_x)) = (1,1,1)\}$
$B = \{x \in E / H(x)=0\} = B_1 \cup B_2 \cup B_3$

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

La famille (A_1, B_1, B_2, B_3) constitue une partition du domaine de définition du constructeur basique C_b.

- **Constructeur étendu C_e :** On considère un constructeur étendu C_e avec un vecteur paramètre $x=(x_1, x_2, \ldots, x_n)$ variant dans un domaine E avec $E=E_1 \times E_2 \times \ldots E_n$.

Chaque constructeur étendu C_e dispose par définition d'au moins un argument de type C.
i.e. $\exists i \in \{1,2,..,n\}$ tels que x_i est un objet de la classe C $(x_i \in I_c)$.

Dans ce type de constructeur l'objet passé en paramètre doit être valide au sens de la spécification (vérifiant l'invariant). La partition du domaine d'entrée se fait de la même manière qu'un constructeur basique C_b.

- **Mutateur M :** On considère un mutateur M tel que $(x,o) \in E \times I_c$ (x le vecteur paramètre de M et o l'objet récepteur) (Fig. 4).

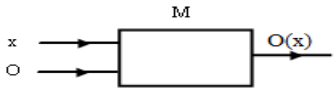

Figure 4. Schéma d'un mutateur

On a $H(x,o) : [P(x,o) \Rightarrow (Q(x,o) \wedge Inv(o))], (x,o) \in E \times I_c$
$Inv(o)$ désigne l'invariant à vérifier après l'appel (state after). L'analyse des partitions pour un mutateur correspond au cas général de partition des domaines étudié dans le sous paragraphe *A*.

Observateur O : L'observateur ne crée pas d'objets, et il ne modifie pas l'état de ses objets d'entrée. Pour ce type de méthode, La contrainte H aura la forme simplifiée suivante :

$$H(x) : P(x) \Rightarrow Q(x), x \in E$$

Cela signifie que l'invariant est égal à 1 dans la formule H.
$A = \{x \in E / H(x)=1\} = A_1 \cup A_{22} \cup A_{24}$
$A_1 = \{x \in E / (P(x), Q(x)) = (1,1)\}$
$A_{22} = \{x \in E / (P(x), Q(x)) = (0,0)\}$
$A_{21} = \emptyset$ et $A_{23} = \emptyset$
$A_{24} = \{x \in E / (P(x), Q(x)) = (0,1)\}$
$B = \{x \in E / H(x)=0\}$
$B_1 = \emptyset$ et $B_2 = \{x \in E / (P(x), Q(x)) = (1,0)\}$ et $B_3 = \emptyset$
$B = B_2 = \{x \in E / (P(x), Q(x)) = (1,0)\}$
La famille $(A_1, A_{22}, A_{24}, B_2)$ constitue donc une partition du domaine E de l'observateur O.et la famille (A_1, B_2) constitue une partition du domaine de définition de l'observateur O

V. GENERATION DES DONNEES DE TEST DE CONFORMITE

Dans ce paragraphe, nous donnons une définition formelle pour décrire la notion de validité d'une méthode d'une classe. Cette définition permettra de mettre en place une base théorique pour tester la conformité des méthodes et générer des données de test.

A. Modèle formel de test de conformité

Nous considérons une méthode ***m*** de la classe ***C*** avec ***o*** l'objet récepteur et ***x*** le vecteur de paramètres ($(x,o) \in E \times I_c$).

Définition2 (Méthode valide) Une méthode ***m*** est dite valide ou conforme à sa spécification si pour tout couple du domaine $E \times I_c$, la contrainte ***H*** est vérifiée : $\forall (x,o) \in E \times I_c$ on a $H(x,o) = 1$.

Autrement, pour une méthode valide : $\forall (x,o) \in E \times I_c$ si (x,o) vérifie ***P*** Alors (x,o) vérifie ***Q*** et ***Inv***, i.e. $(x,o) \in A_1$. Soit ***D*** l'ensemble des couples (x,o) valides vérifiant la pré-condition : $D = \{(x,o) \in E \times I_c / P(x,o) = 1\}$.

Corollaire1 : m *valide* $\Leftrightarrow \forall (x,o) \in E \times I_c : [(x,o) \in D \Rightarrow (x,o) \in A_1]$ Autrement dit une méthode ***m*** est valide si pour tout couple d'entrée (x,o) : si (x,o) vérifie la pré-condition ***P*** alors il vérifie la post-condition ***Q*** et l'invariant ***Inv***.

Corollaire2 : m *valide* $\Leftrightarrow \forall (x,o) \in D : (Q(x,o) \wedge Inv(o))$.

Pour que la méthode ***m*** ne soit pas conforme à sa spécification, il suffit qu'il existe un couple (x,o) du domaine $E \times I_c$ de ***m*** tel que (x,o) ne vérifie pas la contrainte ***H*** .

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

i.e. : $\exists (x,o) \in E \times I_c : P(x,o) \wedge (\neg Q(x,o) \vee \neg Inv(o))$. On dit que ***m*** n'est pas conforme à sa spécification s'il existe un couple (x,o) qui vérifie la pré-condition ***P*** et ne vérifie pas $Q \wedge Inv$.

B. Algorithmes de test de conformité

Les algorithmes de test proposés dans ce paragraphe permettent de générer aléatoirement des données à l'entrée de la méthode afin de déterminer si la méthode à tester est conforme à sa spécification. Cela nécessite que la génération aléatoire porte sur des entrées valides (vérifiant la pré-condition ***P***). L'exécution de chaque algorithme de test de conformité s'arrête lorsque la contrainte ***H*** devient fausse $(H(x,o)=0)$ ou lorsqu'on atteint le seuil maximal de test avec ***H*** vérifiée. Nous appliquons les algorithmes de test de conformité aux constructeurs basiques et étendus, aux mutateurs et observateurs.

- **Test d'un constructeur basique C_b :** La figure 5 montre un fragment de l'algorithme du test de conformité d'un constructeur basique.

```
do{
do{
                    for ( xi in cb  parameter)
                    xi = generate ( Ei) ;
                    x= (x1,x2,…,xn) ;
    }while(!P(x));
Ox=invoke "cb(x)" ;
if(Q(x,Ox)&& Inv(Ox))
                    A1.add(x);
else if( Q(x,Ox)&& !Inv(Ox))
                    B1.add(x);
else if( !Q(x,Ox)&&Inv(Ox))
                    B2.add(x);
else
                    B3.add(x)
    }while(A1.size()<N && B1.isEmpty()&& B2.isEmpty()&& B3.isEmpty());
```

Figure 5. Algorithme de test de conformité d'un constructeur basique

N : le nombre de fois du test, est une valeur limite qu'on doit prendre suffisamment grande.

- **Test d'un mutateur M :** La figure 6 montre un fragment de l'algorithme du test de conformité d'un mutateur : L'analyse de la condition de sortie permet d'apporter des informations utiles sur la validité de la méthode ***m***, et elle permet en cas de la non-conformité de la méthode de connaitre exactement les contraintes affectées par le problème :

- Cas 1 : B_1 n'est plus vide

i.e. $\exists (x,o) \in E \times I_c$, $(x,o) \in B_1$ **(i.e. H=0)** ***m*** n'est donc pas valide et plus précisément ***m*** ne respecte pas l'invariant ***Inv*** de sa classe.

- Cas 2 : B_2 n'est plus vide

i.e. $\exists (x,o) \in E \times I_c$, $(x,o) \in B_2$ **(i.e. H=0).** Il s'en suit que ***m*** n'est pas valide et plus précisément ***m*** ne respecte pas sa post-condition.

- Cas 3 : B_3 n'est plus vide

i.e. $\exists (x,o) \in E \times I_c$, $(x,o) \in B_3$ **(i.e. H=0).** Il s'en suit que ***m*** n'est pas valide et plus précisément ***m*** ne respecte ni sa post-condition ***Q***, ni l'invariant ***Inv*** de sa classe.

- Cas 4 : A_1 atteint le nombre de test **N.** On note les éléments de A_1 ,les couples $(x_1,o_1),(x_2,o_2),\ldots,(x_N,o_N)$ tels que $\forall (x,o) \in \{(x_1,o_1),(x_2,o_2),\ldots,(x_N,o_N)\} : H(x,o)=1$.

Nous précisons dans ce dernier cas que, si le nombre de fois de test ***N*** est suffisamment grand ($N \rightarrow \infty$), et si $\mathbf{A_1}$ est rempli, cela permet de diminuer le risque de ne pas avoir rencontré une éventuelle entrée non conforme à la spécification. Et par conséquent, nous pouvons admettre que la méthode est valide avec une marge d'erreur négligeable.

```
do{
        do{
        for ( xi in M  parameter)
                xi = generate ( Ei) ;
                x= (x1,x2,…,xn) ;
                o = generate_object ( );
        }while(!P(x,o));
        invoke"o.m(x)"
                if(Q(x,o)&&Inv(o))
                        A1.add(x,o);
                elseif( Q(x,o)&&!Inv(o))
                        B1.add(x,o);
                elseif( !Q(x,o)&&Inv(o))
                        B2.add(x,o);
                else
                        B3.add(x,o)
    }while(A1.size()<N&&B1.isEmpty()&& B2.isEmpty()&& B3.isEmpty());
```

Figure 6. Algorithme de test de conformité d'un mutateur M

Le test de conformité pour les constructeurs étendus et les observateurs utilise respectivement le même principe que les constructeurs basiques et mutateurs.

C. Exemple d'étude

Nous considérons la méthode ***transfer*** de la classe ***Account*** (Fig. 3) :

public void transfer (int x_1***,Account*** x_2***)*** avec $x=(x_1, x_2)$

La contrainte ***H*** de la méthode ***transfer*** sous forme algébrique : $H(x,o) : P(x,o) \Rightarrow [Q(x,o) \wedge Inv(o)]$ avec :

$P(x,o)$: $(x_1 \geq 0 \wedge x_1 \leq balance(o) / 2)$

$Q(x,o)$: $((balance(o_{(a)}) = balance(o_{(b)}) - x_1) \wedge (balance(x_{2(a)}) = balance(x_{2(b)}) + x_1))$

$Inv(o)$: $balance(o) \geq 0$

Un test principal ou de conformité de la méthode ***transfer*** avec $N=100$ et x_1 dans l'intervalle $]-200,200[$ donne les résultats ci-dessous. Le nombre d'éléments de A_1 est exactement 100, et la contrainte H est vérifiée pour toutes les itérations ($H=1$), ce qui permet de conclure que la méthode ***transfer*** est valide (ou conforme à sa spécification) (Tab. 2).

TABLE II. RESULTAT D'UN TEST PRINCIPAL DE LA METHODE TRANSFER

Iteration number:	$x = (x_1, x_2)$	O	$P(x,o)$	$(x,o) \in$	$H(x,o)$
1	*(21, Account(34))*	*Account(74)*	*1*	A_1	*1*
2	*(59, Account(98))*	*Account(164)*	*1*	A_1	*1*
3	*(48, Account(109))*	*Account(182)*	*1*	A_1	*1*
…	…	…	…	…	…
….	….	….	….	….	….
…..	…..	…..	…..	…..	…..
98	*(36, Account(7))*	*Account(107)*	*1*	A_1	*1*
99	*(73,Account(30))*	*Account(199)*	*1*	A_1	*1*
100	*(19, Account(77))*	*Account(185)*	*1*	A_1	*1*

Nous considérons la méthode ***deposit*** de la classe ***Account*** (Fig. 3) :

public void deposit (int x_1) avec $x=x_1$ le vecteur x est réduit à un seul élément x_1. La contrainte H de ***deposit*** sous forme algébrique : $H(x,o) : P(x,o) \Rightarrow [Q(x,o) \wedge Inv(o)]$ avec :

$P(x,o)$: $x_1 \geq 0$

$Q(x,o)$: $balance(o_{(a)}) = balance(o_{(b)}) + x_1$

$Inv(o)$: $balance(o) \geq 0$

Un test principal de la méthode ***deposit*** avec $N=100$ et x_1 dans l'intervalle $]-200,200[$ donne les résultats ci-dessous (Tab. 3):

TABLE III. RESULTAT D'UN TEST PRINCIPAL DE LA METHODE DEPOSIT

Iteration number:	$x=x_1$	O	$P(x,o)$	$(x,o) \in$	$H(x,o)$
1	*193*	*Account(71)*	*1*	*A1*	*1*
2	*56*	*Account(112)*	*1*	*A1*	*1*
3	*188*	*Account(95)*	*1*	*A1*	*1*
4	*168*	*Account(143)*	*1*	*A1*	*1*
5	*62*	*Account(79)*	*1*	*A1*	*1*
6	*149*	*Account(88)*	*1*	*A1*	*1*
7	*170*	*Account(151)*	*1*	*B2*	*0*

La méthode ***deposit*** n'est pas valide : $\exists (x,o) \in ExI_c , (x,o) \in B_2$ i.e. $\exists (x,o) \in ExI_c , H(x,o)=0$

Ce couple (x,o) qui ne vérifie pas la contrainte H est celui de l'itération numéro 7, cela montre qu'il y'a un problème au niveau de l'implémentation de cette méthode, en effet la méthode ***deposit*** n'ajoute au compte courant que 75% du solde x_1 passé en argument lorsque ce solde est un multiple de 5 (Fig. 3). On signale que le test principal des méthodes ***transfer*** et ***deposit*** a nécessité le passage par un test d'un constructeur basique c_b afin d'utiliser des objets valides.

VI. CONCLUSION

Dans ce travail, nous avons proposé un modèle théorique pour décrire la notion de conformité d'une méthode d'une classe d'objets par rapport à sa spécification. Ce modèle est basé dans le côté pratique sur la génération aléatoire des données de test. L'étude de l'évolution mathématique de ce modèle pour intégrer d'autres tests dans le but d'analyser les anomalies dues à des données invalides qui induisent des résultats valides. L'établissement des partitions correspondantes et leurs applications aux programmes Java Card spécifiés, sont les perspectives de nos travaux futurs.

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] 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

[4] Yoonsik Cheon, Antonio Cortes, Martine Ceberio, and Gary 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*, July 1-3, 2008, San Francisco, CA, pages 861-866, July 2008.

[5] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. Experimental assessment of random testing for object-oriented software. *In International Symposium on Software Testing and Analy-sis, pages* 84–94. ACM, 2007.

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

[7] C. Oriat. Jartege: A tool for random generation of unit tests for Java classes. *In International Conference on the Quality of Software Architectures*, volume 3712 of LNCS, pages 242–256. Springer-Verlag, Sept. 2005.

[8] J. Zhang, C. Xu, and X.Wang. Path-oriented test data generation using symbolic execution and constraint solving techniques. *In International Conference on Software Engineering and Formal Methods, Beijing, China*, September 26-30, pages 64–71, 2004

[9] C. Boyapati, S. Khurshid, and D. Marinov .Korat: Auto-mated testing based on Java predicates. *In ACM SIGSOFT International Symposium on Software Testing and Analysis, Rome, Italy*, pages 123–133, July 2002