Animation de modèles JML et génération de tests fonctionnels

Frédéric Dadeau
Laboratoire d'Informatique de l'université de Franche-Comté
16 route de Gray
25030 Besançon cedex
dadeau@lifc.univ-fcomte.fr

Résumé : Cet article présente des travaux relatifs à l'animation et à la génération de tests fonctionnels à partir de modèles formels écrits en Java Modeling Language, le langage de modélisation de Java. Pour ce faire, le modèle de données Java est représenté dans une structure logico-ensembliste qui permet de représenter un état du système orienté-objet par un ensemble de contraintes. Cette représentation symbolique permet de calculer les objectifs de tests, et l'animation de la spécification permet de calculer les cas de tests abstraits qui sont ensuite concrétisés sans effort pour être exécutés sur l'implantation sous test.
Mots-clés : génie logiciel, validation, animation, tests fonctionnels, Java, JML.

1 INTRODUCTION

L'arrivée récente des langages d'annotations fournit un contexte idéal pour l'intégration des méthodes formelles dans le processus de développement d'un logiciel. De tels langages sont associés à un langage de programmation, pour lequel ils permettent d'exprimer un modèle formel. L'avantage majeur de ces langages est la proximité qu'ils présentent avec le programme dans lequel ils sont embarqués, puisque tous deux se placent au même niveau d'abstraction. L'apprentissage et l'adoption de ces langages sont facilités par le fait que la syntaxe des annotations est en général fortement inspirée de la syntaxe du langage de programmation.

La génération de tests à partir de modèles [Beizer, 1995] est une pratique du génie logiciel visant à la validation de programmes en s'appuyant sur des cas de tests calculés à l'aide d'un modèle formel du programme. Le modèle est utilisé pour calculer les cas de tests, en tant que succession d'appels d'opérations, et aussi pour fournir l'oracle, i.e., le résultat attendu par le programme lorsque les cas de tests sont exécutés sur celui-ci. Le principal problème de cette approche est de réussir à concrétiser les éléments du modèle (souvent abstraits) pour trouver leur correspondance dans le programme. Ceci passe par une phase de concrétisation du cas de test qui va consister à la traduire dans le langage cible, en faisant la passerelle entre les éléments du modèle et les éléments du programme (opérations et variables). Le problème de l'oracle repose sur la correspondance entre les variables du modèle et celles du programme. Si cette correspondance est directe, alors une variable du modèle correspond à une variable du programme. Néanmoins, ce cas n'est pas le plus fréquent et la plupart du temps les variables du modèle ne trouvent pas de correspondance directe. L'utilisateur doit alors établir la conformité en mettant en place des "phases d'observations" qui permettent de comparer les éléments dits "observables" entre le programme et son modèle formel. Ainsi la construction du modèle impose la définition de tels éléments qui est une charge supplémentaire à la mise en place de ce processus.

L'utilisation des langages d'annotations simplifie considérablement les choses. Tout d'abord la concrétisation des cas de tests est directe puisque le modèle et l'implantation partagent les mêmes opérations (ayant, par conséquent, les mêmes signatures). Ensuite, la proximité entre les annotations et le programme permet aisément à des compilateurs spécifiques d'insérer la vérification des prédicats des annotations lors de l'exécution du programme, par l'insertion d'assertions. Ce *monitoring* est utilisé comme oracle dans les approches de test. Si, durant l'exécution du cas de test, une des assertions n'est pas vérifiée, alors le test échoue. A l'inverse, s'il ne viole aucune assertion, le test réussit.

Nous nous intéressons dans ce papier à des travaux ayant trait aux langage d'annotations associé à Java : le Java Modeling Language (JML) [Leavens, 1999]. Nous proposons ainsi une démarche complètement automatique pour la génération de tests à partir d'un programme Java annoté en JML. Pour ce faire, le modèle JML doit d'abord être validé, par animation, pour s'assurer que le comportement du modèle est conforme aux spécifications informelles données dans le cahier des charges. Ensuite, le modèle JML est utilisé pour définir les objectifs de tests. Les cas de tests sont calculés par une séquence d'animation du modèle qui, à partir de l'état initial, permet d'atteindre l'objectif précédemment identifié. Ce processus résume les grandes lignes de cet article.

Tout d'abord, nous verrons en partie 2 une présentation du Java Modeling Language. La partie 3 introduira les travaux autour de l'animation des modèles JML, qui seront utiles par la suite. La partie 4 définira le calcul des objectifs de tests, en particulier le calcul des valeurs dites "aux limites" des données. La partie 5 présentera le calcul des cas de tests. Pour finir, nous présenterons en partie 6 l'environnement qui a été développé pour implanter ces travaux, avant de conclure en partie 7.

```
class Purse {

  //@ invariant balance >= 0;
  protected short balance;

  /*@ public normal_behavior
    @   requires b >= 0;
    @   assignable balance;
    @   ensures balance == b;
    @*/
  public Purse(short b) {...}

  /*@ behavior
    @   requires a > 0;
    @   assignable balance;
    @   ensures balance == \old(balance) + a;
    @*/
  public void credit(short a) {...}

  /*@ normal_behavior
    @   assignable \nothing;
    @   ensures \result <==>
    @           (this.balance == p.balance)
    @*/
  public /*@ pure */ boolean equals(Purse p) {...}

  /*@ behavior
    @   requires a >= 0;
    @   {|
    @      requires balance >= a;
    @      assignable balance;
    @      ensures balance == \old(balance) - a;
    @   also
    @      requires balance < a;
    @      assignable balance;
    @      signals (NoCreditException e)
    @                  balance == \old(balance);
    @   |}
    @*/
  public void debit(short a)
                throws NoCreditException {...}

  /*@ normal_behavior
    @   assignable \nothing;
    @   ensures \result == balance;
    @*/
  public /*@ pure */ short getBalance() {...}

  /*@ normal_behavior
    @    p != null;
    @    assignable balance;
    @    ensures this.equals(p);
    @*/
  public void transfer(Purse p) {...}
}

class LimitedPurse extends Purse {

  //@ invariant max == 10000 && balance <= max;
  //@ constraint \not_modified(max);
  static short max = 10000;

  /*@ normal_behavior
    @   requires b >= 0 && b <= max;
    @   assignable balance;
    @   ensures balance == b;
    @*/
  public LimitedPurse(short b) {...}

  /*@ normal_behavior
    @    requires a >= 0 && balance+a <= max;
    @    assignable balance;
    @    ensures balance == \old(balance)+a;
    @  also
    @    requires a < 0 || balance+a > max;
    @    assignable balance;
    @    ensures balance == \old(balance);
    @*/
  public void credit(short a) {...}
}
```

FIG. 1 – Une spécification JML d'un porte-monnaie électronique simplifié

2 JAVA MODELING LANGUAGE

Le Java Modeling Language [Leavens, 1999] est le langage de modélisation dédié au langage de programmation Java. Il est constitué d'assertions embarquées dans le programme Java. Ces assertions se présentent sous la forme de commentaires qui sont ignorés par les compilateurs Java standards et ne sont pris en compte que par les compilateurs JML spécifiques. Les annotations JML décrivent la spécification associée à la classe en terme de clauses donnant les propriétés statiques ou dynamiques. Les propriétés statiques sont les invariants (clause `invariant`), les contraintes historiques (clause `constraints`), qui sont appliquées à la classe entière. Les propriétés dynamiques sont données par des *spécifications de méthodes* qui précisent l'évolution du système. Les spécifications de méthodes sont composées de préconditions (clause `requires`), d'une postcondition normale (clause `ensures`) ou exceptionnelle (clause `signals`) qui précise la postcondition qui s'applique respectivement lorsque la méthode termine normalement ou exceptionnellement en déclenchant une exception. La syntaxe des prédicats JML est similaire à la syntaxe Java, enrichie de mot-clés spéciaux, commençant par un \, tels que `\result` qui représente la valeur de retour d'une méthode, ou `\not_modified(x)` qui signifie, dans une postcondition, que `x` n'est pas modifié.

JML repose sur les principes de conception par contrat (design by contract), introduits par Eiffel [Meyer, 1997]. De ce fait, le système doit respecter le contrat d'activation de la méthode (i.e., sa précondition) pour l'invoquer. En contre-partie, la méthode s'engage à établir sa postcondition.

La figure 1 illustre l'utilisation de JML à travers un exemple. Cette spécification décrit un porte-monnaie électronique simplifié (classe `Purse`) qui est étendu par un porte-monnaie plafonné (classe `LimitedPurse`). Cette limitation force le solde du porte-monnaie à être inférieur à un montant fixé. Il est possible de créditer ou de débiter le porte-monnaie, ou encore de transférer le contenu d'un porte-monnaie dans un autre porte-monnaie, en copiant son solde.

Le *Runtime Assertion Checker* (RAC) [Leavens, 2002] est un compilateur dédié à JML qui enrichit le bytecode Java avec la vérification des différentes clauses JML. L'exécution d'une classe ainsi compilée permet de vérifier automatiquement les assertions de la spécification lors de l'exécution du programme. Si l'exécution viole une des assertions JML, une exception spécifique est déclenchée, indiquant quelle assertion n'a pas été satisfaite. Cette possibilité est ainsi utilisée comme oracle dans les approches de tests.

JML a deux utilisations principales. La première est de renforcer le code pour aider à la preuve du programme (par exemple en utilisant Jack [Burdy, 2003]

ou ESC/Java2 [Cok, 2004]. Notre philosophie est de considérer JML comme un langage de spécification à part entière qui ne nécessite pas de code Java pour être employé. Si cette hypothèse peut paraître forte pour n'importe quel programme Java, nous pensons que l'effort est payant dans le cadre des programmes embarqués comme les programmes JavaCard [Sun, 2000]. Nous utilisons JML dans notre approche pour la définition des objectifs de test, et le calcul des cas de test par animation.
Nous nous intéressons dans la section suivante à la représentation symbolique de spécifications JML et nous présentons l'animation des spécifications JML.

3 ANIMATION SYMBOLIQUE DE MODÈLES JML

Le concept d'animation symbolique sous-entend l'utilisation de solveurs de contraintes pour la gestion des valeurs des variables du système. Pour réaliser l'animation des spécifications JML, nous avons dans un premier temps besoin d'exprimer le modèle de données Java dans environnement permettant l'animation. Ensuite nous devons extraire les comportements (i.e., les transitions) à partir des spécifications de méthodes JML. Nous terminons cette partie en décrivant comment se réalise l'animation.

3.1 Représentation du modèle de données Java

Nous disposons d'un solveur de contraintes ensembliste dans lequel nous souhaitons exprimer le modèle de données JML. Ce solveur, nommé CLPS-BZ [Bouquet, 2002] permet d'utiliser des structures arithmétiques (couplé avec le solveur CLP(FD) de SICStus Prolog), ensemblistes, relationnelles et fonctionnelles. Initialement, ce solveur était au cœur du projet BZ-Testing-Tools [Ambert, 2002] visant à l'animation et la génération de tests à partir de modèles écrits en B [Abrial, 1996]. Ce solveur fonctionne à partir d'un langage de plus haut niveau, nommé BZP, qui permet la déclaration d'éléments de modélisation tels que des modules, contenant variables, constantes et opérations. Des prédicats exprimés en logique du premier ordre, avec opérateurs arithmétiques, ensemblistes et relationnels permettent d'exprimer des assertions et de décrire les pré- et postconditions des opérations. L'idée principale est d'exprimer le modèle objet dans ce format, pour ainsi pouvoir utiliser les possibilités d'animation et de résolution de contraintes offertes par cet environnement.
Le modèle de données Java est représenté dans ce format de la manière suivante. Chaque classe correspond à un module. Chaque attribut de classe correspond à une variable. On désignera un module spécifique, nommé `system`, qui contient une constante nommée `heap`, notée $\mathcal{H}$, qui modélise l'ensemble de adresses de la mémoire. Chacun des modules représentant une classe C contient une variable `instances`, notée $\mathcal{I}_C$, qui représente l'ensemble des instances de la classe C allouées dans la mémoire. La représentation des attributs distingue les attributs d'instances, dont la valeur est propre à chaque objet, et les attributs statiques, dont la valeur est commune à tous les objets.

Nous représentons l'état du système par un environnement symbolique, dont les valeurs des variables sont gérées par le solveur CLPS-BZ.

Définition 1 (Représentation des attributs) *Soit $\mathcal{C}$ l'ensemble des noms de classes, $\mathcal{A}_{inst}$ l'ensemble des noms d'attributs d'instance, $\mathcal{I}_c$ l'ensemble des instances créées pour la classe $c \in \mathcal{C}$, $\mathcal{D}_{\mathcal{A}_{inst}}$ le domaine des attributs traduits dans le formalisme du solveur* CLPS-BZ*, $\mathcal{K}$ l'ensemble des genre de données, et $\mathcal{T}$ l'ensemble des types* CLPS-BZ *types possibles.*
La représentation des attributs d'instance est donnée par une fonction totale associant à chaque instance sa valeur, pour l'attribut considéré :

$$\mathcal{C} \times \mathcal{A}_{inst} \nrightarrow (\mathcal{I}_c \rightarrow \mathcal{D}_{\mathcal{A}_{inst}}) \times \mathcal{K} \times \mathcal{T} \tag{1}$$

A l'inverse, les attributs statiques $\mathcal{A}_{static}$ ne sont pas dépendants des instances de la classe et leur valeur est obtenue directement :

$$\mathcal{C} \times \mathcal{A}_{static} \nrightarrow \mathcal{D}_{\mathcal{A}_{static}} \times \mathcal{K} \times \mathcal{T} \tag{2}$$

Ainsi un état symbolique $\mathcal{S}$ est composé d'un environnement Env_V auquel est rattaché une système de contraintes C_V reliés par un ensemble de variables V :

$$\mathcal{S} = \langle C_V, Env_V \rangle \tag{3}$$

Après avoir vu comment se présentent les états symboliques, nous présentons désormais l'extraction des comportements qui représentent les transitions entre les états symboliques.

3.2 Extraction des comportements

Les transitions du système sont les comportements extraits des spécifications de méthodes JML. Un comportement est un quadruplet $\langle Pre, Term, Post, Assign \rangle$ composé d'une précondition Pre, d'une terminaison $Term$ –indiquant *no_exception* si la méthode termine normalement ou l'exception qui est déclenchée si elle termine de manière exceptionnelle–, d'une postcondition $Post$ et d'un prédicat $Assign$ indiquant les égalités entre les champs qui ne sont pas modifiés entre l'état avant et l'état après.

```
/*@ behavior
  @    requires P1;
  @    assignable A;
  @    ensures Q1;
  @    signals (E11 e11) S11;
  @    ...
  @    signals (E1M e1M) S1M;
  @ also
  @    ...
  @ also
  @    requires PN;
  @    assignable A;
  @    ensures QN;
  @    signals (EN1 eN1) SN1;
  @    ...
  @    signals (ENP eNP) SNP;
  @*/
Type meth(T1 p1, ...) throws E1, ..., EN { ... }
```

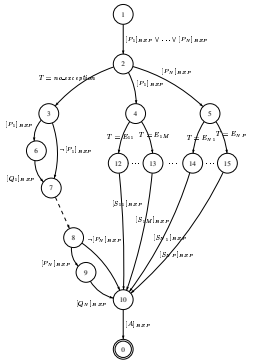

FIG. 2 – Comportements extraits d'une spécification de méthode JML

Ces comportements sont extraits des spécifications de méthode selon les principes décrits dans la figure 2. Chaque comportement est désigné par un chemin du graphe menant du nœud 1 au nœud 0.
Les comportements extraits d'une spécification de méthode JML sont ensuite utilisés pour calculer les transitions entre des états symboliques du système. Nous décrivons à présent ce mécanisme.

3.3 Animation symbolique

L'animation symbolique d'une spécification JML consiste à résoudre un problème de satisfaction de contraintes (CSP) entre (i) un système de contraintes représentant l'état avant, (ii) un système de contraintes représentant l'état après, et (iii) les contraintes entre les deux systèmes exprimées à l'aide de prédicats dans la syntaxe du solveur.
Pour traduire les prédicats BZP dans la syntaxe du solveur CLPS-BZ, il est nécessaire de procéder (i) à une réécriture des opérateurs qui ne sont pas nativement supportés par le solveur, (ii) à un replacement des références aux données par leurs valeurs extraites dans l'environnement. Nous appelons $\mathsf{reduce}(C_V, P_{BZP})$ la fonction qui réalise ce traitement à partir d'un ensemble de contraintes C_V et d'un prédicat P_{BZP} exprimé dans le format BZP.

Définition 2 (Activation d'un comportement) *Soit $\mathcal{S}=\langle C(V), Env(V)\rangle$ un état symbolique, représenté par un système de contraintes $C(V)$ et un environnement d'exécution $Env(V)$ impliquant un vecteur V de variables. Soit T une transition, représentée par une conjonction de prédicats exprimés dans le format intermédiaire BZP. L'état symbolique $\mathcal{S}'=\langle C(V'), Env'(V')\rangle$ résultant de l'exécution de la transition est défini par :*

$$C(V') = C(V) \cup \mathsf{reduce}(Env(V), T) \qquad (4)$$

et Env' désigne l'environnement d'exécution contenant les valeurs après des variables.

Illustrons l'animation par un exemple.

Exemple 3.1 (Animation de l'exemple) Reprenons l'exemple donné en figure 1. Considérons la séquence d'animation consistant à créer un objet de type `Purse` en laissant le paramètre `b` non spécifié. Initialement tous les ensembles d'instances des classes sont vides et aucun attribut n'est initialisé, à part l'attribut statique `max`. L'environnement initial est donc le suivant :

$\langle \emptyset, \{\mathcal{H} \mapsto (\{addr_0, addr_1, \ldots\}, \texttt{constant}, \texttt{set(atom)}),$
$\mathcal{I}_{Purse} \mapsto (\emptyset, \texttt{variable}, \texttt{set(atom)}), \mathcal{I}_{LimitedPurse} \mapsto$
$(\emptyset, \texttt{variable}, \texttt{set(atom)}), Purse.balance \mapsto (\emptyset, \texttt{variable},$
$\texttt{set(pair(atom,int))}), LimitedPurse.max \mapsto (10000,$
$\texttt{variable}, \texttt{int})\}\rangle$

On remarque que l'ensemble $\mathcal{H}$ est arbitrairement fixé à un nombre d'instances donné. La création d'un nouvel objet `Purse` avec un paramètre symbolique, résulte en l'environnement d'exécution suivant :

$\langle \{\_X \in 0..32767\}, \{\mathcal{H} \mapsto (\{addr_0, addr_1, \ldots\}, \texttt{constant},$
$\texttt{set(atom)}), \mathcal{I}_{Purse} \mapsto (\{addr_0\}, \texttt{variable}, \texttt{set(atom)}),$
$\mathcal{I}_{LimitedPurse} \mapsto (\emptyset, \texttt{variable}, \texttt{set(atom)}),$
$Purse.balance \mapsto (\{addr_0 \mapsto \_X\}, \texttt{variable},$
$\texttt{set(pair(atom,int))}), LimitedPurse.max \mapsto (10000,$
$\texttt{variable}, \texttt{int})\}\rangle$

On remarque l'introduction d'une variable CLPS-BZ, notée $\_X$ qui représente la nouvelle valeur du solde du porte-monnaie. Comme l'illustre l'exemple, l'animation symbolique permet de réduire considérablement la taille du graphe d'atteignabilité. Là où une approche énumérative (de type model-checking) aurait obtenu 32768 possibilités pour ce simple exemple d'exécution, notre approche permet de les représenter en une transition et un seul état symbolique.
L'animation d'un modèle permet de valider celui-ci dans l'objectif de s'assurer qu'il se comporte conformément aux descriptions informelles données dans le cahier des charges. En complément, nous nous basons sur l'animation symbolique pour le calcul des cas de test abstraits basés sur le modèle. Pour ce faire, le modèle est animé pour atteindre, par une séquence d'appels de constructeurs d'objets et de méthodes, un objectif de test calculé au préalable. La définition des objectifs de tests est abordée dans la partie suivante.

4 DÉFINITION DES OBJECTIFS DE TESTS

L'originalité de notre approche est de considérer la spécification JML à part entière pour le calcul des tests. On trouve dans la littérature d'autres approches, comme TOBIAS [Ledru, 2004] réalisant du test combinatoire à partir de schémas de tests, ou Jartege [Oriat, 2005], réalisant du test aléatoire, qui n'utilisent la spécification JML qu'*a posteriori* comme oracle, une fois les cas de test calculés. Nous définissons nos objectifs de test en fonction de la couverture du modèle que nous souhaitons avoir. Pour ce faire, nous décomposons cette couverture selon trois critères que nous expliquons à présent : la couverture structurelle, la couverture des disjonctions, et la couverture des données.

4.1 Couverture structurelle

La couverture structurelle consiste à rechercher l'activation des comportements de la spécification. Nous prenons comme hypothèse de travail d'avoir des spécifications de méthodes déterministes (comme illustré en figure 3), dans le sens où la terminaison de la méthode est soit (a) normale, soit (b) exceptionnelle, et ce, sans ambiguïté due à des préconditions de spécification de méthodes trop faibles.
L'objectif de la couverture des comportements est de considérer la spécification de la méthode JML comme un graphe de flot de contrôle que nous cherchons à couvrir.
Un comportement $Cpt = Cpt_{avant} \wedge Cpt_{apres}$ est ainsi vu comme un prédicat avant/après pour lequel nous cherchons à activer la partie avant. L'extraction des comportements se fait sur la base d'un critère *tous les chemins* qui vise à récupérer tous les combinaisons de prédicats compris entre le nœud 1 et le nœud 0 du graphe.

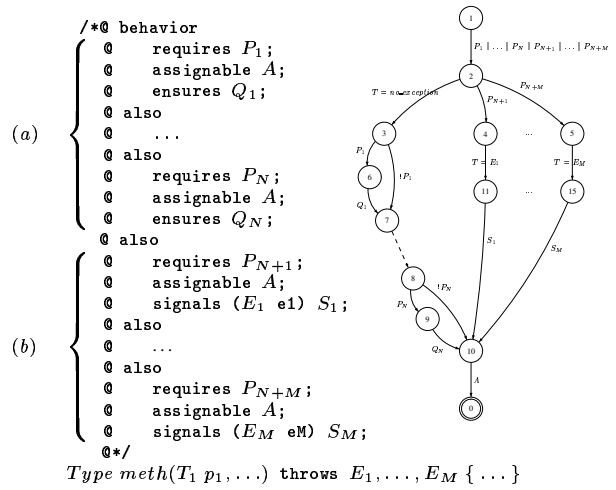

FIG. 3 – Extraction des comportements déterministes d'une spécification de méthode JML

4.2 Couverture des disjonctions

Pour chaque comportement, il est intéressant de voir comment satisfaire les disjonctions qui composent la conjonction de prédicats constituant le comportement. Pour ce faire, nous considérons 4 réécritures, chacune d'elle satisfaisant un critère de couverture spécifique.
Le tableau 1 donne les réécritures des disjonctions possibles et les critères de couverture associés. La réécriture 1 consiste à laisser la disjonction sans la modifier. La réécriture 2 vise à créer une cible de test pour chacun des éléments de la disjonction indépendemment. La réécriture 3 cherche à activer chacun des éléments de la disjonction de manière exclusive. Enfin, la réécriture 4 vise à produire autant de cibles que de possibilités de satisfaire la disjonction.

Id	Réécriture de $P_1 \vee P_2$	Couverture
1	$P_1 \vee P_2$	DC et SC
2	$P_1 \,[]\, P_2$	D/CC
3	$(P_1 \wedge \neg P_2) \,[]\, (\neg P_1 \wedge P_2)$	FPC
4	$(P_1 \wedge P_2) \,[]\, (P_1 \wedge \neg P_2) \,[]\, (\neg P_1 \wedge P_2)$	MCC

TAB. 1 – Réécriture des prédicats disjonctifs

4.3 Couverture des données

Une des originalités de notre approche est de considérer des valeurs "aux limites" pour les données. Pour ce faire, nous définissons la notion de valeur "limite" pour les valeurs numériques et pour les objets.
Pour savoir quels sont les données –attributs ou paramètres– qui doivent être sélectionnées aux limites, nous analysons les données apparaissant dans le comportement ciblé extrait de la méthode qui est testée.
Une valeur numérique est dite "à sa limite" si, pour un contexte donné, elle est à un extremum de son domaine courant.

Définition 3 (Valeur numérique aux limites) *Soit D_{num} l'ensemble des données numériques qui sont employées dans le comportement Cpt_i, extrait de la méthode à tester. Soit Inv l'invariant du système, et soit f une fonction d'optimisation telle que $f(X) = \sum_{x \in X}(x)$. Les données de test des valeurs aux limites de D_{ord} d'un comportement Cpt_i sont calculées en utilisant les fonctions :*

$$BV_i^{min}(Cpt_i) = \mathsf{minimize}(f(D_{ord}), Inv \wedge Cpt_i)$$
$$BV_i^{max}(Cpt_i) = \mathsf{maximize}(f(D_{ord}), Inv \wedge Cpt_i)$$

où minimize *(resp.* maximize*) est la fonction d'instanciation qui calcule la valeur minimale (resp. maximale) de son paramètre, en sélectionnant la borne inférieure (resp. supérieure) de son domaine.*

Un objet aux limites correspond à une valeur spécifique d'un objet associée à des contraintes spécifiques sur cet objet.

Définition 4 (Valeur objet aux limites) *Soit $D_{obj} = \{\langle T_1, o_1\rangle, \langle T_2, o_2\rangle, \ldots\}$ l'ensemble des données objet (composées d'un couple $\langle$ type statique, variable$\rangle$) du comportement Cpt_i extrait de la méthode testée. On appelle ID_{obj} l'ensemble des paramètres d'entrées objets ($ID_{obj} \subseteq D_{obj}$). Les données de test sont définies pour chaque $\langle T_i,\ o_i\rangle \in D_{obj}$ par :*

1. o_i `==` `null`
2. o_i `==` `this`
3. o_i `!=` `null` `&&` o_i `!=` `this` `&&` `\typeof(`o_i`)` `==` `\type(`T_i`)`
4. o_i `!=` `null` `&&` o_i `!=` `this` `&&` `\typeof(`o_i`)` `<:` `\type(`T_i`)`
5. o_i `!=` `null` `&&` o_i `!=` `this` `&&` o_i `==` o_j *où* $o_j \in ID_{obj}$

où la notation JML `\typeof(`A`)` `<:` *(resp.* `==`*)* `\type`B *désigne le fait que le type dynamique de l'objet A est un sous-type (resp. est) du type B.*

Commentons informellement ces valeurs. En 1., nous cherchons à tester les problèmes potentiels dûs aux pointeurs `null`. En 2., si les types sont compatibles, nous tentons de passer une référence à l'objet courant `this`. En 3., nous considérons un objet de type dynamique conforme au type statique attendu. En 4., nous considérons un objet de type un des sous-types du type dynamique attendu. Pour finir, en 5., nous proposons de tester l'aliasing en forçant deux références d'objets de types compatibles à être identiques.
Nous autorisons les deux types de couvertures de données à s'entrelacer, pour permettre notamment à un attribut d'objet d'être aux limites. L'application successive de ces critères de couvertures permet de définir un ensemble de contraintes, appelé prédicat de spécialisation et noté P_{spe}, qui identifie un état symbolique. Avant de passer au calcul de la trace menant à la cible de test, récapitulons par un exemple illustratif les différents critères de couvertures.

Exemple 4.1 (Critères de couverture) Reprenons l'exemple introduit en figure 1. Considérons la méthode `transfer` de la classe `Purse`, héritée par la classe

LimitedPurse. Cette méthode n'a qu'un seul comportement, ne contenant pas de disjonction. Les données mises en jeu dans le comportement sont le paramètre objet p et le champ balance de l'objet this.
Les cibles de tests sont composées de la partie avant du comportement, et d'un prédicat de spécialisation.

```
balance >= 0 && balance <= max && p != null && P_spe
```

Les prédicats de spécialisation possibles sont donnés par le tableau 2, dans lequel R_{num} désigne la règle appliquée pour le calcul des valeurs numériques, et R_{obj} donne la règle appliquée pour le calcul des valeurs objets.

	Prédicat de spécialisation P_{spe}	R_{num},R_{obj}
(a)	`this.balance == 0 && p == this`	min., (1)
(b)	`this.balance == 10000 && p == this`	max., (1)
(c)	`this.balance == 0 && p != null && p != this && p.balance == 0 && \typeof(p)==\type(Purse)`	min.,(3)
(d)	`this.balance == 10000 && p != null && p != this && p.balance == 32767 && \typeof(p)==\type(Purse)`	max.,(3)
(e)	`this.balance == 0 && p != null && p != this && p.balance == 0 && \typeof(p)==\type(LimitedPurse)`	min.,(4)
(f)	`this.balance == 10000 && p != null && p != this && p.balance == 10000 && \typeof(p)==\type(LimitedPurse)`	max.,(4)

TAB. 2 – Prédicat de spécialisation pour la méthode transfer de l'exemple

Ces contraintes définissent un état cible, que nous allons chercher à atteindre par animation symbolique du modèle.

5 CALCUL DES CAS DE TEST

Une fois que l'objectif de test est identifié, nous devons calculer le préambule qui va, depuis l'état initial, construire la séquence d'appels de constructeurs et de méthodes qui va mener à la cible de test. Le test en lui-même, i.e., l'activation du comportement considéré sera ensuite effectué.
Comme nous l'avons vu en partie précédente, un objectif de test est défini par rapport aux paramètres d'entrée et aux attributs des objets qui occurrent dans le comportement considéré. Ainsi, le calcul de préambule est guidé par deux objectifs.

1. Activer le comportement considéré, ceci requiert de créer un objet sur lequel invoquer la méthode testée, mais cela requiert également que les attributs de l'objet aient une structure particulière.
2. Créer les objets qui vont être employés comme paramètres, en plaçant leurs attributs à des valeurs compatibles avec celles attendues.

Ces deux objectifs sont exprimés à l'intérieur d'un état symbolique, associé à un système de contraintes. L'animation symbolique du modèle est ensuite mise en œuvre en utilisant un algorithme de type "meilleur d'abord" guidé par des heuristiques ad hoc, estimant le meilleur coût, terme de distance entre l'état évalué et l'état cible [Colin, 2004].

Une fois les séquences de tests abstraites calculées, le processus de concrétisation pour produire du code Java est immédiat. Ceci est dû, comme nous l'avons déjà signalé, à la proximité du langage de modélisation et du langage de programmation. Le Runtime Assertion Checker de JML est ensuite en charge de vérifier la validité des annotations JML lors de l'exécution des cas de test sur l'implantation.

Exemple 5.1 (Cas de test) Reprenons les cibles de test calculées dans l'exemple 4.3. Les cas de test Java correspondants aux cibles et leurs résultats obtenus en utilisant le Runtime Assertion Checker de JML sont donnés par le tableau 3, où a1 et a2, sont des noms de variables automatiquement générés. Nous remarquons que le cas de test échoué (levant une violation de l'invariant JML) révèle une erreur de conception puisque la méthode transfer n'a pas été redéfinie dans la sous-classe LimitedPurse, et autorise n'importe quelle valeur du paramètre de type Purse à être transféré et particulièrement ceux dont le solde est plus grand que la limitation max.

Cible	Cas de test Java	Verdict
(a)	`LimitedPurse a1 = new LimitedPurse(0);` `a1.transfer(a1);`	Succès
(b)	`LimitedPurse a1 = new LimitedPurse(10000);` `a1.transfer(a1);`	Succès

(c)	`LimitedPurse a1 = new LimitedPurse(0);`
	`Purse a2 = new Purse(0);`
	`a1.transfer(a2);`
		Succès

(d)	`LimitedPurse a1 = new LimitedPurse(10000);`
	`Purse a2 = new Purse(32767);`
	`a1.transfer(a2);`
		Echec

(e)	`LimitedPurse a1 = new LimitedPurse(0);`
	`LimitedPurse a2 = new LimitedPurse(0);`
	`a1.transfer(a2);`
		Succès

(f)	`LimitedPurse a1 = new LimitedPurse(10000);`
	`LimitedPurse a2 = new LimitedPurse(10000);`
	`a1.transfer(a2);`
		Succès

TAB. 3 – Cas de test produits pour la méthode transfer de l'exemple

Le calcul complet de cas de tests en utilisant l'animation de la spécifiction est une réelle originalité, qui n'a encore jamais été ciblée par aucune approche. La plupart des approches, comme TestEra [Khurshid, 2004] ou plus récemment Korat [Boyapati, 2002], utilisent la spécification JML pour déduire des structures de données objet complexes. Néanmoins, les enchaînements de méthodes ne sont pas considérés, et ces outils supposent l'existence d'assesseurs permettant de construire à moindre frais les structures d'objets.
Nous présentons à présent l'implantation qui a été réalisée pour mettre en pratique les idées proposées dans ce papier.

6 JML-TESTING-TOOLS, UN OUTIL DE VALIDATION À PARTIR DE JML

L'outil JML-Testing-Tools implante les travaux présentés dans ce papier sur l'animation des spécifications JML et de la génération de tests. L'architecture de ces outils est donnée par la figure 4. Une spécification JML est tout d'abord analysée, puis traduite en un fichier au format BZP, à partir duquel sont réalisés les processus d'anima-

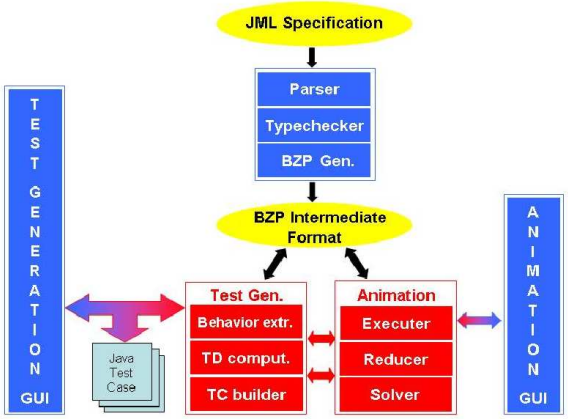

FIG. 4 – Architecture de l'outil JML-Testing-Tools

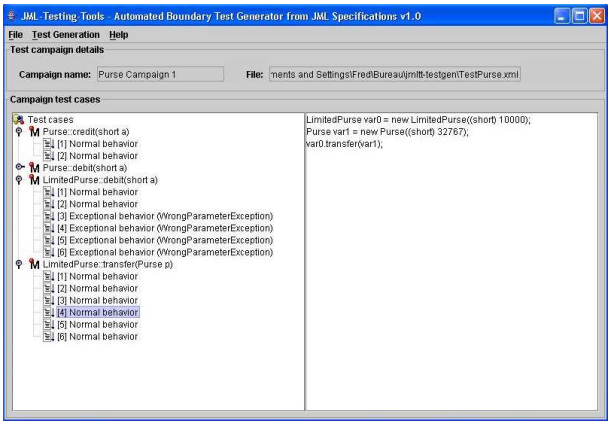

FIG. 6 – L'interface de génération de tests de JML-Testing-Tools

tion et de génération de tests.

L'animateur de spécifications [Bouquet, 2005], présenté en figure 5 permet la validation de spécifications JML par animation symbolique. Pour ce faire, l'utilisateur invoque des constructeurs des classes disponibles pour créer des objets. Une fois ces objets créés, il est possible d'invoquer des méthodes sur ceux-ci. Lorsqu'une invocation requiert la saisie de paramètres, l'utilisateur est libre de saisir une valeur ou de laisser le paramètre non spécifié. Dans ce cas, une valeur symbolique est créée et tous les attributs impactés par ce paramètre prennent également une valeur symbolique.

Le générateur de tests permet de produire de manière automatique des cas de test pour les programmes Java. L'utilisateur sélectionne les méthodes qu'il souhaite tester et les critères de couverture qu'il souhaite appliquer. L'outil présente, en retour, les cas de test calculés comme une suite d'instructions Java. Ces tests sont également produits dans un fichier Java de sortie, intégrant les cas de test dans des méthodes et un pilote de test réalisant les appels à ces fonctions, et récapitulant au final les cas de test ayant réussi et ceux ayant échoués.

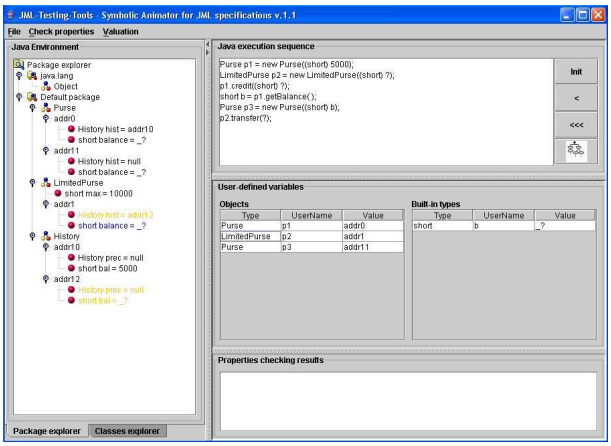

FIG. 5 – L'interface d'animation de JML-Testing-Tools

7 CONCLUSION ET PERSPECTIVES

Nous avons présenté dans ce papier un mécanisme de génération automatique de tests à partir d'un modèle JML. Cette approche est originale dans le sens où nous nous intéressons à produire des cas de test en utilisant la spécification JML à part entière. Tout d'abord, cette spécification est utilisée pour calculer les objectifs de tests. Puis celle-ci est animée pour atteindre un état menant à l'objectif de test. Les cas de tests produits sont ensuite concrétisés pour produire du code Java directement exécutable. Afin de mettre au point une spécification valide pour le processus de génération de test, nous proposons un mécanisme d'animation du modèle JML permettant d'aider à la conception d'un modèle JML efficace. Nous avons également présenté un outil implantant ces principes et nommé JML-Testing-Tools. Si ce processus requiert un effort important de modélisation, notamment au niveau de la description des postconditions des méthodes permettant ainsi l'animation en elle-même, nous pensons qu'il peut toutefois s'appliquer dans le domaine des logiciels embarqués (par exemple le domaine de la carte à puce), qui demande des garanties de sécurité importantes. Une application de ces techniques a d'ailleurs été appliquée à une étude de cas inspirée du monde industriel, l'applet Demoney [Marlet, 2002].

Les perspectives à ces travaux sont nombreuses. Tout d'abord, nous souhaitons nous doter de solveurs adéquats pour augmenter la couverture des données Java/JML supportées. En particulier, l'intégration des tableaux peut se révéler intéressante pour permettre de traiter des études de cas de grande ampleur, sans avoir à réaliser trop d'adaptations pour se conformer aux restrictions de notre approche. Une autre perspective serait de s'appuyer sur le mécanisme d'animation symbolique pour combler les faiblesses des approches de tests combinatoires, comme TOBIAS [Ledru, 2004], en filtrant les séquences de tests produites, pour ne considérer que celles consistantes à l'animation. Finalement, nous nous intéressons à comparer la couverture de notre approche par rapport à d'autres en mesurant les résultats que nous obtenons sur un exercice de détection de mutants.

BIBLIOGRAPHIE

[Abrial, 1996] Abrial, J-R. 1996. *The B-BOOK : Assigning Programs to Meanings*. Cambridge University Press.

[Ambert, 2002] Ambert, F. and Bouquet, F. and Chemin, S. and Guenaud, S. and Legeard, B. and Peureux, F. and Vacelet, N. and Utting, M. 2002. BZ-TT : A tool-set for test generation from Z and B using constraint logic programming. Proc. of the CONCUR'02 Workshop on Formal Approaches to Testing of Software (FATES'02), Brnö, Czech Republic, pp 105–120.

[Beizer, 1995] Beizer, D. 1995. *Black-Box Testing : Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, New York, USA.

[Bouquet, 2005] Bouquet, F. and Dadeau, F. and Legeard, B. and Utting, M. JML-Testing-Tools : a Symbolic Animator for JML Specifications using CLP. In Proc. of Int. Conf. on *Tools and Algorithms for the Construction and Analysis of Systems, Tool session (TACAS'05)*, Edinburgh, United Kingdom, pp. 551–556.

[Bouquet, 2002] Bouquet, F. and Legeard, B. and Peureux, F. 2002. CLPS-B – A constraint solver for B. Proc. of Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02), Grenoble, France, pp. 188–204.

[Boyapati, 2002] Boyapati, C. and Khurshid, S. and Marinov, D. 2002. Korat : Automated Testing based on Java Predicates. In Proc. of *ACM SIGSOFT international symposium on Software testing and analysis*, New York, USA, pp. 123–133.

[Burdy, 2003] Burdy, L. and Requet, A. and Lanet, J.-L. 2003. Java Applet Correctness : a Developer-Oriented Approach. In Proc. Int. Conf. Formal Methods *FME'03*, Pisa, Italy, pp. 422–439.

[Cok, 2004] Cok,, D. R. and Kiniry, J. 2004 ESC/Java2 : Uniting ESC/Java and JML. In Proc. of Int. Workshop on *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, Marseille, France, pages 108–128.

[Colin, 2004] Colin, S. and Legeard, B. and Peureux, F. 2004. Preamble Computation in Automated Test Case Generation using Constraint Logic Programming. *The Journal of Software Testing, Verification and Reliability*.

[Khurshid, 2004] Khurshid, S. and Marinov, D. 2004. Testera : Specification-based testing of Java programs using SAT. *Automated Software Engineering*.

[Leavens, 1999] Leavens, G.T. and Baker, A.L. and Ruby, C. 1999. JML : A Notation for Detailed Design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pp. 175–188. Boston :Kluwer Academic Publishers.

[Leavens, 2002] Leavens, G. T. and Cheon, Y. and Clifton, C. and Ruby, C. and Cok, D. R. 2002. How the Design of JML Accomodates Both Runtime Assertion Checking and Formal Verification. Proc of Int. Symp. on Formal Methods for Components and Objects *FMCO*, Leiden, Netherlands, pp 262–284.

[Ledru, 2004] Ledru, Y. and du Bousquet, L. and Maury, O. and Bontron, P. 2004. Filtering tobias combinatorial test suites. In Proc. of Int. Conf. on Fundamental Approaches to Software Engineering, Barcelona, Spain, pp. 281–294.

[Marlet, 2002] Marlet, R. and Mesnil, C. 2002. Demoney : A demonstrative electronic purse - card specification. Trusted Logic.

[Meyer, 1997] Meyer, B. 1997. *Object-Oriented Software Construction*. ISE Inc., Santa Barbara :Prentice Hall.

[Oriat, 2005] Oriat, C. 2005. Jartege : A tool for random generation of unit tests for java classes. In Proc. of Int. Workshop on Software Quality, SOQUA 2005, Erfurt, Germany, pp. 242–256.

[Sun, 2000] Sun microsystems. 2002. *Java Card 2.1.1 Virtual Machine Specification*.