JML-Testing-Tools, un Animateur Symbolique de Spécifications JML

Fabrice Bouquet, Frédéric Dadeau, Bruno Legeard

Laboratoire d'Informatique de l'Université de Franche–Comté
16, route de Gray – 25030 Besançon cedex FRANCE
Email : {bouquet,dadeau,legeard}@lifc.univ-fcomte.fr

Résumé

Cet article décrit le fonctionnement d'un outil permettant l'animation symbolique de spécifications écrites en Java Modeling Language (JML) dans l'objectif de les valider. L'animation est réalisée en utilisant la Programmation Logique avec Contraintes. Un solveur de contraintes est ainsi utilisé pour représenter les états symboliques durant l'animation des spécifications orientées-objet. Cet outil est également capable de vérifier des propriétés à la volée durant une séquence d'exécution et d'exhiber ainsi des contre-exemples. De ce fait, il peut-être utilisé à des fins de validation et de vérification d'une spécification formelle écrite en JML.

Mots-clés : animation symbolique, Java Modeling Language, programmation logique avec contraintes, validation et vérification.

1 Introduction et motivations

La validation de modèles formels est un enjeu important dans le processus de développement d'un logiciel, et spécialement lorsque le modèle est destiné à la génération de tests. A l'heure actuelle, cette étape de validation s'effectue le plus souvent par l'animation du modèle, si le langage de modélisation le permet. Ainsi, le spécifieur a une plus grande confiance dans son modèle et en particulier dans son comportement. L'animation symbolique sous-entend la représentation symbolique d'états du système. Pour ce faire, notre approche est basée sur l'utilisation de solveurs de contraintes pour la gestion des valeurs des variables d'états du système. De ce fait, le spécifieur, autrefois requis pour choisir l'opération à exécuter et fixer les valeurs des paramètres n'a plus aujourd'hui qu'à choisir l'opération, laissant les valeurs des paramètres être contraintes par les pré-conditions de l'opération considérée.

L'arrivée récente des langages de modélisation orientés-objet plus ou moins formels, comme UML/OCL [6] ou JML [4] offrent de nouvelles possibilités aux spécifieurs, notamment au niveau de la collaboration entre objets, de la modularité ainsi que de la réutilisation des modèles écrits.

Nous présentons un outil, nommé JML-Testing-Tools (JML-TT), permettant de valider des spécifications écrites en JML en réalisant l'animation symbolique du modèle. Cet outil s'appuie sur un moteur de résolution de contraintes ensemblistes et numériques pour traiter les états du système au cours de l'animation. JML-TT est basé sur la technologie de l'animateur BZ-Testing-Tools [1], adaptée pour la manipulation et le traitement des objets [2].

2 Description de JML

Le Java Modeling Language (JML) [4] est le langage de spécification associé au langage de programmation Java, basé sur le concept de la conception par contrat (Design By Contract [5]). Il s'agit d'un langage d'annotations qui s'insère à l'intérieur du code Java pour décrire le comportement des classes, comme illustré par la figure 1. JML offre deux types de vues du système, décrit sous forme de clauses, appliquées à une classe et portant sur les attributs de cette classe. Les clauses dites *statiques*, comme la clause d'invariant (`invariant`) ou les contraintes historiques (`constraints`), décrivent les propriétés du système qui s'appliquent à la classe. À l'inverse, les clauses dites *dynamiques* sont utilisées pour décrire le comportement des différentes méthodes. Ces dernières permettent de décrire un large éventail de spécificités comme les pré-conditions (clause `requires`), les post-conditions normales (clause `ensures`) décrivant la post-condition que le système établit quand la méthode termine sans déclencher d'exception, ou encore les post-conditions exceptionnelles (clause `signals`) qui décrivent la post-condition que le système établit lorsque la méthode termine en déclenchant une exception. À cela s'ajoute la clause `assignable` qui donne la liste des attributs et des objets modifiés par l'exécution de la méthode. Les post-conditions sont exprimées à l'aide de prédicats avant/après dans lesquels les expressions à évaluer à l'état d'avant sont entourées par `\old`. L'appel de méthodes est autorisé dans la syntaxe des prédicats, à la condition que ces méthodes soient "pures" au sens de JML, c'est-à-dire sans effets de bord.

```
class Purse {

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

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

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

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

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

}
```

FIG. 1 – Un exemple de spécification JML

3 Animation symbolique de spécifications JML

L'animation symbolique diffère de l'animation "traditionnelle" par l'utilisation de solveurs de contraintes pour gérer les états du système lors de l'exécution. Les avantages de l'utilisation des contraintes pour l'animation sont triples. Elles permettent (*i*) de conserver le non-déterminisme des données, (*ii*) de donner le choix à l'utilisateur d'instancier les paramètres ou de laisser contraints, et (*iii*) de diminuer la taille du graphe d'atteignabilité. En revanche l'utilisation des contraintes requiert des variables à domaines finis de manière à permettre l'énumération des solutions satisfaisant un store de contraintes.

L'évolution du système est décrite en utilisant des prédicats avant/après dans lesquels les variables d'état du système à l'état d'après sont primées. Ainsi, exécuter une transition t décrite sous la forme d'un prédicat avant-après $prd_t(V, V')$ entre deux états du système revient à résoudre un problème de satisfaction de contraintes.

$$\langle C(V), V\rangle \xrightarrow{t} \langle C(V \cup V') \cup prd_t(V, V'), V \cup V'\rangle$$

où V désigne le vecteur des variables d'état de la spécification et V' ce même vecteur à l'état d'après.

Pour animer une spécification, nous effectuons un découpage des différentes opérations en comportements. Chaque comportement correspond à une conjonction de littéraux représentant un chemin d'exécution possible dans le graphe de flot de contrôle de la spécification. Dans le cas où cette conjonction est inconsistante ou que le comportement ne puisse pas s'appliquer, le système de contraintes résultant est inconsistant. Dans le cas contraire, la transition –le comportement– a réussi à être activé et le système est alors dans l'état suivant. Les comportements extraits des transitions de la spécification sont représentés sous forme de graphes, sur lesquels des optimisations sont réalisées dans le but d'accélérer le processus d'évaluation.

Les spécifications de méthodes JML introduisent des notions de comportements au niveau des types de terminaison des méthodes (normales ou exceptionnelles), en plus des comportements JML standards séparés par le mot-clé `also`, comme illustré par la figure 2.

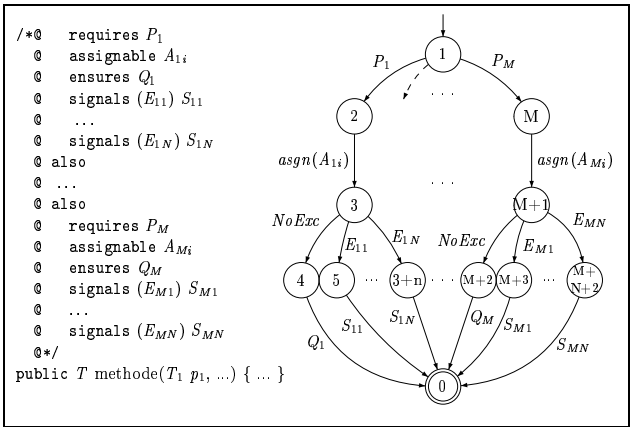

FIG. 2 – Extraction des comportements à partir d'une spécification de méthode JML

Dans cette figure, les P_i désignent les pré-conditions des comportements, Q_i les post-conditions normales lorsqu'aucune exception ($noExc$) n'est déclenchée, et S_{ij} les post-conditions exceptionnelles, lorsque l'exception E_{ij} est déclenchée. Pour finir, $asgn(A_i)$ désigne le traitement qui consiste à affecter aux attributs non modifiés par la méthode leur valeur "avant". Les contraintes correspondant à chacune des branches d'exécutions du graphe, i.e., à chacun des comportements de la méthode, sont ensuite posées une par une dans le store de contraintes. Une fois cette opération terminée, si le store est toujours consistant, le comportement est dit "activé" et le système est alors dans l'état suivant. Un système de "backtracking" permet d'échouer dans l'activation du comportement courant et de remonter au dernier point de choix pour tenter d'activer le comportement suivant.

4 Fonctionnalités de l'animateur JML-TESTING-TOOLS

Un aperçu de la fenêtre principale de l'animateur est donné en figure 3. La fenêtre se décompose en 4 parties distinctes. La partie gauche donne une vue de l'état mémoire du système, décomposé suivant les classes existantes. La partie en haut à droite affiche la séquence d'instructions correspondant à l'animation en cours. La partie droite centrale indique les variables qui peuvent être manipulées par l'utilisateur. Celles-ci sont classées en deux catégories : les variables de type prédéfini et les variables objets. Finalement, la partie en bas à droite indique le résultat des vérifications

de propriétés, comme l'invariant, effectuées à la demande de l'utilisateur. L'interface donne la possibilité d'effectuer les actions suivantes. Un objet peut être créé en sélectionnant un constructeur à partir d'une classe spécifique. L'objet ainsi créé doit être nommé et il apparaît dans la liste des objets disponibles.

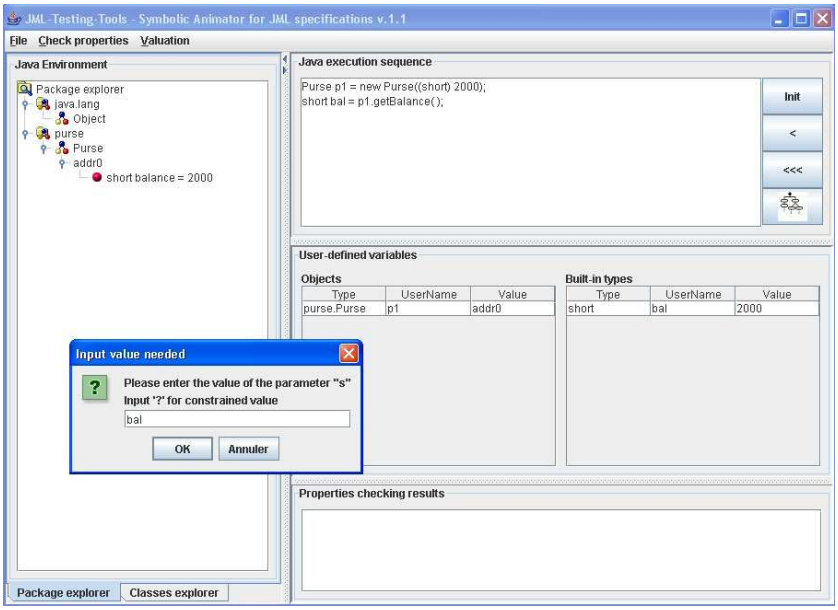

FIG. 3 – La fenêtre principale de JML-TT

L'utilisateur peut alors choisir la méthode qu'il souhaite invoquer sur l'objet. Des boîtes de dialogues apparaissent alors pour lui demander le comportement qu'il souhaite activer, en fonction de l'exception déclenchées et pour saisir les valeurs des paramètres d'entrée de la méthode. Ces derniers peuvent être laissés non spécifiés et sont alors contraints par le comportement choisi. Les paramètres contraints peuvent par la suite être instanciés à la demande de l'utilisateur, incluant éventuellement la pose de contraintes supplémentaires, telle que l'invariant de classe. Il est également possible de revenir à l'état initial, de faire un ou trois pas en arrière, ou encore d'afficher l'arbre des exécutions du système. Une séquence d'animation instanciés peuvent être exportées de manière à produire un cas de test Java basé sur la spécification. Dans le cas d'une séquence d'animation contenant des valeurs contraintes, ceux-ci sont instanciés en combinant les différentes valeurs aux limites des domaines des paramètres. Ceci créé un mécanisme de génération de cas de tests à partir d'un modèle, selon le principe du test combinatoire.

Les fonctionnalités de l'animateur JML-TESTING-TOOLS sont les suivantes :

- Animation symbolique à contraintes des spécifications JML pour la validation du modèle par activation des comportements du système ;
- Possibilité de laisser les paramètres des méthodes contraints par le comportement choisi pour une instanciation ultérieure ;
- Activation des différents comportements des méthodes vis-à-vis de la spécification ;
- Possibilité de vérification des propriétés JML (invariant et contraintes historiques) à la volée pour un état donné de l'animation ;

- Génération de tests Java exécutables à partir d'une séquence d'animation définie à travers l'animateur pour la vérification d'assertions à la volée [3].

5 Conclusion et perspectives

Nous avons développé et présenté un outil de validation d'une spécification JML par animation symbolique. Ce travail, réalisé dans le cadre du projet ACI GECCOO[1], est préliminaire à la génération automatique de tests aux limites pratiquée dans le cadre la méthode BZ-Testing-Tools [1]. L'étape suivante est donc l'utilisation de ce moteur d'animation pour calculer des séquences de tests sur des objets aux limites. Parallèlement, nous envisageons d'utiliser ce moteur d'animation pour la détection automatique de non-conformités entre un programme et sa spécification, par comparaison deux systèmes de contraintes, l'un issu du code et l'autre issu de la spécification. Ce travail est effectué au sein du projet RNTL DANOCOPS[2].

L'animateur symbolique JML-Testing-Tools est diffusé à partir du site du LIFC, à l'adresse : `http://lifc.univ-fcomte.fr/~jmltt/`.

Références

[1] F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, N. Vacelet, and M. Utting. BZ-TT : A Tool-Set for Test Generation from Z and B using Contraint Logic Programming. In Robert Hierons and Thierry Jerron, editors, *Formal Approaches to Testing of Software, FATES 2002 workshop of CONCUR'02*, pages 105–120. INRIA Report, August 2002.

[2] F. Bouquet, F. Dadeau, B. Legeard, and M. Utting. Symbolic Animation of JML Specifications. In *Proceedings of the International Conference on Formal Methods (FM'05)*, volume 3582 of *Lecture Notes in Computer Science*, pages 75–90, Newcastle, United Kingdom, July 2005. Springer-Verlag.

[3] Yoonsik Cheon and Gary T. Leavens. A Runtime Assertion Checker for the Java Modeling Language (JML). In Hamid R. Arabnia and Youngsong Mun, editors, *Proceedings of the International Conference on Software Engineering Research and Practice (SERP '02), Las Vegas, Nevada, USA, June 24-27, 2002*, pages 322–328. CSREA Press, June 2002.

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

[5] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 2 edition, 1997.

[6] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*, addison-wesley edition, 1999.

[1] `http://geccoo.lri.fr`

[2] `http://lifc.univ-fcomte.fr/~danocops`