JAG : Génération d'annotations JML pour vérifier des propriétés temporelles*

Alain Giorgetti et Julien Groslambert
LIFC - Université de Franche-Comté
16 route de Gray - 25030 Besançon CEDEX - France

{giorgetti,groslambert}@lifc.univ-fcomte.fr

Résumé

Cet article présente un outil de vérification de propriétés de sécurité pour du code Java annoté en JML. Ces propriétés temporelles, exprimées dans une extension de JML, sont transformées par l'outil en annotations JML standard et sont intégrées automatiquement dans la classe Java à vérifier.

Mots-clés : Java Modeling Language, logique temporelle, annotations, vérification.

1 Motivations

Nous présentons dans cet article l'outil JAG (pour *JML Annotation Generator*) qui permet, à partir d'une formule exprimée dans un langage de logique temporelle [4, **?**], et d'un fichier Java/JML, de générer les annotations JML (Java Modeling Language) appropriées à la vérification de cette propriété.

Supposons qu'on souhaite vérifier la sécurité d'un système de transaction. La figure 1 présente un extrait d'une classe Java qui implante un tel système proche de celui de la JavaCard. Le code Java y est enrichi par des annotations JML (Java Modeling Language) précédées par `//@` sur une ligne ou délimitées par `/*@` et `@*/`.

Dans cette classe, une méthode `beginTransaction()` démarre une nouvelle transaction et une méthode `commitTransaction()` la valide. Durant la transaction, une méthode `modify()` écrit dans un buffer. La transaction peut être abandonnée par invocation d'une méthode `abortTransaction()`. Des annotations JML avant l'entête de ces méthodes Java permettent d'en décrire le comportement.

Les principales annotations JML sont des invariants (clause `invariant`) décrivant une propriété que tous les états de la classe doivent vérifier, des préconditions de méthode (clause `requires`) et des post-conditions de méthode (clause `ensures`) qui indiquent une propriété qui doit être vraie quand la méthode termine. Les prédicats JML suivent la syntaxe des expressions booléennes Java, étendue avec des mots-clefs spécifiques à JML, comme `\old`, qui fait référence à la valeur d'une variable dans l'état précédent. Des variables JML (`ghost` et `model`) peuvent être déclarées et mises à jour (clauses `set` et `initially`).

Sur cette classe Java, on souhaite vérifier deux propriétés de bon fonctionnement du système de transaction : (i) Le buffer doit être vidé avant de commencer une nouvelle transaction et (ii) toute transaction commencée doit fatalement se terminer. La première

*Recherche partiellement financée par l'ACI sécurité *Geccoo*.

```
package example.transacSystem;
public class TransactionSystem {
...
//@ ghost boolean TrDepth = false;
//@ model int BufferFree;
//@ initially BufferFree == 10;
//@ invariant BufferFree <= 10;

/*@ private  normal_behavior
   @ requires TrDepth == false;
   @ assignable TrDepth;
   @ ensures TrDepth == true;
   @*/
public void beginTransaction() {
//@ set TrDepth = true;
...
};

...
}
```

Fig. 1 – Extrait d'une spécification JML : un système de transaction

propriété est une propriété de sûreté ("quelque chose de mauvais ne doit pas arriver") et la seconde est une propriété de vivacité ("quelque chose de bon doit nécessairement arriver").

Ces propriétés de sécurité peuvent être codées comme des restrictions sur les séquences d'exécution de méthodes Java. Toutefois, il n'est pas aisé de les traduire directement en un ensemble d'annotations JML. C'est pourquoi nous proposons un outil qui met à la disposition du spécifieur un langage compact pour exprimer de telles propriétés et automatise leur traduction en annotations JML, directement insérées dans le code de la classe à vérifier. Dans l'extension de JML proposée dans [4], ces deux propriétés peuvent s'exprimer comme suit.

(i) `after commitTransaction() terminates, abortTransaction() terminates`
`before beginTransaction() called always BufferFree == 10 ;`

(ii) `after beginTransaction() called always {True}`
`until abortTransaction() called, commitTransaction() called`
`under_invariant {true} variant {BufferFree} ;`

La première formule signifie qu'après (`after`) la fin d'une transaction, par validation (commit) ou par rollback (abort), et avant (`before`) qu'une autre transaction ne commence, le buffer doit toujours (`always`) être vide. La seconde formule traduit qu'après (`after`) qu'une transaction ait commencé celle-ci doit fatalement (`until`) se terminer par un rollback ou une validation. On fournit à l'outil un variant pour vérifier cette vivacité. Ce variant est un entier naturel qui doit décroître strictement à chaque appel de méthode, pour garantir qu'on finira par atteindre l'état voulu.

2 Description de l'outil

Principe de fonctionnement de l'outil La structure de l'outil est illustrée dans la figure 2. Le fichier Java/JML d'entrée est analysé syntaxiquement par l'outil JMLTools [2]. Les propriétés temporelles sont analysées avec un analyseur généré avec le même outil (ANTLR) que celui de JMLTools, afin de faciliter une intégration ultérieure.

L'outil réduit tout d'abord chaque formule temporelle à vérifier en une ou plusieurs primitives intermédiaires qui lui sont sémantiquement équivalentes [1, 4]. La primitive Inv représente la partie sûreté d'une propriété temporelle, la primitive Loop représente sa partie vivacité et la primitive Witness représente un état particulier du système ou de l'exécution. Ces primitives optimisent l'étape suivante de génération d'annotations JML équivalentes.

Génération d'annotations JML à partir des primitives Chaque primitive Inv est traduite en un invariant JML. Chaque primitive Loop est traduite en un ensemble d'invariants et de contraintes historiques qui impliquent la décroissance du variant et l'absence

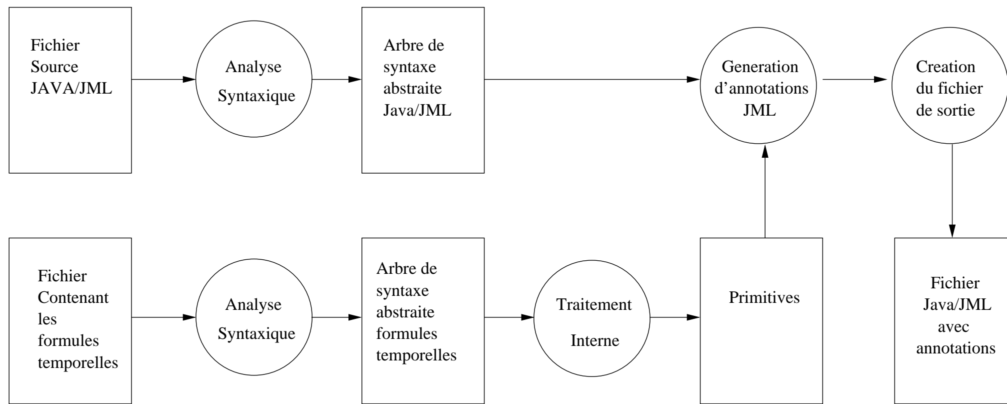

FIG. 2 – Schéma général de l'outil.

```
/*@ ghost public boolean
  @  gTLbeginTransactioncalled_2 = false;
  @ ghost public boolean
  @  gTLcommitTransactionterminates_1 = false;
  @ ghost public boolean
  @  gTLabortTransactionterminates_1 = false;
  @*/

/*@  invariant
  @    gTLcommitTransactionterminates_1
  @ || gTLcommitTransactionterminates_1
  @               ==> BufferFree == 10;
  @*/

//@  invariant BufferFree>= 0;

/*@  constraint
  @   \old(gTLbeginTransactioncalled_2)
  @       ==> (BufferFree) < \old(BufferFree);
  @*/
```

```
/*@  constraint
  @   \old(gTLbeginTransactioncalled_2)
  @     ==> (( TrDepth == false)
  @      || ( BufferFree == 0 && TrDepth == true)
  @      || ( BufferFree > 0 && TrDepth == true)
  @      || ( BufferFree > 0 && TrDepth == true));
  @*/

boolean TrDepth=false;
int BufferFree=10;

/*@ private normal_behavior
  @ requires TrDepth == false;
  @ assignable TrDepth;
  @ ensures TrDepth == true;
  @*/
public void beginTransaction() {
//@ set gTLbeginTransactioncalled_2 = true;
//@ set gTLcommitTransactioncalled_1 = false;
//@ set gTLabortTransactioncalled_1 = false;
...
}
```

FIG. 3 – Extrait d'un fichier de sortie de l'outil JAG.

de blocages dans le système. Chaque primitive Witness est traduite par une variable `ghost`.

L'outil produit en sortie un fichier dans lequel la classe originale est complétée avec les annotations JML générées. La figure 3 présente un extrait[1] de la spécification obtenue pour la vérification des propriétés (i) et (ii) sur le système de transaction. On peut constater que des variables `ghost` sont ajoutées pour "observer" les appels et terminaisons des méthodes impliquées dans la formule temporelle. Le premier invariant assure la satisfaction de la propriété de sûreté (i). Le second assure que le variant de la propriété de vivacité (ii) est bien un entier naturel. Les deux contraintes historiques (`constraint`) assurent respectivement que ce variant diminue après chaque appel de méthode jusqu'à la terminaison de la transaction et qu'il n'y a pas de blocage avant la terminaison de la transaction (au moins une des préconditions de méthode est vraie).

Ces annotations peuvent alors être validées ou prouvées à l'aide des outils disponibles pour JML [2].

[1] Le fichier complet est disponible depuis la page `http://lifc.univ-fcomte.fr/~groslambert/JAG/`

Nom de l'exemple	Nombre de propriétés temporelles à vérifier	Nombre d'annotations générées	Nombre d'OP (dont automatiques)
TransactionSystem	2	18	92 (91)
AtmTransaction	2	21	171 (171)

Tab. 1 – Résultats expérimentaux.

Traçabilité des annotations générées Une traçabilité des annotations JML générées a été implantée dans l'outil JAG. L'interface graphique de JAG permet :

- A partir d'une annotation générée, de retrouver la primitive et la propriété temporelle d'origine.
- D'obtenir des informations sur ce que vérifie chacune des annotations (décroissance de variant, observation d'appel de méthode...)

Cette caractéristique permet un diagnostic plus aisé en cas d'échec de la vérification d'une annotation JML.

3 Résultats expérimentaux

L'outil a été validé sur différents exemples, dont celui présenté dans cet article. La table 1 résume les résultats obtenus en utilisant l'outil Jack [3] pour générer et vérifier en aval les obligations de preuve (OP).

4 Conclusion et travaux futurs

Nous avons présenté un outil de génération d'annotations JML assurant la satisfaction de propriétés temporelles exprimées dans le langage défini dans [4]. La particularité et l'un des atouts de ce travail est de générer des annotations JML standard, l'ensemble des outils JML peuvent donc être utilisés sur le fichier de sortie. L'un des premiers travaux que nous envisageons est d'intégrer notre outil avec ceux utilisés en aval. Nous envisageons également d'étendre notre démarche de génération d'annotations à d'autre langages d'entrée, en particulier à la PLTL.

L'outil et sa documentation sont disponibles en téléchargement à l'adresse `http://www.lifc.univ-fcomte.fr/~groslambert/JAG`.

Références

[1] F. Bellegarde, J. Groslambert, M. Huisman, O. Kouchnarenko, and J. Julliand. Verification of liveness properties with JML. Technical report, INRIA, 2004.

[2] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An overview of JML tools and applications. In Th. Arts and W. Fokkink, editors, *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*, volume 80 of *ENTCS*, pages 73–89. Elsevier, 2003.

[3] L. Burdy and A. Requet. Jack : Java applet correctness kit. In *Gemplus Developers Conference GDC2002*, 2002.

[4] K. Trentelman and M. Huisman. Extending JML Specifications with Temporal Logic. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methodology And Software Technology (AMAST'02)*, number 2422 in LNCS, pages 334–348. Springer, 2002.