Un programme annoté en vaut deux*

A. Giorgetti & J. Groslambert

LIFC - FRE CNRS 2661
Université de Franche-Comté
16 route de Gray
25000 Besançon cedex, France
{giorgetti,groslambert}@lifc.univ-fcomte.fr

Résumé

La sécurité du logiciel passe par les méthodes formelles, mais comment faire passer les méthodes formelles dans le monde réel du développement des logiciels critiques ? Cet article propose un mécanisme et un outil d'aide à la vérification de code, par réduction automatique de propriétés de sécurité en annotations formelles. Ce dispositif s'applique aux programmes Java annotés en JML. Il est illustré ici par l'annotation automatique d'une classe de l'API Java Card. Deux formalismes complémentaires pour l'expression de propriétés sont proposés. L'outil JAG, qui implante leur réduction en annotations JML, est décrit en détail. Le langage d'annotations retenu (JML) étant standard, cet outil s'interface naturellement avec de nombreux outils existants pour la vérification par preuve, test ou model-checking de Java annoté en JML.

Mots-clés : logiciel sûr, Java Modeling Language, propriétés temporelles, annotations, vérification.

1. Introduction

Le logiciel devient de plus en plus incontournable dans des domaines où la sécurité est primordiale, comme les transports, les communications, l'aéronautique, l'automobile ou encore la médecine. Il est donc indispensable d'accompagner son développement avec des méthodes renforcées de vérification du code exécuté. Depuis quelques années, le langage d'annotations JML [21] enrichit dans ce but le langage de programmation objet Java. Il permet d'accroître les possibilités de vérification de la sûreté du code source, ou du bytecode généré par le compilateur Java. Cependant, exprimer directement les conditions de sûreté d'un programme Java sous forme d'annotations JML est fastidieux, car les constructions de ce langage sont trop élémentaires. Dans cet article, nous expliquons comment générer ces annotations JML à partir de propriétés de sécurité de haut niveau, de manière automatique.

C'est dans le logiciel pour carte à puce que cette démarche a trouvé ses premières applications. Pour en rendre compte, nous en illustrons tous les aspects, de l'expression de propriétés jusqu'à la production d'annotations, avec l'exemple de la classe APDU (Application Protocol Data Unit) de l'API Java Card.

L'article s'organise comme suit. La partie 2 présente l'exemple de la classe APDU et le fragment du langage JML utilisé pour traduire les propriétés en annotations. La partie 3 définit et compare deux formalismes complémentaires pour l'expression de propriétés de sécurité. Puis la partie 4 décrit la conception de l'outil JAG (Java Annotation Generator) qui implante cette traduction, première étape pour vérifier que le code Java est conforme à ces propriétés de sécurité. Le mode d'emploi de cet outil est donné dans la partie 5. Enfin, la partie 6 conclut et présente des perspectives de travaux futurs.

* Partiellement financé par l'ACI SI GECCOO.

2. Exemple applicatif et présentation de JML

La figure 1 présente un extrait de la classe `javacard.framework.APDU` de la plateforme Java Card 2.1. Cette classe fait partie de l'API (Application Programming Interface) Java Card, définie par Sun Microsystems. Son interface est décrite dans sa documentation au format javadoc[1], mais son code source n'est pas diffusé.

Le JCRE (Java Card Runtime Environment) utilise une instance de la classe APDU (Application Protocol Data Unit) comme support de communication entre la carte à puce et les terminaux qui l'accueillent. Plus précisément, l'information est lue et écrite dans le buffer de l'APDU. La méthode `getBuffer()` permet d'accéder à ce buffer. Les données échangées pouvant être plus longues que la capacité du buffer, la classe APDU est organisée en méthodes qui fragmentent leur transfert. Ainsi, les méthodes `setIncomingAndReceive(...)` et `receiveBytes(...)` collaborent à leur réception, tandis que les méthodes `setOutgoing ...(...)` et `sendBytes(...)` organisent la réponse de la carte.

Dans la figure 1, nous avons enrichi l'interface de la classe avec des annotations JML [21] (Java Modeling Language). Il s'agit de toutes les lignes précédées par `//@` et de tous les blocks délimités par `/*@` et `@*/`. Les annotations JML avant l'entête des méthodes Java permettent d'en décrire le comportement. Ces annotations formalisent une partie des informations données en langue naturelle dans la documentation HTML de la classe APDU.

Les principales annotations JML sont des invariants (clause `invariant`) décrivant une propriété que tous les états *visibles* 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 se termine. Les états *visibles* sont les états précédant l'appel d'une méthode de la classe et les états de terminaison des méthodes de la classe. 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 visible précédent. Des variables JML (`ghost`) peuvent être déclarées et mises à jour (clauses `set`). Le mot clef `pure` permet de spécifier qu'une méthode est sans effet de bord. De telles méthodes peuvent être utilisées à l'intérieur d'un prédicat JML.

```
package javacard.framework;

public class APDU {
  //@ invariant getBuffer().length >= 37;
  //@ ghost int bufferPosition = 0;

  byte[] /*@ pure @*/ getBuffer() { ... }

  /*@ normal_behavior
    @ requires bufferPosition == 0;
    @ ensures bufferPosition == bOff - 1
    @   || bufferPosition == getBuffer().length
    @*/
  short receiveBytes(short bOff) {
    ...
    le = getBuffer().length;
    /*@ set bufferPosition =
      @   (bOff > le ? le : bOff) - 1;
      @*/
    ...
  }

  /*@ normal_behavior
    @ requires bOff + len <= getBuffer().length;
    @*/
  void sendBytes(short bOff, short len) {
    ...
  }

  short setIncomingAndReceive() { ... }

  short setOutgoing() { ... }

  void setOutgoingAndSend(short bOff, short len) {
    ...
  }

  void setOutgoingLength(short len) { ... }

  short setOutgoingNoChaining() { ... }
}
```

FIG. 1 – Extrait de la classe APDU annotée en JML.

[1] `http://java.sun.com/products/javacard/htmldoc/javacard/framework/APDU.html`

2.1. Propriétés à vérifier

Sur la classe APDU, on souhaite vérifier deux propriétés de bon fonctionnement, exprimées informellement en anglais par Sun Microsystems dans la documentation HTML de la classe : (i) un appel à la méthode `setIncomingAndReceive()` déclenche une exception si elle a déjà été appelée auparavant avec succès, et (ii) la méthode `setOutgoingLength(...)` ne peut pas être appelée à moins que la méthode `setOutgoing()` n'ait été appelée avec succès auparavant.

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 mettons à la disposition du spécifieur des langages compacts pour exprimer de telles propriétés, et un mécanisme de traduction automatique de ces propriétés en annotations JML, directement insérées dans le code de la classe à vérifier.

3. Comment formaliser des propriétés de sécurité

Nous proposons ici un langage de formules temporelles et une structure d'automates pour formaliser des propriétés de sécurité. L'un et l'autre sont issus d'une analyse des besoins les plus fréquents des développeurs Java, en matière d'expression de la sécurité de leur code. Chaque formalisme est illustré par des exemples de propriétés pour la classe APDU.

3.1. Des schémas pour exprimer les propriétés

Afin d'aider les programmeurs Java à écrire des annotations JML, Trentelman et Huisman [26] ont proposé une extension temporelle de JML, inspirée des schémas de spécification (Specification Pattern) [12] du projet Bandera. Une des motivations de ce projet était d'aider le spécifieur à écrire des propriétés temporelles, en lui fournissant des schémas pour les propriétés les plus courantes. A travers une étude de plus de 500 exemples de spécifications, M. Dwyer et son équipe ont montré que 80% des besoins de spécification pouvaient être couverts par un nombre restreint de schémas de propriété.

Le langage défini par Huisman et Trentelman, que nous proposons de nommer JTPL (pour *Java Temporal Pattern Language*, langage de schémas temporels pour Java) suit l'approche des schémas de spécification de Dwyer. L'une de ses spécificités est de tenir compte de la terminaison exceptionnelle de Java, qui survient lorsque l'exécution d'un programme Java lève une exception, qui peut éventuellement être capturée pour poursuivre l'exécution du programme. Nous verrons comment le langage JTPL permet d'exprimer facilement des propriétés relatives à ces levées d'exception.

La sémantique formelle de ce langage est détaillée dans [26], ainsi que la traduction des propriétés de sûreté. La traduction des propriétés de vivacité est expliquée dans [2].

3.1.1. Présentation du langage

Dans cette partie, nous présentons informellement le langage JTPL. Celui-ci est basé sur la notion de propriété d'état. Une propriété d'état est un prédicat JML, m **enabled** ou m **not enabled**, où m désigne une méthode Java de la classe. La structure m **enabled** (resp. **not enabled**) définit implicitement l'ensemble des états à partir desquels une invocation de la méthode m ne déclenchera pas d'exception (resp. déclenchera obligatoirement une exception).

Chaque propriété d'état P doit être encapsulée dans une propriété de trace qui peut être **always** P, **eventually** P, ou encore la conjonction ou la disjonction de deux propriétés de trace. La sémantique de ces propriétés de trace est la suivante :
- **always** P est vraie sur une exécution σ si P est satisfaite sur chaque état de σ.
- **eventually** P est vraie sur une exécution σ si P est satisfaite sur au moins l'un des états de σ.
- la sémantique de la conjonction et de la disjonction est standard.

always P	$\mathsf{G}P$
after E always P	$\mathsf{G}(E \Rightarrow \mathsf{G}P)$
before E always P	$(\mathsf{F}E) \Rightarrow (P\mathsf{U}E)$
always P until E	$P\mathsf{U}E$
always P unless E	$(P\mathsf{U}E) \vee \mathsf{G}P$
after E_1 always P until E_2	$\mathsf{G}(E_1 \Rightarrow (P\mathsf{U}E_2))$
after E_1 always P unless E_2	$\mathsf{G}(E_1 \Rightarrow ((P\mathsf{U}E_2) \vee \mathsf{G}P))$
eventually P	$\mathsf{F}P$
after E eventually P	$\mathsf{G}(E \Rightarrow \mathsf{F}P)$
before E eventually P	$\mathsf{F}E \Rightarrow \neg(\neg P\mathsf{U}E)$
eventually P until E	$\mathsf{F}E \wedge \neg(\neg P\mathsf{U}E)$
eventually P unless E	$(\mathsf{F}E \wedge \neg(\neg P\mathsf{U}E)) \vee (\neg\mathsf{F}E \wedge \mathsf{F}P)$
after E_1 eventually P until E_2	$\mathsf{G}(E_1 \Rightarrow (\mathsf{F}E_2 \wedge \neg(\neg P\mathsf{U}E_2)))$
after E_1 eventually P unless E_2	$\mathsf{G}(E_1 \Rightarrow ((\mathsf{F}E_2 \wedge \neg(\neg P\mathsf{U}E_2)) \vee (\neg\mathsf{F}E_2 \wedge \mathsf{F}P)))$

FIG. 2 – Traduction de quelques formules JTPL en LTL.

Il est bien souvent utile de réduire la portée d'une propriété de trace à une sous-séquence de l'exécution. Ceci est rendu possible en JTPL par la notion d'*événement*.

Un événement peut être :

- m **called**, qui signifie que la méthode m vient d'être appelée, i.e., a été appelée dans l'état précédant immédiatement l'état courant.
- m **normal**, qui signifie que la méthode m a terminé normalement, c'est-à-dire sans lever d'exception.
- m **exceptional**, qui signifie que la méthode m a terminé en levant une exception.
- m **terminates**, qui signifie que la méthode m a terminé, en levant ou non une exception.

Enfin, une propriété JTPL est soit une propriété de trace, soit l'une des constructions suivantes, où E est un ensemble d'événements, C est une propriété JTPL et T est une propriété de trace :

- **after** E C, qui est satisfaite sur une exécution σ si chaque suffixe de σ commençant après une occurrence d'un événement de E satisfait la propriété C,
- **before** E T, qui est satisfaite sur une exécution σ si chaque préfixe de σ terminant par une occurrence d'un événement de E satisfait la propriété de trace T,
- T **until** E, qui est satisfaite sur une exécution σ si un événement de E apparaît inévitablement et si la propriété de trace T est satisfaite sur le segment de σ précédant cet événement de E,
- T **unless** E, qui est satisfaite sur une exécution σ si un événement de E a lieu et si la propriété de trace T est satisfaite sur le segment de σ précédant cet événement de E, ou si T est satisfaite sur toute l'exécution σ et si aucun événement de E n'a lieu.

La logique JTPL est une logique linéaire incluse dans la LTL (logique temporelle linéaire) [13]. Des exemples de traduction de formules JTPL en LTL sont donnés dans la figure 2. Dans cette figure, E désigne en JTPL un ensemble d'événements, tandis qu'il désigne en LTL le prédicat caractéristique des états qui succèdent immédiatement à l'un de ces événements.

Les schémas JTPL sont cependant strictement moins expressifs que les formules LTL sur les mêmes propriétés d'états. Par exemple, la propriété d'équité "*infiniment souvent P*", qui peut s'écrire $\mathsf{GF}P$ en LTL, n'est pas expressible en JTPL. En revanche, les schémas JTPL sont plus simples à comprendre et à utiliser que les formules LTL ou CTL (logique arborescente) [13], tout en couvrant les besoins les plus courants [12]. Par exemple, les propriétés JTPL **before** E **always** P et **after** E_1 **always** P **unless** E_2 s'écriraient respectivement $(\mathsf{F}E) \Rightarrow (P\mathsf{U}E)$ et $\mathsf{G}(E_1 \Rightarrow ((P\mathsf{U}E_2) \vee \mathsf{G}P))$ en LTL.

3.1.2. Exemples de propriétés temporelles

En JTPL, les deux propriétés de la partie 2.1 peuvent s'exprimer comme suit.

```
(i)   after setIncomingAndReceive() normal
          always setIncomingAndReceive() not enabled
(ii)  always setOutgoingLength() not enabled
          unless setOutgoing() normal
```

La première formule signifie qu'après (`after`) un appel réussi à la méthode `setIncomingAndReceive` (`setIncomingAndReceive() normal`), tout autre appel à cette méthode doit lever une exception (`always setIncomingAndReceive() not enabled`).

La seconde formule traduit que tout appel à la méthode `setOutgoingLength` lève une exception, à moins (`unless`) qu'un appel à la méthode `setOutgoing` n'ait réussi (`setOutgoing() normal`).

3.2. Vérification à base d'automates

Outre le langage JTPL, nous proposons la possibilité d'exprimer des propriétés temporelles à l'aide d'automates d'états finis. Ce format d'automates est inspiré d'une proposition d'un partenaire industriel, qui utilisait des diagrammes d'états UML pour exprimer le comportement de ses applications Java Card. Les automates étaient alors transformés manuellement en annotations JML. Cette traduction manuelle avaient comme défauts d'être d'une part fastidieuse et d'autre part source potentielle d'erreurs. Nous avons donc conçu une traduction automatique de ces automates en annotations JML.

Les automates en question sont des automates d'états finis dont les transitions sont étiquetées par des événements JTPL (m **called**, ..., m **terminates**). Nous illustrons cette approche avec la classe APDU de la Java Card, qui supporte quatre modes de transmission de message :
- NO DATA : pas de données en entrée, pas de données en réponse
- IN : données en entrée, pas de données en réponse
- OUT : pas de données en entrée, données en réponse
- IN/OUT : données en entrée et en réponse.

Chacun de ces modes correspond à une séquence particulière d'invocations de méthodes de la classe APDU. Ces séquences sont plus faciles à décrire par un automate, comme celui de la figure 3, que par un ensemble de formules JTPL.

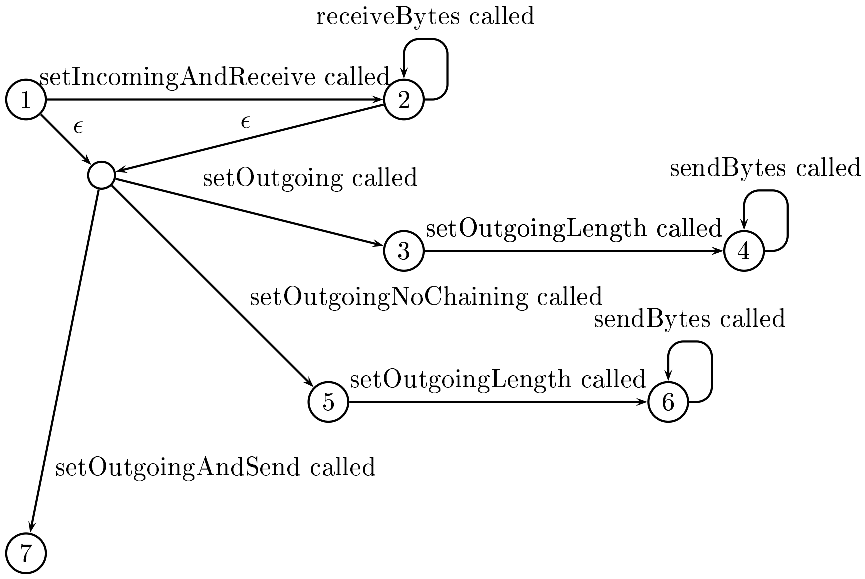

FIG. 3 – Automate de l'APDU.

Dans l'état initial (1), la méthode `setIncomingAndReceive` peut être invoquée pour entrer

```
never
T1_init:
  if
  :: (setIncomingAndReceive called)
        -> goto accept_S2
  :: (setOutgoing called)
        -> goto accept_S3
  :: (setOutgoingNoChaining called)
        -> goto accept_S5
  :: (setOutgoingAndSend called)
        -> goto accept_S7
  fi;
accept_S2:
  if
  :: (receiveBytes called)
        -> goto accept_S3
  fi;
...
```

FIG. 4 – Extrait du fichier de description Promela.

```
public class APDU {
  ...
  //@ ghost int APDUstate = 1;
  //@ invariant APDUstate >= 1;
  //@ invariant APDUstate <= 7;
  ...

  /*@ requires APDUstate == 1;
    @ assignable APDUstate;
    @ ensures APDUstate == 2;
    @*/
  void setIncomingAndReceive() {
    //@ set APDUstate = 2;
     ...
  }
  ...
}
```

FIG. 5 – Extrait de la sortie de JAG pour l'APDU.

dans le mode IN ou le mode IN/OUT. On peut alors invoquer la méthode `receiveBytes` pour lire la donnée d'entrée. Ensuite, la méthode `setOutgoingAndSend` (resp. `setOutgoing` et `setOutgoingNoChaining`[2]) peut être invoquée pour renvoyer une réponse sans données pour les modes IN et NO DATA (resp. avec données pour les modes OUT et IN/OUT). Quand des données de réponse doivent être envoyées (modes OUT et IN/OUT) la méthode `setOutgoingLength` est invoquée pour donner la longueur de la réponse. Enfin, la méthode `sendBytes` envoie la donnée de réponse.

Dans cet automate, ϵ (le mot vide) marque une transition sans événement. Ceci signifie qu'à partir de l'état 1, les méthodes `setIncomingAndReceive`, `setOutgoing`, `setOutgoingNoChaining` et `setOutgoingAndSend` peuvent être invoquées. Cette notation facilite encore l'écriture de tels automates.

L'intérêt des automates est double :

- D'un point de vue théorique, il existe des propriétés exprimables par automate qui ne sont pas expressibles en LTL (donc, a fortiori, en JTPL) [27]. Par exemple, la propriété "La méthode m est appelée dans chaque état indexé par un nombre impair" ne peut pas être exprimée en LTL.
- D'un point de vue pratique, certaines propriétés sont plus faciles à exprimer sous forme d'automate. Par exemple, l'automate de la figure 3 inclut, outre les propriétés JTPL (i) et (ii) de la partie 3.1.2, un grand nombre de propriétés similaires, autorisant ou interdisant certains enchaînements de méthodes.

4. JAG : un générateur d'annotations JML

Cette partie présente la version 0.2 de l'outil JAG (pour *JML Annotation Generator*) qui permet, à partir d'un fichier Java/JML et d'une propriété de sécurité formalisée en JTPL ou par un automate, de générer les annotations JML (Java Modeling Language) appropriées à la vérification de cette propriété. Le code JML produit étant standard, tous les outils existants [8] pour la validation, la vérification algorithmique ou la preuve de code Java annoté en JML peuvent

[2] `setOutgoingNoChaining` remplace `setOutgoing` quand le terminal de la carte ne supporte pas un mode particulier, appelé *block chaining*. Ces considérations techniques n'entrent pas en jeu dans la compréhension de l'exemple.

être utilisés en aval de JAG.

4.1. Principe de fonctionnement

La structure de l'outil est illustrée par la figure 6. Le fichier Java/JML d'entrée est analysé syntaxiquement par l'outil JMLTools [11]. Les propriétés temporelles JTPL sont analysées avec un analyseur syntaxique produit avec le même générateur (ANTLR[3]) que celui de JMLTools, afin de faciliter une intégration ultérieure.

La version actuelle (0.2) de JAG permet de décrire les automates de la partie 3.2 dans le langage Promela [20]. Un extrait de la description Promela de l'automate de l'APDU de la figure 3 est donné dans la figure 4. La prochaine version de JAG permettra à l'utilisateur de dessiner son automate dans un environnement graphique de conception de diagrammes d'états UML. Elle comportera un traducteur vers ce format Promela. L'outil JAG génère automatiquement les variables ghost qui encodent l'automate. Un extrait du fichier de sortie obtenu pour la classe APDU est donné dans la figure 5.

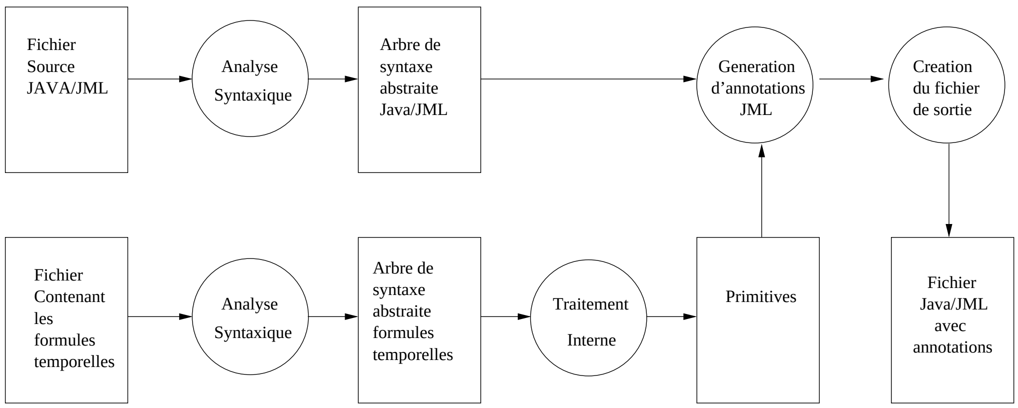

FIG. 6 – Schéma général de l'outil.

Le processus de génération des annotations JML a déjà été décrit dans des publications précédentes [2, 26, 19]. L'outil réduit tout d'abord chaque formule temporelle à vérifier en une ou plusieurs primitives intermédiaires qui lui sont sémantiquement équivalentes. 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. L'utilisation de ces primitives permet ensuite de factoriser le travail de génération d'annotations. L'utilisation de primitives pourra également permettre d'envisager des extensions futures à l'outil JAG. Une adaptation aux systèmes d'événements B existe déjà [17, 18] et une extension au langage d'annotations SPEC♯ [1] est en cours de développement.

4.2. Génération d'annotations JML à partir des primitives

Chaque primitive Inv est traduite en un invariant JML ou en une postcondition de méthode. Chaque primitive Loop est traduite en un ensemble d'invariants et de contraintes historiques qui impliquent la décroissance du variant et l'absence de blocages. 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 7 présente un extrait[4] de la spécification obtenue pour la

[3] `http://www.antlr.org`
[4] Le fichier complet est disponible depuis la page `http://lifc.univ-fcomte.fr/~jag`.

```
/*@ ghost public boolean
  @  gTLsetIncomingAndReceive_normal = false;
  @ ghost public boolean
  @  gTLsetOutgoing_normal = false;
 @*/

/*@ private normal_behavior
  @ ...
  @ ensures ! \old(gTLsetIncomingAndReceive_normal);
  @*/
public short setIncomingAndReceive() {
...
//@ set gTLsetIncomingAndReceive_normal = true;
}
```

```
public short setOutgoing(){
...
//@ set gTLsetOutgoing_normal = true;
}

/*@ private normal_behavior
  @ ...
  @ ensures \old(gTLsetOutgoing_normal);
  @*/
public void setOutgoingLength(short len){
...
}
```

FIG. 7 – Extrait d'un fichier de sortie de l'outil JAG.

vérification des propriétés (i) et (ii) sur la classe APDU. On peut constater que des variables `ghost` sont ajoutées pour "observer" les appels et terminaisons des méthodes impliquées dans la formule temporelle. Par exemple, la valeur `true` est donnée à la variable `gTLsetOutgoing_normal` lorsque la méthode `setOutgoing` termine normalement, i.e., lorsque l'événement `setOutgoing normal` apparaît durant l'exécution. Une postcondition est ajoutée à la méthode `setIncomingAndReceive`, qui assure que lorsque, dans le passé, la méthode `setIncomingAndReceive` a déjà été invoquée avec succès (la variable `gTLsetIncomingAndReceive_normal` a la valeur `true`), la méthode ne pourra terminer normalement, i.e., sa postcondition sera obligatoirement évaluée à faux. De même, tant que la variable `gTLsetOutgoing_normal` n'aura pas la valeur `true`, i.e., la méthode `setOutgoing` n'aura pas terminé normalement, la postcondition de `setOutgoingLength` ne pourra être établie.

4.3. 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, en particulier pour la recherche automatique de contre-exemples.

Dans un travail préliminaire [5], nous avons montré comment générer des cas de test conformes à une propriété JTPL. L'exécution de ces tests, avec vérification à la volée des annotations JML, permet soit de valider la propriété soit d'exhiber un contre-exemple.

4.4. Résultats expérimentaux

L'outil a été validé sur différentes classes Java, dont la classe APDU présentée dans cet article.

Une étude plus importante a été réalisée sur un porte monnaie électronique pour la Java Card. L'outil JAG a été utilisé pour vérifier des propriétés temporelles de sûreté et de vivacité sur la spécification JML de ce porte-monnaie. Sur cette spécification, constituée de 500 lignes de JML, l'outil JAG a généré des annotations supplémentaires, pour vérifier des propriétés d'unicité de la personnalisation de la carte et d'atomicité d'opérations décomposées en plusieurs méthodes. Ensuite, la vérification de la consistance des annotations générées vis-à-vis du modèle JML a été réalisée à l'aide de l'outil JML2B [4, 3], un générateur d'obligations de preuve de consistance des modèles JML basé sur une traduction vers B. Certaines des obligations de preuve générées à partir des annotations supplémentaires ont dû être prouvées de manière interactive. La table 1 résume les résultats obtenus en utilisant les outil Jack [9] et JML2B (pour le modèle JML du porte-monnaie electronique) pour générer et vérifier en aval les obligations de preuve (OP).

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)
Electronic Purse (500 lignes de JML)	2	25	14 (12)

TAB. 1 – Résultats expérimentaux.

De plus, la spécification JML du porte-monnaie a été utilisée pour générer des tests avec l'outil JML-TT [6]. Ces tests ont ensuite été exécutés sur l'application elle-même. Enfin, l'outil JAG a encore été utilisé pour valider une implémentation Java du porte-monnaie électronique [5]. Les annotations ont dans ce cas été directement générées dans le code Java de l'implémentation et ont été vérifiées à l'exécution des tests par le vérificateur d'annotations JML à la volée (JML-RAC, Runtime Assertion Checker) des JMLTools.

5. Exemple d'utilisation

Dans cette partie, nous expliquons comment utiliser l'outil JAG et comment le combiner avec le générateur d'obligations de preuve Jack [10], qui confronte les annotations JML avec le code de la classe annotée. Ce tutoriel est fondé sur un exemple simple[5] de système de transactions.

5.1. Exemple illustratif

Nous illustrons la démarche de vérification avec JAG et Jack sur l'exemple présenté dans la figure 8. Il s'agit d'une classe qui implante un système de transactions avec tampon (buffer).

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 tampon temporaire. La transaction peut être abandonnée par invocation d'une méthode `abortTransaction()`.

Les deux propriétés que l'on souhaite vérifier sont les suivantes : (i) Le buffer doit être vidé avant de commencer une nouvelle transaction et (ii) toute transaction commencée doit fatalement se terminer.

En JTPL, ces deux propriétés peuvent s'exprimer comme suit.

```
(i)   after commitTransaction() terminates, abortTransaction() terminates
      before beginTransaction() called  always {bufferPosition == 0};
(ii)  after beginTransaction() called always {true}
       until abortTransaction() called,  commitTransaction() called
      under variant {buffer.length - bufferPosition};
```

Ces exemples montrent comment la syntaxe abstraite de JTPL est concrètement enrichie par les balises `{` et `}` de début et de fin de prédicat JML, et par le marqueur `;` de fin de propriété.

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. C'est une propriété de sûreté ("quelque chose de mauvais ne doit pas arriver").

La seconde formule signifie qu'après (`after`) qu'une transaction ait commencé elle doit fatalement (`until`) se terminer par un rollback ou une validation. Pour cette seconde formule, qui est une vivacité ("quelque chose de bon doit nécessairement arriver") on doit de plus fournir un variant (`under variant`), un entier naturel qui doit décroître strictement à chaque appel de méthode, pour garantir qu'on finira par atteindre l'état voulu.

[5]Il n'était pas possible d'illustrer ce tutoriel avec la classe APDU, car son code source, requis pour l'utilisation de Jack, n'est pas diffusé.

```
public class TransactionSystem {
  final int LENGTH = 30;
  byte [] status = new byte[LENGTH];
  byte [] buffer = new byte[LENGTH];
  int bufferPosition = 0;

  //@ invariant buffer != null;
  /*@ invariant
    @   buffer.length == LENGTH;
    @*/

  /*@ invariant
    @   bufferPosition <= LENGTH &&
    @   bufferPosition >= 0;
    @*/

  //@ ghost boolean trDepth = false;

  //@ requires trDepth == false;
  void beginTransaction() {
    buffer = new byte[LENGTH];
    //@ set trDepth = true;
  }

  //@ requires trDepth == true;
  void commitTransaction() {
    status = buffer;
    //@ set trDepth = false;
  }

  //@ requires trDepth == true;
  void abortTransaction() {
    bufferPosition = 0;
    //@ set trDepth = false;
  }

  /*@ requires trDepth == true &
    @   bufferPosition < buffer.length;
    @*/
  void modify(byte b) {
    buffer[bufferPosition] = b;
    bufferPosition++;
  }
}
```

FIG. 8 – Exemple illustratif : Un système de transactions.

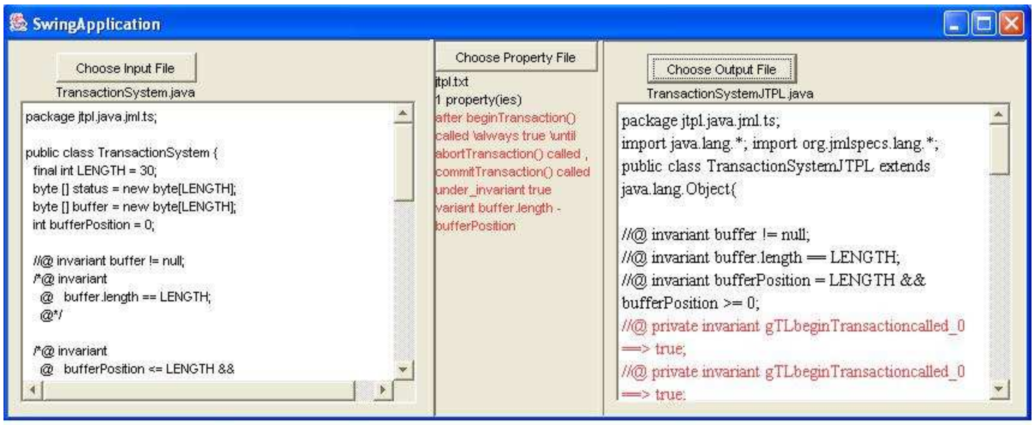

FIG. 9 – Interface principale de JAG.

5.2. Utilisation de l'interface graphique

L'outil JAG dispose d'une interface graphique représentée dans la figure 9. Elle permet de sélectionner et de visualiser les fichiers de travail. La fenêtre principale est composée de trois parties : à gauche une fenêtre permet de consulter le fichier source ; au centre apparaît la liste des propriétés temporelles à vérifier ; enfin, à droite, on peut visualiser le fichier de sortie généré. Les annotations ajoutées par l'outil apparaissent de la même couleur que la propriété à partir de laquelle elles ont été générées. Les étapes de la génération d'annotations avec l'interface graphique de l'outil JAG sont les suivantes.

1. **Chargement de l'outil.** L'outil JAG s'ouvre sur une interface principale présentée dans la figure 9.
2. **Sélection du fichier d'entrée.** En cliquant simplement sur le bouton "Choose Input File" (Choisir un fichier d'entrée), une boîte de dialogue s'ouvre afin de choisir un fichier Java. Le fichier est alors analysé et un contrôle de type est réalisé – nous utilisons pour cela l'analyseur syntaxique fourni par les JMLTools. Si le fichier ne comporte pas d'erreur de syntaxe, il est affiché dans le cadre (1).
3. **Sélection du fichier contenant les propriétés temporelles.** En cliquant sur le bouton "Choose Property File" (choisir un fichier de propriété), on ouvre une boîte de dialogue qui permet de sélectionner le fichier de propriétés temporelles. Après analyse syntaxique, la liste des propriétés est affichée dans le cadre central (2). Les primitives ainsi que les annotations

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

/*@  invariant
  @    gTLcommitTransactionterminates_1
  @ || gTLabortTransactionterminates_1
  @               ==> bufferPosition == 0;
  @*/

//@  invariant buffer.length - bufferPosition >= 0;

/*@  constraint
  @   \old(gTLbeginTransactioncalled_2)
  @       ==> (buffer.length - bufferPosition)
  @           < \old(buffer.length - bufferPosition);
  @*/


/*@  constraint
  @   \old(gTLbeginTransactioncalled_2)
  @     ==> ((TrDepth == false) || (TrDepth == true)
  @      || (bufferPosition < buffer.length
  @             && TrDepth == true));
  @*/

public void beginTransaction() {
//@ set gTLbeginTransactioncalled_2 = true;
//@ set gTLcommitTransactioncalled_1 = false;
//@ set gTLabortTransactioncalled_1 = false;
...
}
```

FIG. 10 – Extrait d'un fichier de sortie de l'outil JAG.

sont générées automatiquement en mémoire.

4. **Génération du fichier de sortie.** En cliquant sur le bouton "Choose Output File" (choisir un fichier de sortie), une boîte de dialogue s'ouvre afin de choisir le nom et l'emplacement du fichier de sortie. JAG crée alors le fichier de sortie en parcourant le fichier d'entrée et en ajoutant les annotations générées à partir des propriétés temporelles (voir figure 10). Le résultat de la génération est enregistré dans le fichier sélectionné et il est affiché dans le cadre de droite (3). La traçabilité est visualisée ici, car les annotations générées sont affichées de la même couleur que la propriété temporelle dont elles sont issues.

Dans la figure 10, 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 - Cela est obtenu en générant la disjonction des préconditions des méthodes). Le lecteur pourra remarquer que dans le cas présent, la dernière contrainte historique est trivialement satisfaite.

5.3. Vérification des annotations générées

Pour vérifier que le code Java est bien conforme aux annotations générées, on peut utiliser n'importe lequel des outils de vérification pour JML. Nous utilisons ici l'outil Jack [10], qui est un générateur d'obligations de preuve : à partir du code Java et des annotations JML, Jack génère un ensemble de formules logiques appelées *obligations de preuve*. La satisfaction de ces obligations de preuve implique la correction du code Java vis-à-vis des annotations JML. La vérification de cette satisfaction des obligations de preuve est confiée à des outils de preuve automatique ou interactive comme Simplify [22], Coq [25] ou haRVey [24]. Jack est distribué sous la forme d'un plug-in de l'environnement de développement Eclipse[6]. Ce plug-in ajoute un environnement spécifique qui est présenté dans la figure 11. Le cadre (A) permet de sélectionner la classe Java que l'on veut vérifier et/ou de consulter la liste des obligations de preuves générées. Le cadre (B) affiche des statistiques sur la génération d'obligations de preuve (pourcentage des obligations de preuve actuellement prouvées, temps nécessaire à la réalisation des preuves...) et des informations relatives à l'obligation de preuve que l'on est en train de consulter (le cadre affiche par exemple le but qui est à prouver et les hypothèses disponibles). Enfin, le cadre (C) affiche la classe Java sélectionnée en colorant les annotations JML et les portions de code Java correspondant à l'obligation de preuve sélectionnée.

Nous chargeons le fichier de sortie généré par JAG dans l'outil Jack et nous lançons la génération des obligations de preuve en choisissant la sortie vers le prouveur automatique Simplify [22].

[6] `http://www.eclipse.org`

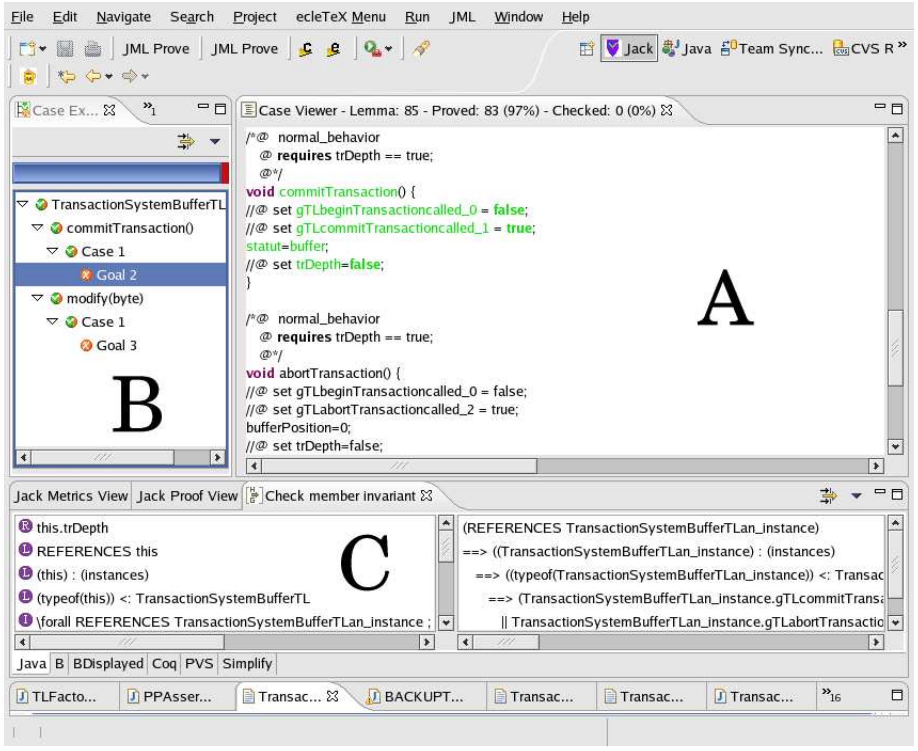

FIG. 11 – Interface principale de Jack.

L'interface de Jack nous informe alors que 85 obligations de preuve ont été générées et que 83 d'entre elles ont été vérifiées automatiquement. Pour les deux obligations de preuve qui restent à prouver, on utilise la traçabilité de l'outil Jack, illustrée dans la figure 12. Celle-ci permet facilement de déterminer que la première obligation de preuve correspond à l'invariant généré suivant :

```
/*@ invariant
  @   gTLcommitTransactionterminates_1 || gTLabortTransactionterminates_1
  @             ==> bufferPosition == 0;
  @*/
```

Ceci signifie que l'outil ne parvient pas à prouver que cet invariant est préservé par l'exécution des méthodes `commitTransaction()` et `modify()`. Pour poursuivre cette analyse, on utilise la traçabilité de JAG, comme détaillé dans la partie suivante.

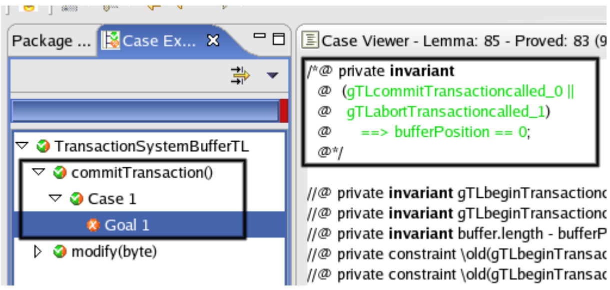

FIG. 12 – Traçabilité des obligations de preuve de Jack.

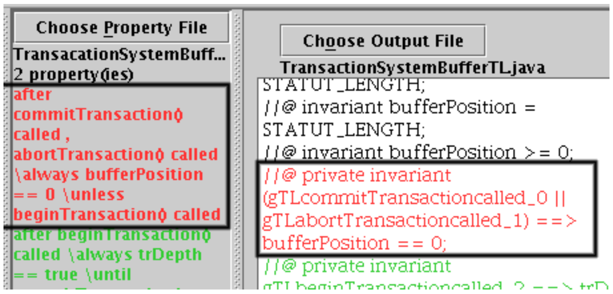

FIG. 13 – Traçabilité des annotations de JAG.

5.4. Traçabilité des annotations et diagnostic

La traçabilité des annotations implantée dans JAG permet d'aider l'utilisateur en cas d'échec de la preuve, comme c'est le cas ici. Deux cas sont alors à envisager :

- Soit la propriété que l'on veut prouver n'est pas satisfaite, ce qui indique la présence d'une erreur dans le programme.
- Soit la propriété est satisfaite mais la preuve est trop complexe pour être réalisée de manière automatique. En particulier, cela peut arriver lorsque le prouveur manque d'hypothèses pour établir automatiquement le but de l'obligation de preuve.

Grâce à la coloration de l'interface de JAG, illustrée par la figure 13, on trouve que cet invariant a été généré à partir de la propriété temporelle (i), qui signifiait qu'après tout appel de la méthode `commitTransaction()`, le tampon (buffer) devait être vide.

Dans le code source de la méthode `commitTransaction()`, qui est

```
statut=buffer;
//@ set trDepth=false;
```

nous constatons que la variable *bufferPosition* n'est pas remise à `0`. L'erreur venait donc d'une mauvaise implémentation du code source. On peut noter que cette erreur aurait pu être exhibée en générant automatiquement, avec JML-TT, des tests orientés par la propriété temporelle [5]. Nous la corrigeons donc en ajoutant la ligne de code

```
bufferPosition = 0;
```

dans le corps de la méthode `commitTransaction()`.

Pour la seconde obligation de preuve dont la preuve a échoué, en utilisant à nouveau l'interface de JAG, nous trouvons qu'elle concerne aussi la propriété temporelle (i). La propriété (i) peut ne pas être satisfaite par un appel à la méthode `modify()` quand aucune transaction n'est en cours (après un commit ou un abort et avant un appel à `beginTransaction()`). Cependant, dans ce cas, la précondition de `modify()` ne peut pas être satisfaite, c'est-à-dire que `modify()` ne peut pas être appelée, donc il n'y a pas d'erreur de programmation. C'est la propriété temporelle qui n'est pas assez précise. On la complète en ajoutant la négation de la précondition de `modify`, ce qui donne

```
after commitTransaction() terminates, abortTransaction() terminates
before beginTransaction() called
always { bufferPosition == 0
 &&!(trDepth == true && bufferPosition < buffer.length) };
```

Ainsi, l'ensemble des obligations de preuve est prouvé de manière automatique et les deux propriétés temporelles sont satisfaites par la classe Java.

6. Conclusion et travaux futurs

Nous sommes convaincus que les méthodes formelles de vérification du logiciel ont un rôle essentiel à jouer dans la sécurité des logiciels critiques. Cependant, pour être adoptées par la communauté de développement la plus large possible, il est essentiel que ces méthodes soient appliquées aux langages les plus populaires. Ceci a été bien compris par les auteurs du langage d'annotations JML, dont la syntaxe est très proche du langage Java. Ces annotations permettent d'exécuter de nombreux outils de vérification statique ou dynamique du code annoté. Indépendamment, les schémas de propriétés temporelles de Dwyer *et.al.* ont été conçus pour rendre l'expression de propriétés temporelles plus accessible aux programmeurs. Ces efforts ont été unifiés grâce au langage JTPL, qui a été choisi pour cette raison comme premier format d'entrée de l'outil de génération d'annotations JML présenté ici. Depuis, nous avons complété cet outil avec un formalisme d'automates, également présenté ici.

Ainsi, notre outil de génération automatique d'annotations JML s'intègre naturellement dans une chaîne logicielle qui permet au programmeur Java de vérifier des propiétés temporelles en assimilant un minimun de connaissances supplémentaires. La particularité et l'un des atouts de ce travail est de générer des annotations JML standard, sur lesquelles l'ensemble des outils JML peuvent être appliqués.

Afin de rendre cette chaîne logicielle plus opérationnelle, nous travaillons actuellement dans deux directions principales. Tout d'abord, nous étendons la génération d'annotations à d'autres formalismes d'entrée, en particulier à la LTL [23] et aux diagrammes d'états UML 2.0. Parallèlement, nous préparons une intégration plus forte de JAG avec les outils en aval, en particulier avec les outils de génération d'obligations de preuve JML2B, Krakatoa et Jack.

En particulier, une intégration plus forte avec Krakatoa consisterait à générer directement un équivalent des annotations JML dans le format intermédiaire de WHY [14, 15], en ajoutant les variables et invariants nécessaires. Une telle intégration permettrait de vérifier de manière similaire les programmes Java/JML, mais également les programmes C annotés avec Caduceus [16].

La réalisation effective d'une chaîne logicielle intégrée de vérification et/ou de validation des propriétés temporelles nécessiterait une importante coopération entre trois formats fédérateurs : Le format WHY pour la génération des obligations de preuve, le format BZP [7] pour la validation par animation symbolique, génération de tests et recherche de contre-exemples et le format interne de représentation de propriétés temporelles et d'annotations de l'outil JAG.

L'outil JAG et sa documentation sont disponibles en téléchargement à l'adresse

`http://www.lifc.univ-fcomte.fr/~jag`.

Remerciements. Les auteurs tiennent à remercier les relecteurs de la pertinence de leurs remarques, qui leur ont permis d'améliorer la qualité de ce document.

Références

[1] M. Barnett, K. Rustan, M. Leino, and Wolfram Schulte. The Spec♯ programming system : An overview. In *CASSIS 2004*, LNCS. Springer, 2004.

[2] F. Bellegarde, J. Groslambert, M. Huisman, J. Julliand, and O. Kouchnarenko. Verification of liveness properties with JML. Technical Report RR-5331, INRIA, 2004.

[3] F. Bouquet, F. Dadeau, and J. Groslambert. Checking JML specifications with B machines. In H. Treharne, S. King, M. Henson, and S. Schneider, editors, *Procs of the Int. Conf. on Formal Specification and Development in Z and B, (ZB'05)*, volume 3455 of *LNCS*, pages 435–454, Guildford, UK, April 2005. Springer.

[4] F. Bouquet, F. Dadeau, and J. Groslambert. JML2B : Checking JML specifications with B machines. In *B'2007, the 7th Int. B Conference - Tool Session*, volume 4355 of *LNCS*, pages 285–288, Besancon, France, January 2007. Springer.

[5] F. Bouquet, F. Dadeau, J. Groslambert, and J. Julliand. Safety property driven test generation from JML specifications. In *FATES/RV'06, 1st Int. Workshop on Formal Approaches to Testing and Runtime Verification*, volume 4262 of *LNCS*, pages 225–239, Seattle, WA, USA, August 2006. Springer.

[6] F. Bouquet, F. Dadeau, and B. Legeard. Automated boundary test generation from JML specifications. In *FM'06, 14th Int. Conf. on Formal Methods*, volume 4085 of *LNCS*, pages 428–443, Hamilton, Canada, August 2006. Springer.

[7] F. Bouquet, B. Legeard, and N. Vacelet. BZP : Un format fédérateur pour l'évaluation de spécifications formelles. In *JFPLC'03, Journées Francophones de Programmation en Logiques et Contraintes*, pages 203–216, Amiens, France, June 2003. Hermes, TSI.

[8] 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.

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

[10] L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness : A developer-oriented approach. In *FME 2003*, volume 2805 of *LNCS*, pages 422–439. Springer, 2003.

[11] Y. Cheon and G.T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). Technical Report 02-05, Department of Computer Science, Iowa State University, March 2002. In SERP 2002, pp. 322-328.

[12] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. Patterns in property specifications for finite-state verification. In *International Conf. on Software Engineering*, pages 411–420. IEEE Computer Society Press/ACM Press, 1999.

[13] E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B : Formal Models and Semantics, chapter 14, pages 996–1072. Elsevier, 1990.

[14] J.-C. Filliâtre. Verification of Non-Functional Programs using Interpretations in Type Theory. *Journal of Functional Programming*, 13(4) :709–745, July 2003.

[15] J.-C. Filliâtre. Why : a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003.

[16] Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of c programs. In *6th International Conference on Formal Engineering Methods, ICFEM 2004*, number 3308 in LNCS, pages 15 – 29, Seatle, 2004.

[17] J. Groslambert. A JAG extension for verifying LTL properties on B event systems. In *B'2007, the 7th Int. B Conference - Tool Session*, volume 4355 of *LNCS*, pages 262–265, Besancon, France, January 2007. Springer. To appear.

[18] J. Groslambert. Verification of LTL on B event systems. In *B'2007, the 7th Int. B Conference*, volume 4355 of *LNCS*, pages 111–125, Besancon, France, January 2007. Springer. To appear.

[19] J. Groslambert, J. Julliand, and O. Kouchnarenko. JML-based verification of liveness properties on a class. In *SAVCBS'06, Specification and Verification of Component-Based Systems*, pages 41–48, Portland, Oregon, USA, November 2006.

[20] G.J. Holzmann. The model checker SPIN. In *IEEE Trans. on Software Engineering*, volume 23-5, pages 279–295, 1997.

[21] G.T. Leavens, A.L. Baker, and C. Ruby. JML : a Java Modeling Language. In *Formal Underpinnings of Java Workshop (at OOPSLA '98)*, October 1998.

[22] G. Nelson. *Techniques for Program Verification.* PhD thesis, Stanford University, 1980.

[23] A. Pnueli. The temporal logic of program. In *18th Ann. IEEE Symp. on foundations of computer science*, pages 46–57, 1977.

[24] S. Ranise and D. Deharbe. Light weight theorem proving for debugging and verifying units of code. *Proc. of the International Conference on Software Engineering and Formal Methods (SEFM03).IEEE Computer Society Press,Camberra,Australia*, September 2003.

[25] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V7.1*, October 2001. http ://coq.inria.fr.

[26] 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.

[27] Pierre Wolper. Temporal logic can be more expressive. In *FOCS*, pages 340–348. IEEE, 1981.