Université Pierre et Marie Curie-Paris 6 INRIA-Saclay

Rapport de stage de Master 2

Invariants et raffinements en présence de partage

Asma Tafat

18 septembre 2009

Stage effectué du 01 Avril au 30 Septembre 2009
sous la direction de Sylvain Boulmé et Claude Marché

Résumé

Pour raisonner sur un composant logiciel qui comporte un état, la notion d'invariant joue un rôle central : un invariant est une propriété de l'état qui peut invariablement être observée par les clients du composant. Typiquement, on peut vouloir décrire le comportement du composant (par exemple une pile bornée) à partir d'une représentation abstraite de l'état (par exemple une suite d'éléments), la représentation concrète (par exemple un tableau) étant masquée aux clients du composant. Le lien entre la représentation concrète et la représentation abstraite correspond à un invariant, appelé invariant de collage. On peut ainsi prouver que l'implémentation du composant raffine (c'est-à-dire simule) la description abstraite du composant.

La difficulté de cette technique vient du fait que des violations de l'invariant peuvent être observées par des clients du composant si un tiers peut modifier arbitrairement une portion de l'état sur lequel porte l'invariant. En même temps, il semble trop restrictif d'interdire le partage d'état dans les invariants des composants.

Le but de ce stage est proposer une approche pour supporter le raffinement de programmes Java, à base de champs modèles, d'invariant de collage et d'Ownership. C'est cette relation d'Ownership qui nous assure que le partage d'états n'engendre pas de violation d'invariants par des tiers.

Table des matières

Table des figures

Remerciements

Au terme de ce travail, je tiens à exprimer ma profonde gratitude ainsi que mes sincères remerciements à :

Claude Marché et Sylvain Boulmé, mes encadrants de stage pour leur collaboration, leurs conseils pertinents et leur soutien durant ma période de stage. Que ce travail soit le modeste témoignage de ma haute considération et mon profond respect.

Je remercie également, toute l'équipe pédagogique de l'Université Pierre et Marie Curie et les responsables de la formation Sciences et Technologies du Logiciel.

Enfin, un grand remerciement à l'ensemble des personnes qui ont participé de près ou de loin à l'élaboration de ce travail.

Chapitre 1

Présentation

J'ai effectué mon stage de recherche sous la direction de Sylvain Boulmé (Verimag, Équipe DCS, http://www-verimag.imag.fr/async/) et de Claude Marché (INRIA Saclay, Équipe Demons-ProVal, http://proval.lri). Les deux équipes font partie de la même Action de Recherche Coopérative (ARC) de l'INRIA, à savoir CeProMi (http://www.lri.fr/cepromi).

L'équipe Demons-Proval est une équipe-projet du centre de recherche INRIA Saclay commune avec le LRI (Laboratoire de Recherche en Informatique) et l'Université Paris-Sud 11. Elle a pour objectif de proposer des outils et des méthodes pour spécifier et prouver des programmes impératifs de type Java ou C, tels que Krakatoa [26] ou Caduceus [20], respectivement. Ces outils prennent, en entrée, des programmes annotés par des spécifications décrivant leur comportement attendu et les compilent en programmes **Why** [21] . **Why** est une plateforme générique, développée par l'équipe, qui permet d'extraire des obligations de preuve et de les transmettre à des outils de preuves interactifs (Coq, Isabelle, PVS ...) ou automatiques (Alt-Ergo, Simplify, Z3 ...).

L'équipe DCS de Verimag, quant à elle, à plusieurs axes de recherche, parmi lesquels : les méthodes formelles pour la sécurité informatique, la vérification de programmes ou encore les langages de spécification et de vérification. Dans ce domaine, l'équipe étudie la possibilité d'étendre le formalisme de raffinement de la méthode B [2].

Mon travail consiste à exploiter les techniques de raffinement, utilisées dans B, dans des programmes tels que Java. Dans ce but, on a utilisé une des techniques autorisant le partage d'état dans les invariants, à savoir l'approche Boogie [22, 7].

Le présent rapport sera organisé comme suit : Il y aura deux principaux chapitres.

Dans le premier qui est une sorte d'état de l'art, on définit le contexte de travail, et on posera au fur et à mesure la problématique étudiée, en présentant les différents concepts sur lesquels porte le stage. On verra par exemple pourquoi il est aussi utile de spécifier et de vérifier les programmes. Et sur-

tout comment le faire en utilisant la notion d'invariant de classe. Dans cette optique, différentes méthodologies ont été définies : la méthode Boogie, le raffinement, les champs modèles . . .Solutions qui sont, en général, destinées à des programmes du type Java.

On présentera ensuite, dans le chapitre suivant, nos contributions en exposant l'approche proposée.

Chapitre 2

Contexte et problématique

2.1 Spécification et vérification de programmes

2.1.1 Preuves de correction de programmes

Ce qu'il faut savoir, c'est que prouver la correction d'un programme ne veut pas dire rechercher les erreurs et les **corriger**. Cette tache s'appelle le débogage.

Le problème avec ce type de processus, c'est qu'il peut ne qu'établir l'inexactitude d'un programme incorrect. Il ne doit, ainsi, aucunement être considéré comme une méthodologie de preuve de programme. Ainsi, prouver la correction d'un programme requière des méthodologies mathématiquement rigoureuses qu'on appelle **méthodes formelles**.

Pour assurer qu'un programme respecte certaines propriétés, on lui associe une spécification formelle sur laquelle il est possible de raisonner mathématiquement. Le programme n'étant rien d'autre que la réalisation exécutable d'une spécification, vérifier qu'un programme n'a pas d'erreur, revient à s'assurer qu'il est bien conforme à sa spécification. Les spécifications sont le plus souvent données sous forme de **contrats**.

2.1.2 Programmation par contrats

Le principe de ce paradigme de programmation, introduit par *B.Meyer* dans son langage *Eiffel* [27], est d'offrir au programmeur la possibilité de préciser ce qui doit être vrai à un moment donné de l'exécution. Le déroulement des traitements est régit par des règles appelées **assertions**.
Il existe trois types d'assertions :

1. **pre-condition** : La condition qui doit être vérifiée avant l'exécution d'un programme donné. Elle garantit que l'exécution du programme est possible sans erreur.

2. **Invariant** : Il s'agit d'une condition qui est toujours vraie. Selon le type d'invariant, cette assertion caractérise l'état interne de tout le programme, ou seulement d'une partie comme par exemple pour un invariant de boucle ou un invariant d'objet.
3. **Post-condition** : La condition qui doit être vérifiée après le déroulement d'un traitement donné. Cette condition doit garantir que le traitement a bien réalisé son travail.

Ces assertions permettent notamment de faire des preuves de correction des codes par rapport à leur spécifications, reprenant le cadre formel du calcul de la plus faible pre-condition de Dijkstra ou la logique de Hoare.

La programmation par contrat a été mise en oeuvre au dessus : de Java par l'intermédiaire du *Java Model Language* (*JML*) [11], de C par l'intermédiaire de ACSL [9] et VCC [31], et également au dessus de C# par l'intermédiaire de *Spec#*.

2.1.3 Logique de Hoare

La vérification déductive des propriétés des programmes est une tache difficile qui a longtemps été abordée. Mais la première contribution significative est celle de Floyd en 1967 [30] en Hoare en 1969 [12], introduisant la logique de Floyd-Hoare.

La logique de Hoare mise en place est un formalisme logique permettant de raisonner sur la correction des programmes informatiques. Ainsi, un programme est associé à une pre-condition P, qui permet de caractériser l'ensemble des états possibles avant l'exécution du programme S, et une post-condition Q qui vérifie les états possibles après l'exécution du S. Les évolutions d'un état vers un autre sont modélisées par des règles. Un programme est représenté par le triplet :

$$\{P\}S\{Q\}.$$

Où P et Q sont des propriétés exprimées en logique des prédicats.

On dit que le triplet $\{P\}S\{Q\}$ est valide ou que le programme S est *correct* vis-à-vis de sa spécification P, Q , si, à partir de tout état initial vérifiant P ; si le programme termine alors il fait passer le système dans un état satisfaisant Q.

Considérons, par exemple, la fonction *Power* suivante qui prend en argument un réel x et un entier n et est sensée retourner la valeur de x^n (figure 2.1).
Cette fonction *Power* annotée par sa spécification formelle est présentée dans la figure 2.2.
La spécification ainsi définie comporte :

```
Power (real x,integer n) {
real r = 1;
real t = x;
  integer i = n;
   while(i > 0) {
    if(i mod 2 == 1) {
      r = r * t;
    }
    t = t * t;
    i = i/2;
  }
  return r;
}
```

FIG. 2.1 – Fonction *Power* non spécifiée

```
Power (real x,integer n) {
  {P : n ≥ 0}
  real r = 1;
  real t = x;
  integer i = n;
   while(i > 0) {
   {Invariant : i ≥ 0 and x^n == t^i * r}
    if(i mod 2 == 1) {
      r = r * t;
    }
    t = t * t;
    i = i/2;
  }
  {Q : r == x^n}
  return r;
}
```

FIG. 2.2 – Spécification de la fonction *Power*

1. La condition sous laquelle la fonction peut être invoquée (la pre-condition). $P : n \geq 0$
2. L'invariant de la boucle **while** devant être valide au début et à la fin de chaque itération de boucle : $i \geq 0$ *and* $x^n == t^i * r$
3. La condition qui doit être vérifiée à la fin de la méthode (la post-condition). $Q : r == x^n$.

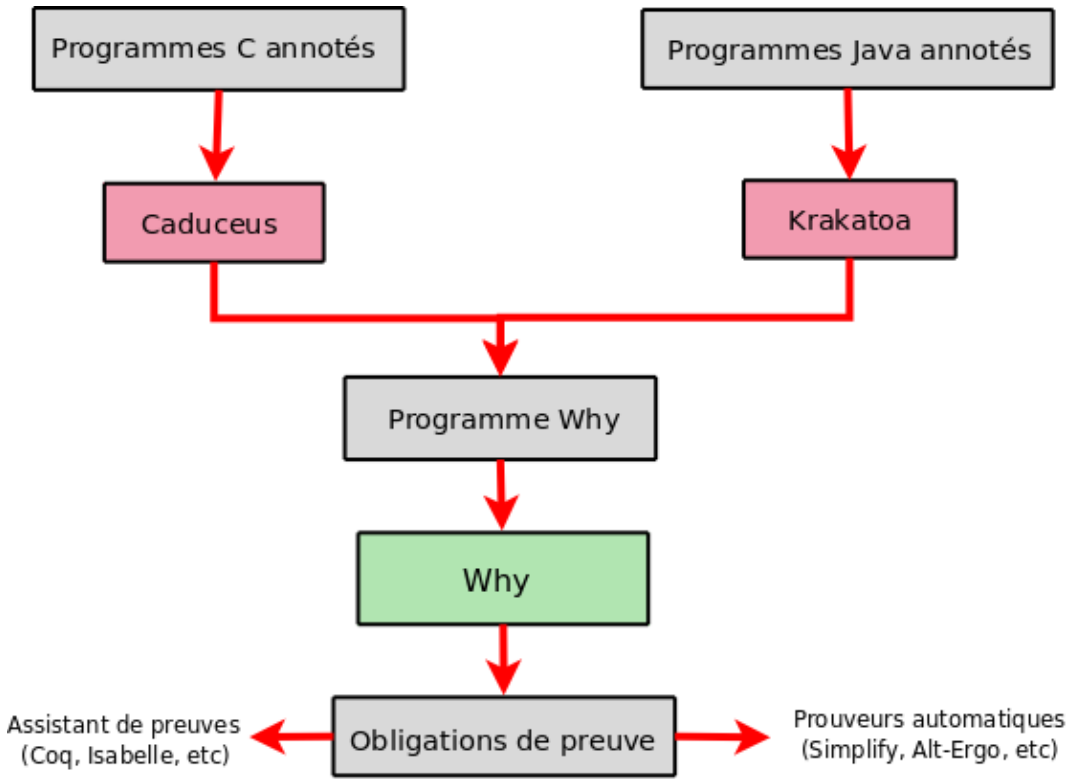

FIG. 2.3 – La plateforme Why

2.1.4 Obligations de preuve

Une fois le programme annoté par une spécification formelle, un ensemble de formules logiques exprimant la sûreté et la correction du programme sont extraites. Ces formules qu'on appelle **obligations de preuve** sont ensuite prouvées. La première phase (l'annotation du programme) est du ressort du programmeur, la deuxième phase (l'extraction) est un algorithme automatique, elle peut donc être réalisée par un outil tel que *Why*. Enfin, la troisième phase (la preuve) peut être traitée soit par des prouveurs automatiques, soit à l'aide d'assistants de preuves.

La plateforme Why

Why est une plateforme générique de vérification de programmes. A partir d'un programme source annoté par des spécifications formelles, la plateforme Why extrait les obligations de preuve et les transmet à des outils de preuves automatiques (Simplify, Yices, Alt-Ergo, ...) ou interactifs (Coq, Isabelle, ...).

```
/*@ requires n ≥ 0;
  @ ensures \result == lpower(x, n);
  @*/
Power (double x,int n) {
   double r = 1;
   double t = x;
   int i = n;
  /*@ loop_invariant :
    @ i ≥ 0 &&  lpower(x, n) == lpower(t, i)*r;
    @ loop_variant n
    @*/
   while(i>0) {
      if(i mod 2 == 1) {
         r=r*t;
      }
      t=t*t;
      i=i/2;
   }
   return r;
}
```

FIG. 2.4 – Spécification de la fonction *Power* avec Krakatoa

L'outil Krakatoa

L'outil Krakatoa est intégré à la plateforme Why. Tout comme *JML*, Krakatoa adopte le paradigme de programmation par contrat au langage Java. Il offre le moyen d'ajouter des annotations (pre-condition, post-condition et invariant) sous la forme de commentaires dans le code Java et permet ainsi, de prouver de nombreuses propriétés des programmes en utilisant la méthode de Hoare. Dans la figure 2.4, la méthode *Power* présentée précédemment est annotée avec Krakatoa.

On remarque, dans cette spécification, l'utilisation de *lpower* qui est prédicat logique. Car, contrairement à *JML*, l'utilisation des méthodes pures dans les annotations n'est pas autorisée dans *KML* (Krakatoa Model Language). Cependant, ce dernier permet la déclaration de nouvelles fonctions logiques, de prédicats ou même de *blocs axiomatiques* dans lesquels peuvent être déclarés des types, déclarations axiomatiques de prédicats et de fonctions logiques. Dans notre exemple, le bloc axiomatique est défini sur la figure 2.5

Pour prouver qu'un programme, annoté avec Krakatoa, est correct, dans le sens où il vérifie sa spécification, on lance le processus de vérification avec la commande `gwhy Nomfichierfichier.java`, ce qui a pour conséquence :

1. La lecture du fichier et la génération des obligations de preuve.
2. Les formules générées sont ensuite affichées dans une fenêtre graphique

```
/*@ axiomatic Power {
  @ logic real lpower(real x, integer n) ;
  @ axiom power_zero :
  @   \forall real x ; lpower (x, 0) == 1.0 ;
  @ axiom power_n :
  @   \forall integer n, real x ; n > 0 ==>
  @     lpower(x, n)==lpower(x, n-1)*x ;
  @ axiom power_carre_div2_pair :
  @   \forall integer n, real x ; n % 2 != 1 ==>
  @     lpower(x*x, n/2) == lpower(x, n) ;
  @ axiom power_carre_div2_impair :
  @   \forall integer n, real x ; n % 2 == 1 ==>
  @     lpower(x*x, n/2)*x == lpower(x, n) ;
  @ }
  @*/
```

FIG. 2.5 – bloc axiomatique *Power*

de la plateforme Why comme le montre la figure 2.6.

Dans la partie gauche de la fenêtre, on voit les obligation de preuves générées par l'outil (lignes) et les prouveurs automatiques (colonnes). On a effectué le test avec cinq prouveurs Alt-Ergo [15], Simplify [18], Yices [17], Z3 [16], CVC3 [8].

On voit que Alt-Ergo est le seul à prouver toute les obligations de preuve générées. Cependant, il ne prouve pas les deux lemmes (mult_assoc et mult_comm) définis, ni Simplify, alors que Yices, Z3 et CVC3 le font.

Dans la fenêtre en haut à droite, l'obligation de preuve est affichée (le but est sous la barre et les hypothèses au dessus), et enfin en bas à droite, on reconnaît le code source avec un bloc orange qui indique l'origine de l'obligation de preuve sélectionnée dans la liste.

2.1.5 Invariants de classe

La notion d'invariant est très importante dans les langages de spécification. Un invariant peut être associé à un programme ou a une structure de données. On s'intéresse, dans ce qui suit, aux invariants de classe. C'est à dire aux invariants permettant de spécifier une propriété devant être observée dans toutes les instances de la classe annotée.

Par, exemple, si on considère la classe *Purse* (figure 2.7) qui a un champ *balance* représentant le contenu de la bourse, une méthode $credit(int\ s)$ pour créditer la bourse de s et une méthode $withdraw(int\ s)$ pour la débiter de s. La classe est également annotée avec un invariant de classe $balance \geq 0$ ce qui veut dire que toute instance de la classe *Purse* a une balance positive ou null.

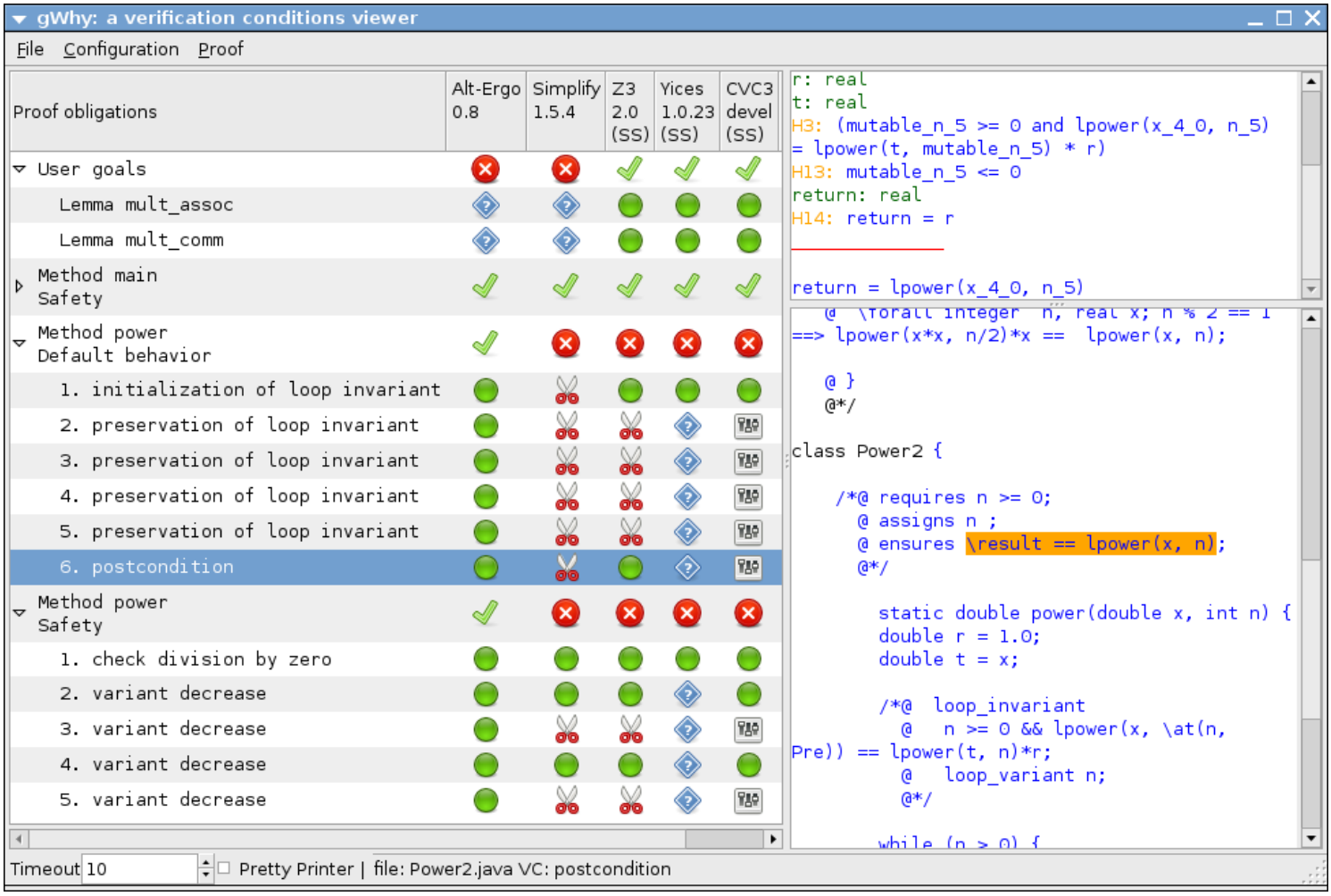

FIG. 2.6 – Les obligations de preuve affichées dans GWhy

Ce qui est tout a fait logique. Le contenu d'une bourse ne peut aucunement être négatif.

On remarque dans la spécification de la classe *Purse* qu'en plus des clauses **requires** et **ensures** qui introduisent les pré et post conditions respectivement, le contrat de la méthode *withdraw* contient une clause **assigns** permettant d'indiquer les champs modifiables par la méthode, notamment *balance*. On y spécifie également que la méthode a deux comportement possibles :

1. un comportement *normal* si s est inférieur à *balance*, et dans ce cas la post-condition est que la nouvelle valeur de *balance* est égale à l'ancienne valeur ($\backslash old(balance)$) moins s
2. un comportement *exceptionnel* si la valeur de s est supérieure à *balance*. Dans ce cas là, une exception est levée, et c'est ce que permet d'introduire la clause **signals**.

```
class Purse {
    private int  balance;
    //@ invariant balance ≥ 0;
   /*@ requires amount > 0;
    @ assigns balance;
    @ ensures balance == amount;
   @*/
    public Purse(int amount) {
        balance = amount;
    }
 /*@ requires s >= 0;
    @ assigns balance;
   @ ensures balance == \old(balance) + s;
   @*/
    public void credit(int s) {
        balance+ = s;
    }
 /*@ requires s >= 0;
 /*@ assigns balance;
   @ ensures s <= \old(balance)&&balance == \old(balance) − s;
   @ behavior amount_too_large :
   @    assigns \nothing;
   @    signals (NoCreditException)s > \old(balance);
   @*/
    public void withdraw(int s) throws NoCreditException {
        if (s < balance)
            balance = balance − s;
        else
            throw new NoCreditException();
    }
}
```

FIG. 2.7 – Invariant de classe

Le résultat obtenu en faisant tourner cet exemple avec `gwhy` est représenté sur la figure 2.8.

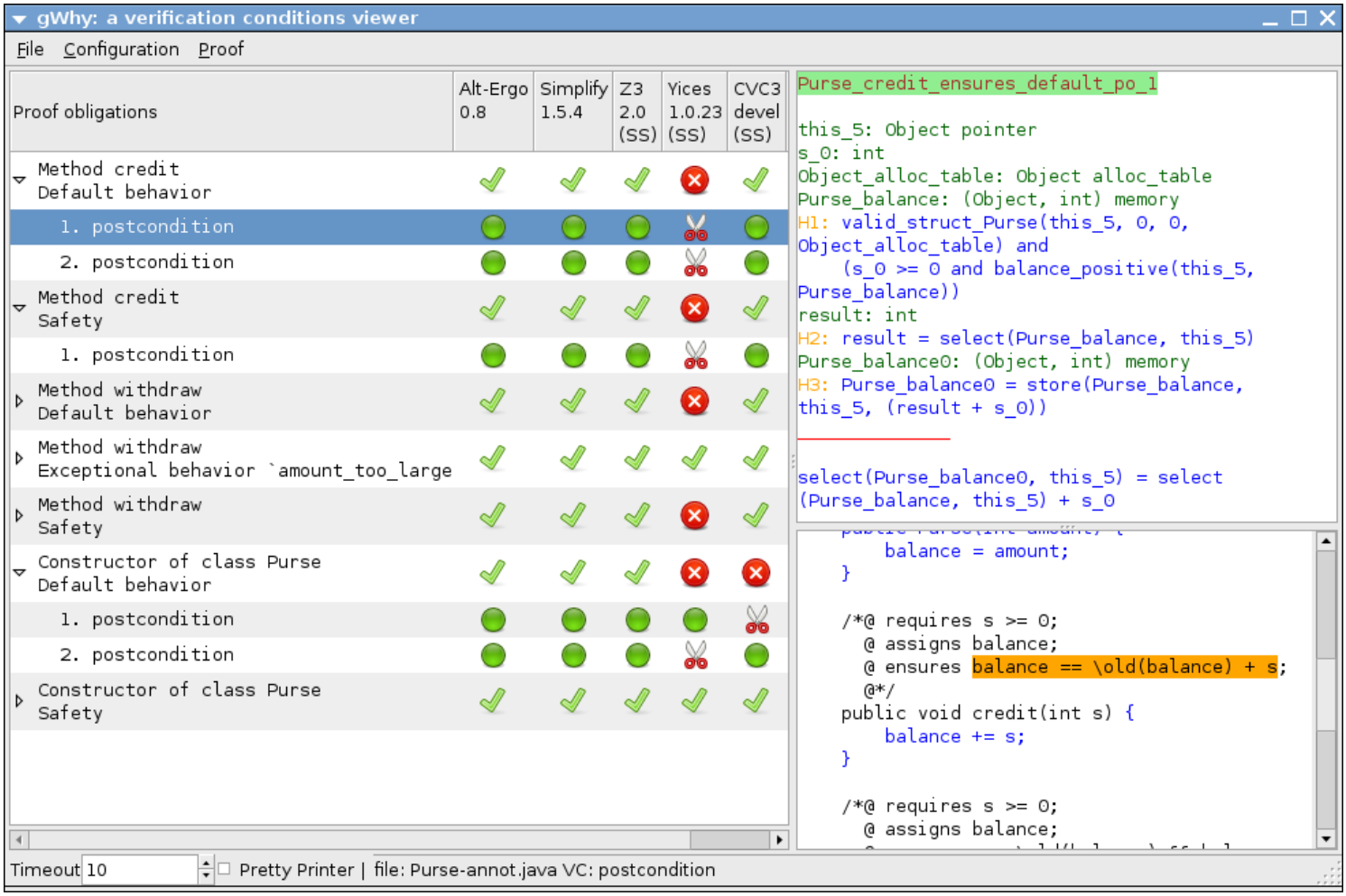

FIG. 2.8 – Les obligations de preuve de Purse affichées dans GWhy

2.1.6 Limites

Le traitement des invariants est assez difficile, car il soulève un certain nombre d'interrogations. Pour quelles instances faut-il vérifier si l'invariant est préservé ou s'il est rompu ? Quand doit-on le faire ? Le spécifier explicitement peut être en contradiction avec le fait que les données et les détails de l'implantation ne doivent être révélées au client.

Si on considère l'exemple suivant, où la classe *Car* a un champ *sensor* de type *Sensor*, on peut constater que l'invariant peut être rompu dans le corps de la méthode *update* de la classe *Car* (figure 2.9).

```
class Sensor {
    float rpm; // tours de roue/minute
     ⋮
    public void read() {
        rpm = ...;
    }
}

class Car {
    int displayed_speed; // km/h
    Sensor sensor;
   //@ invariant displayed_speed == round(K*sensor.rpm);
    ⋮
    public void update() {
        sensor.read();
        //L'invariant est momentanément rompu.
        displayed_speed = K * sensor.rpm;
    }
}
```

FIG. 2.9 – Invariant momentanément rompu

```
s = new Sensor();
c = new Car(s);
    ⋮
s.read(); //viole l'invariant de c.
```

FIG. 2.10 – Aliasing

L'idée la plus naturelle, pour résoudre ce type de problème, est de considérer qu'un invariant de classe doit être vérifié au début et à la fin de chaque méthode. C'est ce qui est utilisé dans Krakatoa actuellement, et qui a permi de faire l'exemple Purse précédent. Néanmoins, cette solution reste insuffisante. La notion d'invariant de classe, telle que définie ci-dessus, étant trop rigide pour spécifier des programmes réalistes : il arrive qu'un objet brise l'invariant d'autres objets temporairement, sans mettre en cause la correction d'un programme. Dans le code de la figure 2.10, on a un partage de références entre s et le $c.sensor$. Ce qui fait que l'invocation de la méthode $read()$ sur l'objet s revient modifie la valeur de champs $c.sensor.rpm$ et viole l'invariant de la classe Car.

2.2 Préservation des invariants de classe : La méthode Boogie

Spec# est un environnement de programmation dont l'objectif est le développement et la maintenance de logiciel correct. Il est constitué d'un langage de programmation (C# plus contrats), d'un compilateur qui vérifie les contrats à l'exécution et d'un vérificateur statique de programmes, *Boogie*, qui tente de prouver la correction des programmes par rapport à leur contrats. En d'autres termes, **Boogie** est un langage intermédiaire pour la vérification de programmes Spec#. Il joue un rôle similaire à celui de Why pour Krakatoa. C'est une méthodologie garantie correcte et aisément implantable.

Dans [7], une solution qui repose sur la méthode Boogie est proposée pour résoudre les problématiques énoncées dans la section précédente.

L'idée est que chaque objet possède un champ booléen spécial inv, qui indique s'il vérifie son invariant. L'utilisateur contrôle, ainsi, les portions de programme où il s'autorise à violer, temporairement, l'invariant de classe. L'intuition, derrière ce champ, est qu'un objet est une sorte de boîte, le champ inv indique si elle est ouverte ou fermée. Si $o.inv = true$ (la boite est fermée), alors l'objet o satisfait son invariant. Si, par contre, $o.inv = false$ (la boite est ouverte), alors l'invariant de o peut être rompu.

Le champ inv est spécial parce qu'il ne peut apparaître que dans la spécification, mais pas dans l'invariant, ni dans l'implantation. Sa valeur ne peut être modifiée, dans l'implantation, qu'à l'aide des deux nouvelles instructions **pack** et **unpack** définies comme suit :

Pour toute expression o de type T :

pack $o \equiv$
  assert $o \neq null \wedge \neg o.inv$;
  assert $Inv_T(o)$;
  $o.inv := true$;

unpack $o \equiv$
  assert $o \neq null \wedge o.inv$;
  $o.inv := false$;

L'instruction **pack** vérifie que l'invariant de l'objet o est bien valide et change la valeur $o.inv$ de $false$ à $true$, et l'instruction **unpack** met la valeur de $o.inv$ à $false$.

Étant donnée que la mise-à-jour de champ $o.f := E$ peut rompre l'invariant de classe (figure 2.10), cette opération est soumise à la pre-condition $o.inv = false$ qui peut être satisfaite en faisant précéder l'instruction $o.f := E$ par un **unpack** o pour s'assurer que l'invariant peut bien être rompu (figure 2.11). Dans [7], on prouve que cette méthode est correcte dans le sens où : pendant

```
public void update() {
unpack this; //Ouverture de l'objet
sensor.read();
displayed_speed := K * sensor.rpm;
pack this; //Fermeture de l'objet
}
```

FIG. 2.11 – Utilisation des **pack** et **unpack**

toute l'exécution du programme, *tout objet fermé vérifie son invariant*. La preuve sera vue en détail dans la première section du chapitre 3, car nous allons adapter l'instruction **pack** et par conséquent refaire la preuve.

2.2.1 Composants

Dans une classe Java, certains champs sont déclarés comme étant des composants. Par exemple, dans la classe Car, le champ *sensor* est un composant. On dit que *sensor* appartient à Car.

Pour coder la présence de composants dans une classe, un modificateur **rep** est introduit pour identifier ces champs. L'invariant de classe peut maintenant dépendre de tous les champs $x.f_0 \ldots .f_n.g$, où chaque f_i est déclaré avec un modificateur **rep**.

Un autre champ booléen explicite *committed* est rajouté dans chaque objet. $o.committed = true$ signifie que o **appartient** à un objet fermé. Avant l'appel du constructeur, le champ *committed* (respectivement *inv*) est mis à $false$.

Pour prendre en compte ce nouveau champ, les instructions **pack** et **unpack** sont redéfinies comme suit :

pack $o \equiv$

assert $o \neq null \wedge \neg o.inv$;
assert $Inv_T(o)$;
assert $\bigwedge_{f \in Comp_T(o)}(o.f = null \vee (o.f.inv \wedge \neg o.f.committed))$;
{**if** $(o.f \neq null)$ { $o.f.committed := true$ }| $f \in Comp_T(o)$ }
$o.inv := true$;

unpack $o \equiv$

assert $o \neq null \wedge o.inv \wedge \neg o.committed$;
$o.inv := false$;
{**if** $(o.f \neq null)$ { $o.f.committed := false$ }| $f \in Comp_T(o)$ }

où $Comp_T(o)$ est l'ensemble des champs **rep** dans T.

Ainsi, l'instruction $s.read()$ (figure 2.10) qui violait l'invariant de la classe Car devient interdite, car, à cet instant de l'exécution, l'objet c est fermé, et donc on ne peut pas accéder à ses composants, notamment $c.sensor$.

2.2.2 Héritage

Jusque là, on n'a pas parlé d'héritage, pourtant c'est un concept qui existe bien en Java. Par conséquent, d'une part, les champs d'une classe peuvent soit être déclarés dans la classe elle même, soit hérités d'une super-classe. D'autre part, si chacune des classes, mère et fille, sont annotées par un invariant, lequel doit être vérifié dans une instance de la classe fille ?

L'idée, dans Boogie, est de diviser l'ensemble des champs d'une classe donnée en *class frames*. Une pour chaque classe de la hiérarchie dont la racine est la classe *Object*, et le niveau le plus bas est le type dynamique de l'objet. Pour plus de clarté, considérons les déclarations suivantes :

class *Object* {
  //pré-déclarée par le langage
}

class *A* **extends** *Object*{
  $w : W$;
  $x : X$;
  //**invariant** ...w ...x ;
}

class *B* **extends** *A*{
  $y : Y$;
  $z : Z$;
  //**invariant** $...w ...x ...y ...z$;
}

Un objet dont le type dynamique est B, par exemple, possède trois *class frames* :

1. Celle correspondant à *Object*, qui ne possède aucun champs et dont l'invariant est *true*.
2. Celle correspondant à A. L'invariant de cette *class frame* ne peut dépendre que de ses champs, à savoir w et x.
3. Et enfin celle correspondant à B, qui a pour champs y et z. Son invariant peut dépendre de l'ensemble les champs appartenant à A et à B.

On dit qu'un objet est valide pour une *class frame* donnée s'il vérifie l'invariant de sa classe la plus dérivée. Ici, les objets de B sont valides pour $\{Object\}$, $\{Object, A\}$ ou bien pour $\{Object, A, B\}$.

Ainsi, le champ booléen *inv*, qui indiquait si un objet vérifiait son invariant ou pas, ne suffit plus tel quel. On aimerait avoir la possibilité spécifier la *class frame* pour laquelle l'objet est valide. Pour cela, il suffit d'avoir un champ, en l'occurrence *inv*, qui prend pour valeur un nom de classe.

Par exemple, étant données les classes précédentes, le champ *inv* d'un objet de type B peut prendre une des trois valeurs : $Object$, A, B pour la première, seconde et troisième *class frame* respectivement.

Avec ce nouveau codage, les instructions **pack** et **unpack** définies auparavant ne sont plus valables. Il faut donc les redéfinir.

Pour toute classe T avec comme super classe immédiate S et pour chaque expression o dont le type est une sous classe de T :

pack o as $T \equiv$

assert $o \neq null \wedge o.inv = S$;
assert $Inv_T(o)$;
assert $\bigwedge_{f \in Comp_T(o)}(o.f = null \vee (o.f.inv = type(p) \wedge \neg o.f.committed))$;
{**if** $(o.f \neq null)$ { $o.f.committed := true$} | $f \in Comp_T(o)$ }
$o.inv := true$;

unpack o from $T \equiv$

assert $o \neq null \wedge o.inv = T \wedge \neg o.committed$;
$o.inv := S$;
{**if** $(o.f \neq null)$ { $o.f.committed := false$ } | $f \in Comp_T(o)$ }

Où **type**(o) retourne le type dynamique de l'objet o.

La condition sous laquelle l'instruction $o.f := E$ est permise est aussi différente en présence d'héritage. Si le champ f est déclaré dans la classe T, alors l'instruction n'est autorisé que si o est "suffisamment ouvert", c'est-à-dire que $o.inv$ est une des super-classe de T.

2.3 Raffinement et méthode B

On veut utiliser les invariants de classe dans un contexte de raffinement. C'est pourquoi on s'intéresse de plus près à ce concept, dans cette section.

2.3.1 Raffinement

> "... le programme concret implémente (ou raffine) le programme abstrait correctement lorsque toute utilisation du programme concret ne conduit pas à une observation qui n'est pas aussi une observation du programme abstrait." — Paul Gardiner et Carroll Morgan

La notion de raffinement a été introduite dans les années 1970 par Dijkstra [19], puis formalisée par Back [3, 4] dans les années 1980. Plusieurs travaux ont ensuite développé cette notion, en particulier Abadi et Lamport [1], Back [3, 4], Morgan [14], Morris [28] et Abrial [2].

interface $Calc$ {

$values$: collection de réels

//reset le calculateur

//@assigns $values$;

//@ensures $values = \emptyset$;

void $reset()$;

//ajouter un nouvel élément.

//@assigns $values$;

//@ensures $values = \backslash old(values) \uplus \{x\}$;

void $add(double\ x)$; //retourne la moyenne des éléments de la collection.

//@requires $values \neq \emptyset$;

//@ensures $result = \frac{sum(values)}{card(values)}$;

real $mean()$;

}

FIG. 2.12 – Le calculateur de Morgan

Tout comme la programmation par contrats, la théorie du raffinement permet de montrer la correction d'une implantation par rapport à sa spécification. C'est un moyen de construire des programmes qui satisfont leurs spécifications.

Un exemple classique pour étudier le raffinement, est le Calculateur de Morgan dont on présente l'interface : Le but de ce calculateur étant de collecter un ensemble de réels et d'en fournir la moyenne, son interface comporte :

- Un champ abstrait $values$ qui représente l'ensemble des réels collectionnés.
- Les méthodes $reset()$, $add(real\ x)$ et $mean()$ qui permettent de remettre la collection à zéro, rajouter un nouvel élément x à la collection et calculer la moyenne des éléments de la collection et la retourner, respectivement.

On constate que pour cette même abstraction, on peut avoir plusieurs implantations possible. On parle de non déterminisme du raffinement.

Une implantation astucieuse, serait de ne considérer que le nombre d'éléments dans la collection et leur somme. On obtiendrait l'implantation suivante :

$Calc\_Impl$ qui raffine l'interface $Calc$, est annotée par un **invariant de collage** reliant les champs abstrait déclarés dans l'interface($values$) et les champs concrets déclarés dans l'implantation (sum et $count$).

$$Inv_{calc} : sum = sumbag(values) \wedge count = cardinal(values).$$

Dire qu'une classe raffine une interface, c'est dire que chaque méthode de la classe raffine la méthode qui lui correspond dans l'interface.

```
Calc_Impl {
    real sum;
    int count;
    //Invariant de collage.
    invariant
        sum = sumbag(values) && count = cardinal(values);
    void reset() {
        sum := 0.0;
        count := 0;
    }
    void add(real x) {
        sum := sum + x;
        count := count +1;
    }
    real mean() {
        return sum/count;
    }
}
```

FIG. 2.13 – Implémentation du calculateur de Morgan

Pour pouvoir affirmer qu'une concrétisation $\{Pre\_Conc\}Conc\{Post\_Conc\}$ raffine une abstraction $\{Pre\_Abs\}Abs\{Post\_Abs\}$, il faut montrer les obligations de preuve suivantes :

- $Pre\_Abs \wedge Inv \Rightarrow Pre\_Conc$.
- $\forall abs \forall conc \bullet (Pre\_Abs \wedge Inv) \Rightarrow Post\_Conc \Rightarrow \exists abs' \bullet (Post\_Abs \wedge Inv)$.

Où Inv c'est l'invariant de collage, abs sont les champs abstraits déclarés dans Abs et $conc$ les champs déclarés dans $Conc$. Donc pour chacune des méthodes définies dans le calculateur, on a les obligations de preuve présentées ci-dessus. Par exemple, celles de la méthode *add* sont :

- $true \Rightarrow true$
- $\forall\ values \bullet Inv_{calc} \Rightarrow (sum = \backslash old(sum) + x\ \&\&\ count = \backslash old(count) + 1) \Rightarrow \exists\ values' \bullet (values' = values \uplus \{x\} \wedge Inv_{calc})$

Parmi les nombreuses applications du raffinement, on cite ici la méthode B.

2.3.2 La méthode B

B est une méthode transformationnelle qui part d'un modèle abstrait(machine abstraite) ou une spécification pour engendrer graduellement, par un processus de raffinement, du code correct. Dans B, chacune des étapes du cycle de développement des logiciels peut être considérée comme une activité impliquant

l'écriture de preuve mathématique pour justifier ses résultats. C'est précisément l'ensemble des preuves qui prouve que le système est correct et qui fait que les programmes ainsi construits sont **corrects par construction**.

L'unité de spécification, dans cette méthode, est un module appelé *machine abstraite*, constitué d'invariants et des méthodes spécifiées avec des pré et post-conditions.

Le langage de spécification de la méthode B n'est pas un sous ensemble du langage de programmation cible, mais au contraire un langage de spécification ayant toute la puissance des mathématiques : la théorie des ensemble.

Dans [10], on propose une solution qui consiste à introduire les instructions pack et unpack dans B.

2.4 Les champs modèles de JML

Dans l'approche par contrat, les spécifications visibles par le client doivent être exprimées indépendamment de l'implantation, de façon à permettre l'encapsulation. L'abstraction des données est un moyen d'y parvenir : on met en relation l'état concret de la classe avec une valeur abstraite, par l'intermédiaire d'une fonction d'abstraction : par exemple une liste chaînée avec le concept mathématique de suite. Le comportement de la classe peut alors être exprimé en terme d'opérations sur les suites. Le moyen le plus souvent utilisé pour mettre en oeuvre l'abstraction des données est le recours aux *champs modèles* (model fields).

Contrairement aux champs concrets, les champs modèles ne peuvent pas être directement modifiés par un programme. Ils n'ont d'existence que du point de vue de la spécification et de la preuve, et leur valeur est déterminée par la fonction d'abstraction.

La déclaration d'un champ modèle a la forme suivante :

$$\textbf{model}\ T\ m\ \textbf{constraint\_by}\ E\,;$$

Où T est le type du champ et E spécifie une contrainte sur $this.m$. C'est une expression booléenne pouvant référencer les champs modèles.

La classe *Rectangle*, dans la figure 2.14, illustre comment les champs modèles sont utilisés dans une spécification. Un objet *Rectangle* est défini par les coordonnées de ces deux points opposés, tel que mentionné dans l'invariant. La valeur du champ modèle *width* qui représente la largeur du rectangle, est déterminée par la différence entre les abscisses des deux points opposés, x_2 et x_1. Cette relation entre le champ modèle *width* et les champs concrets x_1 et w_2, qu'on appelle *contrainte* de *width*, est exprimée dans la partie **constraint_by** de la déclaration du champ modèle.

La clause **ensures**, dans la spécification de la méthode *ScaleH*, utilise le champ modèle pour formuler sa fonctionnalité sans référencer l'implantation

```
class Rectangle {
    int x1, x2, y1, y2 ;
    model int width constrained_by width = x2 − x1 ;
    model int height constrained_by height = y2 − y1 ;
    //@ invariant x1 ≤ x2 ∧ y1 ≤ y2 ;
    Rectangle() {
        x1 := 0; y1 := 0;
        x2 := 0; y2 := 0;
        pack this ;
    }
    // requires 0 ≤ factor ;
    //requires inv = Rectangle∧]committed ;
    //assigns width, x2 ;
    //ensures width = old(width) ∗ actor/100 ;
    void ScaleH(int factor) {
        unpack this ;
        x2 := (x2 − x1) ∗ factor/100 ∗ x1
        pack this ;
    }
}
```

FIG. 2.14 – Une spécification avec des champs modèle.

concrète. La seconde clause **requires** est requise par la méthodologie pour l'utilisation des instruction **pack** et **unpack**.

Le concept de base de l'abstraction des données est bien connu. Cependant, les techniques de vérification existantes pour les champs modèles peinent du point de vue de leur correction, de leur modularité et de leur expressivité.

Les problèmes qui se posent quand on introduit ce type de champ dans un programme sont liés à la signification qu'on va leur donner. Comment seront-ils interprétés lors de la vérification de programme ? Comment les mettre à jour à chaque fois que les champs dont ils dépendent le sont ? Dans la classe *Rectangle*, par exemple, les valeurs des champs modèles *width* et *height* doivent être modifiées lorsque les coordonnées des points le sont.

Dans [23], les auteurs proposent une solution partielle permettant de gérer les problèmes liés à l'introduction des champs modèles. L'idée clé est de traiter les champs modèles comme de simple champs concrets. C'est une approche basée sur la méthodologie Boogie. Elle consiste à inclure la mise-à-jour des champs modèles dans le traitement effectué par l'opération **pack**, en rajoutant les déclarations suivantes :

pour tout m déclaré dans T ou hérité d'une superclasse {
    if $\neg R_m^T(o, o.m)$ {

assert $(\exists r \bullet R_m^T(o, r))$;
$o.m :=$ **choisir** r **tel que** $R_m^T(o, r)$;
}
}

Cette approche a néanmoins des limites. Elle impose des restrictions reliées au non déterminisme du raffinement de certaines interfaces.

En effet, si on l'applique sur la méthode *add* du calculateur de Morgan, alors l'instruction **pack** qu'il faudra rajouter à la fin de la méthode,

```
void add(double x)
{
        unpack this;
        sum += x;
        count ++;
        pack this;
    }
```

générera, en plus, l'obligation de preuve suivante :

$\exists values \bullet sum = Sum(values) \wedge count = Card(values)$.

Cependant, rien, dans cette expression, ne garantit que la collection *values* retournée est bien l'union de l'ancienne collection et de $\{x\}$ à l'ancienne collection. Effectivement, il existe une infinité de collection contenant *count* éléments dont la somme est égale à *sum*.

Dans l'approche que nous proposons, on lève cette restriction. C'est une solution qui s'inspire du raffinement en assimilant des champs modèles à des variables abstraites.

Chapitre 3

Contributions

Notre objectif est le développement de programmes Java par raffinement. La différence avec la méthode B, est que le niveau le plus raffiné n'est pas le résultat de transitions à partir d'une machine abstraite, mais une véritable méthode java.

Le problème dans un tel contexte impliquant des structures de données mutables, est qu'il n'y a pas d'études théoriques assurant que les obligations de preuve ainsi générées sont suffisantes.

On présente dans ce qui suit une approche pour supporter le raffinement de programme Java, à base de champs modèles, d'invariant de collage et d'ownership.

3.1 Champs modèles non déterministes

Dans la section 2.3, on a mis en évidence les problème liés à l'utilisation des champs modèles. On a montré également, que l'approche proposée par *Leino-Muller* pour la prise en charge de ce type de champs, imposait certaines restrictions.

Dans ce qui suit, on propose, une autre solution qui permet notamment de lever ces restrictions. Pour ce, on se donne un mini langage qui offre la possibilité de définir des **interfaces abstraites** dans lesquelles peuvent être déclarés des :

- Champs modèles.
- Des profils de méthodes ; c'est à dire des entêtes de constructeurs et des méthodes annotées par des contrats.

Le calculateur de Morgan écrit dans ce langage est présenté sur la figure 3.1.

Revoyons maintenant la méthode *add*. Ce à quoi on veut aboutir, et que ne garantit pas l'approche de *Leino-Müller*, c'est que la nouvelle collection soit l'union de l'ancienne collection et du singleton $\{x\}$.

Si on considère, maintenant, l'obligation de preuve suivante ;

```
abstract interface Calc {
model Bag<double> values;

//reset le calculateur
//@assigns this.values;
//@ensures this.values == ∅;
void reset();

//ajouter un nouvel élément.
//@assigns this.values;
//@ensures this.values == old(this.values) ⊎ {x};
void add(real x);

//retourne la moyenne des éléments de la collection.
//@requires this.values ≠ ∅;
//@assigns nothing;
//@ensures result == sum(this.values) / card(this.values);
double mean();
}
```

FIG. 3.1 – **interface abstraite** pour le calculateur de Morgan

$\exists values \bullet (sum = Sum(values) \wedge count = Card(values)) \wedge values = \backslash old(values) \uplus \{x\}$.

alors, la collection *values* finale est bien celle voulue. Le fait de rajouter la formule $values = \backslash old(values) \uplus x$, qui n'est autre que la post-condition abstraite de la méthode, nous a donc assuré que la valeur finale est unique et bien celle recherchée.

On définit alors une nouvelle instruction **pack** qui permettra de générer les obligations de preuve définies dans l'approche de *Leino-Müller* **et** la post-confition abstraite de la méthode. On notera cette instruction **pack** o **with** $M_0 := v_0, \ldots M_n := v_n$ **such that** P, où o est l'objet sur lequel le **pack** sera appliqué, M_i sont les champs modèles qui seront modifiés par l'instruction, v_i dont les variables fraîches dénotant les nouvelles valeurs des champs modèles par analogie à ce qui se fait dans la méthode B où on aurait écrit : $M_i :=$ any v_i where $formule$ then $M_i := v_i$. Enfin P, est une assertion pouvant faire appel aux v_i, aux M_i précédent et à des champs concrets.

Le résultat de l'utilisation de la nouvelle instruction **pack** dans la méthode *add* du calculateur de Morgan est présenté

Dans B, cela donnerait : **ANY** c **WHERE** $c = values \uplus \{x\}$ **THEN** $values := c$.

```
void add(double x) {
  unpackthis;
  sum += x;
  count ++;
  pack this with values := c such that c = this.values ⊎ {x};
}
```

Formellement, l'instruction **pack with**, ainsi définie, s'écrit comme suit dans la syntaxe *Why* :

pack o **with** $M_0 := v_0,\ M_1 := v_1,\ \ldots\ M_n := v_n$ **such that** $P\ \equiv$
{
$o \neq null \wedge o.inv\ =\ S\ \wedge$
$\exists\ v_0,\ \ldots v_n\ \bullet Inv_T[this.M_i\ \mapsto v_i][\ this \mapsto o]\ \wedge\ P\ \wedge$
$\bigwedge_{f \in Comp_T(o)}\{\ o.f = null \vee (o.f.inv\ =\ type(o.f) \wedge \neg o.f.committed);\}$
}
unit writes $M_0,\ \ldots\ M_n,\ o.inv,\ \{o.f.committed \mid f \in Comp_T(o)\}$;
{
$o.inv = T\ \wedge\ Inv_T[this \mapsto o]\ \wedge\ (\backslash old(P))[v_i \mapsto o.M_i] \wedge$
$\bigwedge_{f \in Comp_T(o)}\ (o.f\ \neq\ null) \Rightarrow o.f.committed$
}

Où $Inv_T[this.M_i\ \mapsto\ v_i][this\ \mapsto\ o]$ est l'invariant de collage dans lequel les instances des champs modèle M_i, apparaissant dans la clause **with** sont substituées par de nouvelles valeurs v_i

A partir de ce qui a été présenté jusque la à propos de notre approche, on peut déjà énoncer, puis prouver, le théorème suivant :

Théorème 1 : Les conditions suivantes sont des invariants de programme :

$\forall o, T \bullet o \neq null \wedge o.committed \Rightarrow o.inv = \mathbf{type}(o)$ (1)
$\forall o, T \bullet o \neq null \wedge o.inv <: T \Rightarrow Inv_T(o)$ (2)
$\forall o, T \bullet o \neq null \wedge o.inv <: T \Rightarrow (\forall p \in Comp_T(o) \bullet p = null \vee p.committed)$ (3)
$\forall o, T, o', T' \bullet (\forall p \in Comp_T(o), p' \in Comp'_T(o') \bullet o.inv <: T \wedge o'.inv <: T' \wedge$ (4)
$q \neq null\ \wedge q.committed \wedge p = p' = q \Rightarrow o = o' \wedge T = T')$

Où $<:$ dénote la relation de sous typage.

Preuve : La preuve de ce théorème se fait par induction sur la séquence d'exécution (pas à pas). Etant donnée la séquence d'exécutions suivante :

$$H_0 \xrightarrow{instr_0} H_1 \xrightarrow{instr_1} H_2 \xrightarrow{instr_2} \ldots \xrightarrow{instr_{i-1}} H_i \xrightarrow{instr_i} H_{i+1} \ldots \xrightarrow{instr_{n-1}} H_n.$$

Où les H_i sont des états mémoire.

En H_0, aucun objet n'est encore alloué, et donc l'invariant est vrai.

On suppose, maintenant, que les quatre invariants sont vérifiés en H_i et on montre qu'ils le restent en H_{i+1} et ce quelque soit l'instruction qui fait passer le programme de H_i vers H_{i+1}.

On ne prend en compte que les instructions qui peuvent étendre la portée des quantificateurs (allocation d'objet) ou bien modifier le champ d'un objet, à savoir :

1. Le constructeur de la classe *Object*
2. L'instruction **pack with**.
3. L'instruction **unpack**.
4. L'affectation.

Les autres instructions n'ayant aucune influence sur les invariants.

Cas 1 : Le constructeur de la classe *Object*.

Le constructeur de *Object* met *o.committed* à $false$ pour tout objet o fraîchement alloué, établissant, ainsi, l'invariant (1). Il affecte aussi la valeur *Object* au champ *o.inv* , ce qui prouve donc (2), vu que l'invariant de la classe *Object* est égal à *true* (la classe *Object* n'a pas d'invariant).

De façon similaire, étant donné que la classe *Object* n'a aucun champ **rep**, la conjonction $\bigwedge f \in Comp_{Object}(o)$ est vraie (l'ensemble $Comp_{Object}(o)$ est vide), alors les invariants (3) et (4) sont également établis.

Cas 2 : L'instruction **pack** obj **with** $M_i := v_i$ **such that** P.

Pour chaque objet p dans lequel la valeur *true* est affectée à *p.committed*, il existe une pre-condition correspondante telle que p est valide ($p.inv = type(p)$). Par conséquent, l'invariant (1) est vérifié pour tout objet p de type **rep**.

Par ailleurs, l'hypothèse de récurrence nous assure que si $obj.inv \neq type(obj)$, alors $\neg obj.committed$ (la contraposée de l'invariant énoncé dans le théorème). Ainsi, la pre-condition $obj.inv = S$ et le fait que **type**$(obj) <: T$ (d'après la vérification statique des types, le type dynamique d'un objet **type**(obj) est une sous classe de son type statique) impliquent que $obj.inv \neq type(obj)$ et donc que $\neg obj.committed$ avant l'exécution de l'instruction **pack with**. La valeur du champ *obj.committed* n'étant pas modifiée par l'instruction, on a $\neg obj.committed$ également après l'exécution, et par conséquent que l'invariant (1) est maintenu par l'instruction **pack with**.

D'une part, l'instruction **pack with** met la valeur de *obj.inv* à T ce qui rend la partie gauche de l'invariant (2) vraie. D'autre part, la pre-condition $\exists\ v_0,\ \ldots v_n\ \bullet Inv_T[this.M_i\ \mapsto v_i][\ this \mapsto obj]$ garantit que $Inv_T[this \mapsto obj]$ est aussi vrai, pour M_i substitué par v_i vérifiant $Inv_T[this.M_i\ \mapsto v_i][\ this \mapsto$

$obj]$ $\forall\ i, 0 \leq i \leq n$, dans la pre-condition. Et donc $Inv_T(obj)$ est vérifié et l'invariant (2) maintenu.

L'invariant (3) est, pour sa part, une conséquence directe de la post-condition de l'instruction **pack with**.

Pour prouver que l'invariant (4) est maintenu, on considère une instanciation de o, o', p, p', T, T', q. Deux cas sont à prendre en compte. Le premier suppose que $q.committed$ est égal à $false$ en H_i. Ainsi, l'invariant (3) assure qu'il n'y a aucun o, T, p tels que :

$$o.inv <: T \ \wedge \ p \in Comp_T(o) \ \wedge p = q\,;$$

est vérifiée à l'état initial. Par conséquent, si la partie gauche de l'invariant (4) est vérifiée en H_{i+1}, alors o et o' font référence à l'objet sur lequel le **pack with** est appliqué (qui est le seul objet dont le champ inv change), et donc la partie droite est vérifiée.

Le second cas suppose que $o.committed$ est égal à $true$ en H_i. Alors, à partir de l'hypothèse de récurrence de (4) et la prè-condition de l'instruction **pack with**, on conclut que ni o ni o' ne référencent l'objet sur lequel le **pack with** est invoqué. Ainsi, la partie gauche de (4) n'a pas été modifiée, et donc (4) est maintenu.

Cas 3 : L'instruction **unpack**.

L'invariant (1) est maintenu,car l'instruction **unpack** a pour seul effet de modifier la valeur du champ $committed$ de $true$ à $false$ et de ne changer que le champ inv des objets dont $committed$ est à $false$. Les invariants de programmes (2) et (4) sont préservés, car l'instruction n'a pas d'effet sur ces prédicats.

Pour l'invariant (3), noter que l'instruction met $p.committed$ à faux pour les objets $p \in Comp_T(o)$ uniquement, lorsque o et T sont les arguments de l'instructions. Si pour ces o et T, l'instruction modifie $o.inv$ pour fausser la partie gauche de l'implication , il reste seulement à prouver qu'il n'y a pas d'autres o', T' et p' qui satisfont $p' \in Comp'_T(o')$ tels que la valeur $p'.committed$ est modifiée. D'après l'hypothèse de récurrence, l'invariant (4) est vérifié en H_i, et donc on a $o = o' \ \wedge \ T = T'$; ce qui conclut la preuve.

Cas 4 : L'affectation.

L'affectation ne modifie pas les champs inv et $committed$ et maintient donc (1).

Soient f un champ déclaré dans une classe F, et l'instruction $x.f := E$. Supposons maintenant que l'instruction a un effet sur $Inv_T(o)$ pour o et T donnés. Ce qui veut dire que l'invariant declaré dans la classe T contient une expression $x.f$. On considère n'importe quelle expression de ce type dont l'objet (x) n'est pas null. De telles expressions d'accès dans l'invariant de classe

ont la forme $this.g_0.g_1 \ldots g_{n-1}.f$ pour g_0, g_1, ...,g_{n-1} des champs **rep** déclarés dans les classes M_0, M_1, ..., M_{n-1} ($n \geq 0$). C'est-à-dire pour tout j : $0 \leq j < n$, $X.g_j \in Comp_{M_j}(X)$. Pour plus de commodité, soit M_n un synonyme pour F ; alors M_0 est T. Maintenant, la pre-condition de la mise-à-jour implique $\neg(o.inv <: M_0)$, alors en appliquant n fois (1) et (3), on a $\neg(o.inv <: M_0)$. Plus précisément, à chacune de ces n applications, pour tout j : de $\neg(o.g_0.g_1 \ldots g_{j+1}.inv <: M_{j+1})$, la vérification statique des types, et l'invariant (1), on obtient $\neg o.g_0.\ldots.g_{j+1}.committed$, et à partir de ça, du fait que $o.g_0.\ldots.g_{j+1}$ est non null, et (3) instancié avec $o := o.g_0.\ldots.g_j$, on obtient : $\neg(o.g_0.\ldots.g_j.inv <: M_j)$. Ceci montre que (2) est préservé pour o en H_{i+1}, prouvant ainsi que l'invariant (2) est maintenu par la mise-à-jour.

Finalement, l'instruction $x.f := E$ ne peut changer la valeur de l'expression p dans $Comp_T(o)$ que si f est un champ rep dans T. Mais la pre-condition de l'instruction implique que $\neg(x.inv <: T)$, alors les parties gauches de (3) et (4) sont fausses. Donc, les invariants (3) et (4) sont maintenus.

3.2 Raffinement de programmes Java

Ce à quoi on veut aboutir dans cette deuxième partie, c'est prouver la *modularité du raffinement* tel que schématisé dans la figure 3.2 ; Si on a, d'une part, un programme P prouvé correct par rapport à une interface I, et d'autre part, une implémentation qui raffine I, alors le programme P est **garanti correct** vis à vis de l'implémentation. Par exemple, si on considèrele code-ci dessous.

```
Calc c ;
...
c.reset() ;
c.add(14.0) ;
c.add(17.0) ;
m = c.mean() ;
}
```

En utilisant la spécification du calculateur de Morgan présentée précédemment, on peut prouver que m est bien égal à 15.5. L'idée alors est de montrer que cette preuve reste valide pour toute implémentation (raffinement) de Calc. Plus précisément, si des propriétés ont été prouvées pour l'exécution abstraite d'un programme S, alors elles restent vraies pour son exécution concrète. C'est-à-dire, quelque soit le prédicat Q, s'il est prouvable dans un état mémoire abstrait H, alors il l'est également dans l'état mémoire concret corresondant $H \cup F$. Ce qu'on appelle ici "état mémoire abstrait" H, c'est l'état mémoire dans lequel on n'a que des champs abstraits. L'état mémoire

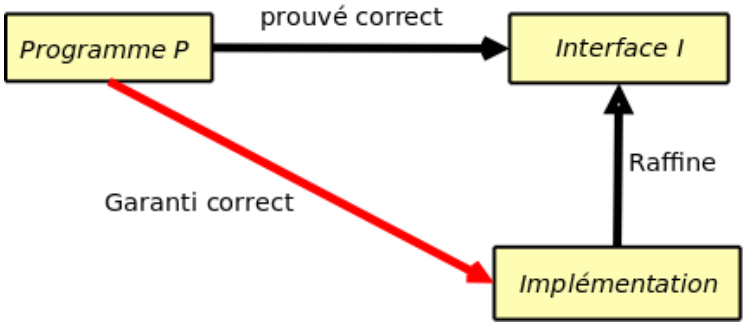

FIG. 3.2 – Modularité par raffinement

concret, par contre, est constitué de l'ensemble des champs modèles H plus l'ensemble des champs concrets F.

$$\forall\, Q \bullet\ H \models Q \Rightarrow H \cup F \models Q \tag{3.1}$$

Concretement, on se donne la séquence d'exécutions abstraites suivante :

$$H_0 \overset{instr_0}{\longrightarrow} H_1 \overset{instr_1}{\longrightarrow} \ldots \overset{instr_{i-1}}{\longrightarrow} H_i \overset{instr_i}{\longrightarrow} H_{i+1} \ldots \overset{instr_{n-1}}{\longrightarrow} H_n.$$

et la concrétisation correspondante :

$$H_0 \cup F_0 \overset{instr_0}{\longrightarrow} H_1 \cup F_1 \overset{instr_1}{\longrightarrow} \ldots \overset{instr_{i-1}}{\longrightarrow} H_i \cup F_i \overset{instr_i}{\longrightarrow} H_{i+1} \cup F_{i+1} \ldots \overset{instr_{n-1}}{\longrightarrow} H_n \cup F_n.$$

où les H_i et F_i sont des états mémoire comportant les champs abstraits et les champs concrets respectivement.

Avant de pouvoir énoncer le théorème de modularité par raffinement, on a besoin d'introduire une propriété que doivent vérifier les états mémoire, à savoir la propriété de clôture.

Propriété de clôture pour les composants : On dit qu'un état mémoire H est clos, si pour tout o appartenant à H, $o.f$ appartient aussi à H.

$$\forall\, o \bullet\ o \in H \Longrightarrow o.f \in H.$$

En pratique, cela signifie que $o.f$ ne pointe jamais sur une zone non allouée de la mémoire. En Java, cette propriété est garantie par le garbage collector.

On va également avoir besoin du lemme suivant

Lemme de monotonie de l'invariant global : Soient H un sous ensemble clos de l'état mémoire clos H' ($H \subseteq H'$), et INV l'invariant du théorème 1.

$$INV(H') \Rightarrow INV(H).$$

Preuve : D'une part, les quantificateurs des invariants portent sur tous les objets de H', et plus particulièrement ceux de H ($H \subseteq H'$). D'autre part, H est clos, ce qui veut dire que les champs $o.f$ sont dans H pour tout o dans H. Par conséquent, si les invariant du théorème 1 sont vérifiés pour les objets de H', alors ils le sont pour ceux de H.

On peut maintenant énoncer notre théorème.

Théorème 2 : Tout programme prouvé correct utilisant une interface I s'exécute correctement avec toute implémentation raffinant I.

Preuve : Prouver le théorème revient donc à prouver la formule (3.1). Pour cela, on procède par récurrence sur i.

En H_0, aucune instruction n'est encore exécutée et aucun objet n'est encore alloué en mémoire ($H_0 = F_0 = \emptyset$). Donc, si $H_0 \models P$ alors ce prédicat est toujours vérifié dans $H_0 \cup F_0$.

Supposons maintenant que la propriété est vérifiée en H_i : $H_i \models P \Rightarrow H_i \cup F_i \models P$ et montrons qu'elle l'est en H_{i+1} : $H_{i+1} \models Q \Rightarrow H_{i+1} \cup F_{i+1} \models Q$.

$$\begin{array}{ccc} H_i \models P & \overset{instr_i}{\longrightarrow} & H_{i+1} \models Q \\ \Downarrow & & \Downarrow \\ H_i \cup F_i \models P & \overset{instr_i}{\longrightarrow} & H_{i+1} \cup F_{i+1} \models Q \end{array}$$

Deux cas sont à distinguer. Soit l'instruction int_i faisant passer le programme de l'état H_i à H_{i+1} est un appel de méthode de la classe C, soit elle n'en est pas un.

abstract interface C {
  model $\overrightarrow{M}$;
  ...
  //@requires $pre(m)$;
  //@assigns M ;
  //@ensures $post(m)$;
  $m(\overrightarrow{x})$ {
  }
}

Cas 1 : int_i n'est pas un appel de méthode de C.

Les instructions **pack** et **unpack** sont interdites au niveau abstrait : dans le cas contraire, on ne saurait pas quel invariant doit être maintenu.

On n'a pas non plus d'accès à des champs du type $o.f$ où o est une instance d'une classe abstraite C, car les champs de C ne sont pas accessibles(visibles). Donc, $inst_i$ ne change pas F_i, ni les champs modèles de C. Par conséquent, $H_i \cup F_i \models Q \xrightarrow{instr_i} H_{i+1} \cup F_{i+1} \models Q$

Cas 2 : $inst_i$ est un appel à $o.m()$ de C.

On utilise, dans la preuve de Q, la règle de Hoare :

$$H \models Pre(m)[x/e] \Longrightarrow H, H', v \models Post(m)[x/e[result/v]].$$

Où v est la valeur retournée par la méthode m.

On a d'abord, $H_i \models Pre(m)[x/e]$, donc d'après l'hypothèse de récurrence

$$H_i \cup F_i \models Pre(m)[x/e] \tag{3.2}$$

Ensuite, dans le théorème 1, on a prouvé que l'invariant est vrai dans tout état accessible à partir de la heap. Ainsi, si on note INV le méta-invariant (l'invariant de collage), on a la relation :

$$\forall i \; H_i \cup F_i \models INV(H_i \cup F_i) \tag{3.3}$$

D'après (3.2) et (3.3) on a d'une part ;

$$H_i \cup F_i \models Pre(m)[x/e] \wedge INV(H_i \cup F_i) \tag{3.4}$$

D'autre part, lorsqu'on a prouvé le raffinement, on a montré que quelque soit v, pour un h qui ne contient que la partie de la mémoire visible par $o.m()$,

$$\textbf{si } h[x \mapsto v] \models Pre(m) \wedge INV(h) \textbf{ alors } h, h', result \models post(m) \tag{3.5}$$

En appliquant le lemme de monotonie sur la formule (3.4)(possible car $H_i \cup F_i$ est clos) pour ce h, on obtient :

$$h \models Pre(m)[x/e] \wedge INV(h) \tag{3.6}$$

On peut donc déduire de (3.5) et (3.6) :

$$h, h', result \models Post(m)[x/e]. \tag{3.7}$$

et conclure enfin $H \cup F, H' \cup F', result \models Post(m)[x/e]$.

3.3 Exemples

On présente dans cette section, des exemples sur lesquelles on applique notre approche.

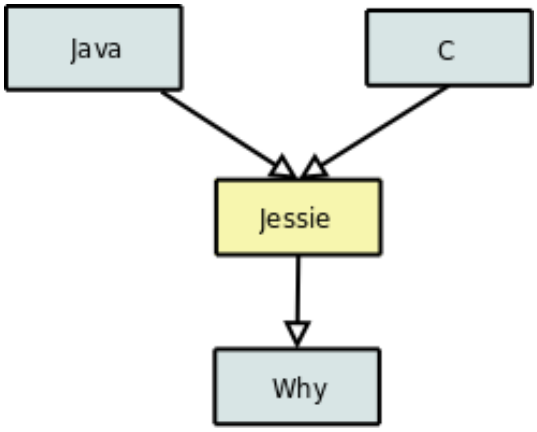

FIG. 3.3 – Le langage intermédiaire Jessie

3.3.1 Calculateur de Morgan implanté en Jessie

Le premier exemple est une implantation du calculateur de Morgan en jessie [24], qui est un langage entre Java et C d'une part, et Why d'autre part tel que le montre la figure 3.3.

Les instructions **pack** et **unpack** telles que définies dans Boogie on été implantées en 2007 par *Romain Bardou* [5]. La souplesse de Why nous pemet de simuler l'instruction **pack with** en utilisant cette implantation.

Dans la figure 3.4, on déclare un bloc axiomatique permettant de définir un `Bag` (collection). On définit également un prédicat `glue_invL`qui, étant donnés une collection c et une instance de calculateur, le champ `count`, respectivement `sum`, du calculateur est égal au cardinal, respectivement la somme de tous les éléments, de la collection.

Est définie ensuite une structure de données représentant le calculateur (voir figure 3.5). Cet enregistrement possède deux champs (concrets) `count` et `sum`, un champ abstrait `coll` de type `Bag`, et est annoté par un invariant de collage qui définit la relation entre le champ abstrait et les champs concrets.

Les figures 3.6 et 3.7 montrent des simulations de l'instruction **pack with** utilisée dans les méthodes `reset` et `add` respectivement.

La pre-condition de la fonction `simulpack_add` :

```
∃bag c ; c == union(this.coll,single(x)) && glue_inv(c,this)
```

n'est autre que l'obligation de preuve générée par l'instruction **pack with** pour la méthode *add*.

$$\exists c \bullet sum = Sum(c) \wedge count = Card(c) \wedge c = values \uplus \{x\}$$

Les obligations de preuves $this \neq null \wedge this.inv = S$ sont, elles, assurées par la post-condition du **unpack**, et enfin les obligations de preuve qui restent sont spécifiées dansla clause ensures de `simulpackadd`.

```
 # InvariantPolicy = Ownership
axiomatic Bag {
 logic type bag
 logic bag emptybag()
 logic bag union(bag b1, bag b2)
 logic bag single(real x)
 logic integer cardinal(bag b)
 axiom cardinal_empty : cardinal(emptybag()) == 0
 axiom cardinal_single : ∀ real x ; cardinal(single(x)) == 1
 axiom cardinal_union : ∀ bag b1,b2 ;
 cardinal(union(b1,b2)) == cardinal(b1) + cardinal(b2)
 axiom cardinal_non_empty : ∀ bag b ; b != emptybag() ==>
cardinal(b) > 0
 logic real sumbag(bag b)
 axiom sumbag_empty : sumbag(emptybag()) == 0.0
 axiom sumbag_single : ∀ real x ; sumbag(single(x)) == x
 axiom sumbag_union : ∀ bag b1,b2 ;
 sumbag(union(b1,b2)) == sumbag(b1) + sumbag(b2)
}
logic glue_invL(bag c, Calc[0] this) =
 this.count == cardinal(c) && this.sum == sumbag(c)
```

FIG. 3.4 – Déclaration d'une collection dans Jessie

```
tag Calc = {
 abstract bag coll ;
 integer count ;
 real sum ;
 invariant glue(this) =
 this.count == cardinal(this.coll) && this.sum ==
sumbag(this.coll) ;
}
type Calc = [ Calc ]
```

FIG. 3.5 – Déclaration du calculateur dans Jessie

```
unit simupack_reset(Calc[0] this)
 requires ∃ bag c ; c == emptybag() && glue_inv(c,this) ;
 behavior default :
 assigns this.coll ;
 ensures this.coll == emptybag() && glue(this) &&
 \mutable(this,Calc) && this.committed == false ;
;
unit reset(Calc[0] this)
 requires \mutable(this,Calc) && this.committed == false ;
 behavior default :
 assigns this.coll,this.count,this.sum ;
 ensures this.coll == emptybag() &&
 \mutable(this,Calc) && this.committed == false ;
{
 unpack(this) ;
 this.count = 0 ;
 this.sum = 0.0 ;
 simupack_reset(this) ;
}
```

FIG. 3.6 – Simulation du **pack with** dans la méthode `reset`

Constater que le champ *o.inv* est noté \mutable(*o*, valeur).

On montre enfin dans la figure 3.9, une simulation du calculateur, où on remet la collection à zéro, on y insére deux éléments, et on en calcule la moyenne. Les assertions, nous permettent de vérifier que chacune des méthodes retourne bien le résultat attendu. Tel qu'on peut le voir sur la figure 3.10, les prouveurs arrivent à prouver les obligations de preuve de raffinement.

On peut constater sur la figure 3.10 que les prouveurs arrivent à prouver les obligations de preuve de raffinement.

3.3.2 Design Pattern Observer

Le principe du design pattern Observer est de définir une relation entre objets de type un-à-plusieurs, de façon que, si un objet change d'état, tous ceux qui en dépendent en soient informés et mis à jour automatiquement.

Cet exemple est un cas classique où l'utilisation des invariants de classe est difficile. Dans son papier [29], *Parkinson* remet en cause l'utilisation des invariants de classes et propose de revenir à un concept plus général et d'utiliser les prédicats pour spécifier les propriétés de la structure générale du programme. Un invariant de classe ne serait alors qu'un prédicat particulier. Il déclare que le design pattern Observer ne peut pas être traité avec la méthode Boogie.

```
unit simupack_add(Calc[0] this, real x)
 requires ∃ bag c ; c == union(this.coll,single(x)) &&
glue_inv(c,this) ;
 behavior default :
 assigns this.coll ;
 ensures this.coll == union(\old(this.coll),single(x)) &&
glue(this) &&
 \ mutable(this,Calc) && this.committed == false ;
;
unit add(Calc[0] this, real x)
 requires \mutable(this,Calc) && this.committed == false ;
 behavior default :
 assigns this.coll,this.count,this.sum ;
 ensures this.coll == union(\old(this.coll),single(x)) &&
 \mutable(this,Calc) && this.committed == false ;
{
 unpack(this) ;
 this.count++ ;
 this.sum += x ;
 simupack_add(this,x) ;
}
```

FIG. 3.7 – Simulation du **pack with** dans la méthode `add`

```
real mean(Calc[0] this)
 requires \mutable(this,Calc) && this.committed == false &&
 this.coll != emptybag() ;
 behavior default :
 assigns \nothing ;
 ensures \result == sumbag(this.coll) / cardinal(this.coll) ;
{  return this.sum / this.count ;
}
```

FIG. 3.8 – La méthode *mean* du calculateur en Jessie

```
lemma hint0 : \ real_of_integer(2) == 2.0
lemma hint2 : ∀ integer i ; i != 0 ==> \real_of_integer(i) != 0.0
lemma hint1 : (14.0 + 17.0) / \real_of_integer(2) == 15.5
unit f(Calc[0] c)
 requires \mutable(c,Calc) && c.committed == false ;
{  var real m ;
 reset(c) ;
 add(c,14.0) ;
 add(c,17.0) ;
 m = mean(c) ;
 assert cardinal(c.coll) == 2 ;
 assert sumbag(c.coll) == 31.0 ;
 assert m==15.5 ;
}
```

FIG. 3.9 – Programme test

Voyons ça plus en détails.

Solution 1 : Invariant dans la classe Observer

On propose une première solution dans laquelle c'est la classe *Observer* qui est annotée par un invariant $I_{Observer}$: $this.sub.val = this.cache$. Cette invariant fait référence à un champs $this.sub$. Par conséquent, *sub* doit être un champs **rep** dans *Observer* (figure 3.11).

Etant donnée cette solution, considérons le bout de code suivant :

$s = newSubject()$;
$o_1 = newObserver(s)$;
$o_2 = newObserver(s)$;

Une fois l'objet s alloué, le champ $s.committed$ a pour valeur $false$ et $s.inv = Subject$, ce qui vérifie la pé-condition du constructeur de *Observer*. Ce dernier peut donc être invoqué pour la création du premier *Observer* o_1. Mais, l'instruction **pack** à la fin du constructeur, affecte la valeur *true* au champ $s.committed$ (ce qui est exprimé en post-condition), vu que s appartient à o_1. En conséquence, pour la deuxième invocation du constructeur de *Observer*, pour la création de l'objet o_2, la pre-condition n'est plus vérifiée et donc la méthode non exécutée.

En résumé, cette solution impose qu'un objet *Subject* n'ait qu'un seul *Observer*. Techniquement, ceci est du au fait que dans Boogie, un objet ne peut avoir qu'un seul propriétaire (owner) et donc une instance de *Subject* ne peut appartenir qu'à un seul *Observer* ?

Une solution alternative serait de mettre un invariant dans la classe *Subject*.

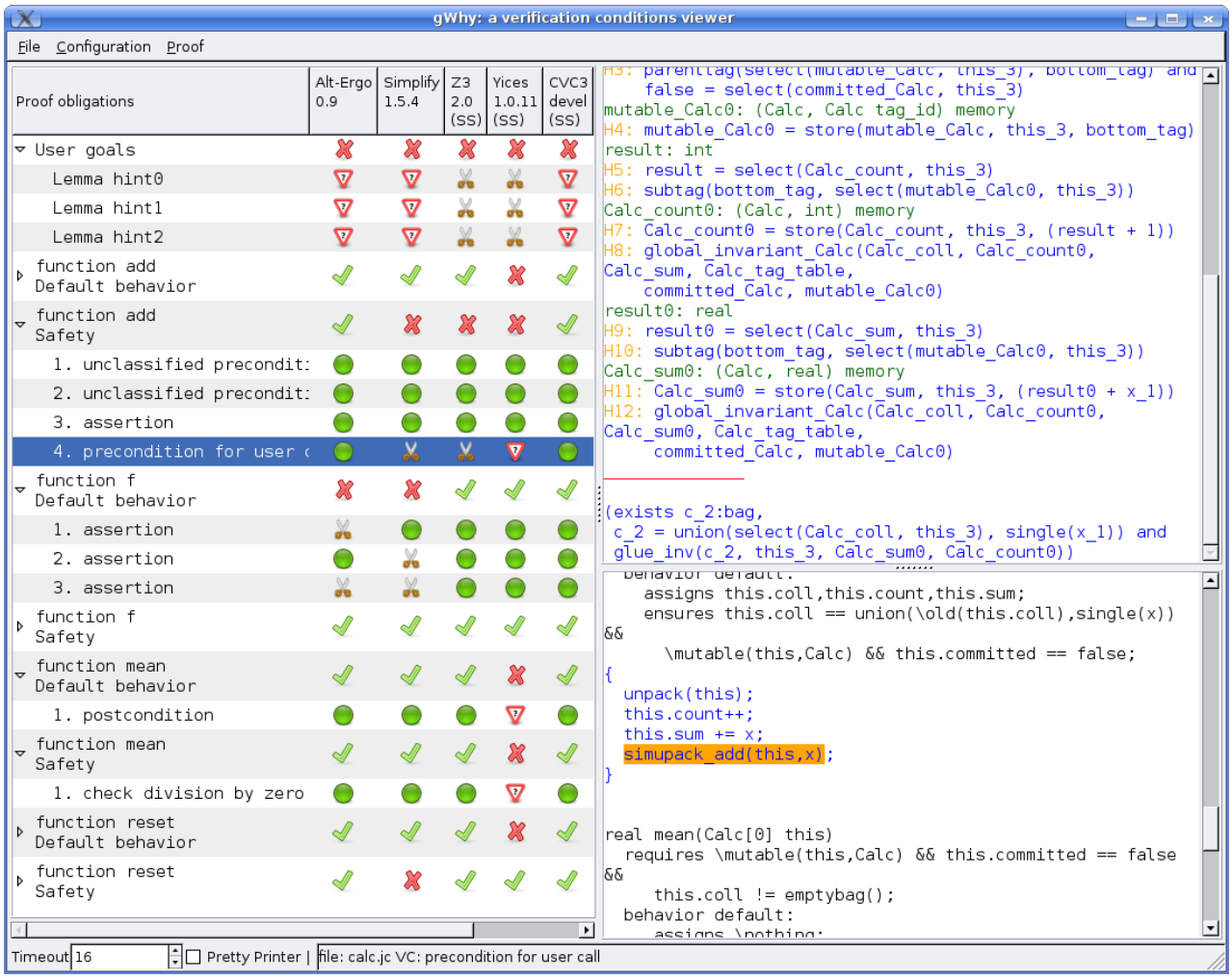

FIG. 3.10 – Preuves du calculateur de Morgan en jessie

```
class Subject {
Collection<Observer> obs;
int val;
Subject() {
obs = new List();
pack(this);
}
//requires this.inv = Subject ∧ ∀o ∈ obs • o.inv = Object;
//ensures ∀o ∈ obs • o.inv = Object;
void register(Observer o) {
this.obs.add(o);
o.notify(this.val);
}
//requires this.inv = Subject;
//ensures this.inv = Subject;
void update(int n) {
this.val = n;
foreach(Observer o :obs)
unpack(o);
o.notify(this.val);
pack(o);
}

class Observer {
rep Subject sub;
int cache;
//@invariant I_O : this.sub.val = this.cache;
//requires ¬s.committed ∧ s.inv = Subject;
//ensures s.committed ∧ this.inv = Observer;
Observer(Subject s) {
this.sub = s;
sub.register(this);
pack(this);
}
//requires this.inv = Object;
//ensures this.inv = Object;
void notify(int n) {
this.cache = n;
}
int val() {
return cache;
}
```

FIG. 3.11 – Invariant dans la classe Observer

Solution 2 : Invariant dans la classe Subject

Dans cette solution, c'est donc la classe *Subject* qui est annotée par un invariant. Par exemple, $\forall Observer o, o \in obs \Rightarrow this.o.cache = this.val$. De la même façon que dans la première solution, vu que l'invariant référence *o.cache* pour tout *o* appartenant à la collection *obs*, alors le *obs* est un champ **rep** dans *Subject*.

On fait l'hypothèse qu'un modificateur **rep** sur une collection d'objet est interprété comme un **rep** sur chaque élément de la collection. Nous reparlerons de ce point dans la conclusion.

Avec cette solution, on peut ajouter autant d'objet *Observer* qu'on le veut à un objet *Subject* donné.

```
class Subject {
rep Colelction<Observer> obs;
int val;
//@invariant I_S : ∀Observero, o ∈ obs ⇒ this.o.cache = this.val;
//ensures this.inv = Subject;
Subject() {
obs = new List();
pack(this);
}
//requires this.inv = Subject ∧ ¬this.committed ∧ o.inv = Observer;
//ensures this.inv = Subject ∧ ∀o ∈ obs • o.committed;
void register(Observer o) {
unpack(this);
this.obs.add(o);
o.notify(this.val);
pack(this);
}
//requires this.inv = Subject ∧ ¬this.committed;
//ensures this.inv = Subject;
void update(int n) {
unpack(this);
this.val = n;
foreach(Observer o :obs)
o.notify(this.val);
pack(this);
}

class Observer {
Subject sub;
int cache;
//requires s.inv = Subject ∧ ¬s.committed;
//ensures this.inv = Observer ∧ this.committed;
Observer(Subject s) {
this.sub = s;
pack(this);
sub.register(this);
}
//requires this.inv = Observer ∧ ¬this.committed;
//requires this.inv = Observer ∧ ¬this.committed ∧ this.cache =
    this.sub.val;
void notify(int n) {
unpack(this);
this.cache = n;
pack(this);
}

int val() {
return cache;
}
```


FIG. 3.12 – Invariant dans la classe Subject

Chapitre 4

Conclusion

4.1 Conclusion

On a exposé dans ce document une approche combinant les concepts d'ownership, de champs modèles et d'invariant de collage, et permettant de supporter le raffinement de programme Java.

Deux principaux résultats ont été énoncés et prouvés ; le premier concerne la préservation des invariants et le second est relatif à la modularité du raffinement.

4.2 Travaux reliés

Comme on a pu le constater en étudiant l'exemple du design pattern Observer, on ne sait pas vraiment ce que veut dire un modificateur **rep** sur une collection d'objets. Est-ce que chaque objet de la collection est rep, ou bien est que c'est l'objet représentant la collection qui est rep ?

L'invariant de la classe *Subject*, $\forall Observer o, o \in obs \Rightarrow this.o.cache = this.val$, dépend des éléments de la collection *obs*, et non pas de ces champs comme l'exige la méthode Boogie. Pour notre part, nous avons juste fait une hypothèse. Mais il est clair que c'est beaucoup plus compliqué que ça. Des travaux sont effectués dans ce sens, notament dans le cadre de l'ARC CePromi [6, 13].

4.3 Perspectives

Un problème auquel on a été confronté, est que ça peut facilement devenir pesant de devoir spécifier, à chaque méthode, dans les pre/post conditions des méthodes les valeurs des champs *inv* et *committed*. Certains choix peuvent être faits pour simplifier les annotations des méthodes. Une idée qui pourrait

être intéressante, serait de définir une valeur que prendrait le champ lorsque sa valeur n'est pas mentionné explicitement, comme ce qui se fait dans Spec#.

Dans l'approche qu'on a proposée, on n'a considéré que le raffinement à deux niveaux. Mais on a bien dit lorsqu'on a présenté le raffinement, qu'il pouvait se faire à plusieurs niveaux comme c'est le cas dans la méthode B. Dans [25], on présente un exemple dans lequel on a un premier niveau d'abstraction où est définie une interface Queue repésentant les files de priorité. On a ensuite un second niveau d'abstraction dans lequel des décisions qui concernent l'implantation auquel il est question d'aboutir sont prises. Par exemple, une représentation classique des files prioritaires, c'est les *heap* qui sont des arbres binaires. Arrive enfin, le niveau concret, où sont représentés des tableaux Java dans lesquels on stocke les éléments des arbres binaires. Il devrait être possible d'étendre notre approche à ce type de programmes.

Une implantation complète reste à faire, mais nous avons pu tester sur le calculateur de Morgan que les obligations de preuve générées sont accessibles aux prouveurs automatiques. Mais avant d'implanter, il serait souhaitable de résoudre clairement le problème des champs rep sur les collections.

Bibliographie

[1] M. Abadi and L. Lamport. The existence of refinement mappings. 1988. Technical report, Digital Systems Research Center, Palo Alto, California.

[2] J.R. Abrial. *The B-Book*. Cambridge University Press, 1996.

[3] R.J. Back. *On the correctness of refinement in program development*. PhD thesis, University of Helsinki, 1978.

[4] R.J. Back. A calculus of refinements for program derivations. *Acta Informatica*, page 593–624, 1988.

[5] Romain Bardou. Invariants de classe et systèmes d'ownership. Master's thesis, Master Parisien de Recherche en Informatique, 2007.

[6] Romain Bardou. Ownership, pointer arithmetic and memory separation. In *Formal Techniques for Java-like Programs (FTfJP'08)*, Paphos, Cyprus, July 2008.

[7] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6) :27–56, 2004.

[8] Clark Barrett and Cesare Tinelli. Cvc3. In W. Damm and H. Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07), Berlin, Germany*, Lecture Notes in Computer Science. Springer, 2007.

[9] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL : ANSI/ISO C Specification Language*, 2008. http://frama-c.cea.fr/acsl.html.

[10] S. Boulmé and M.-L. Potet. Interpreting invariant composition in the B method using the Spec# ownership relation : a way to explain and relax B restrictions. In *B 2007*, volume 4355 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

[11] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 2004.

[12] C.A.R.Hoare. An axiomatic basis for computer programming.

[13] Arthur Charguéraud and François Pottier. Functional translation of a calculus of capabilities. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 213–224, September 2008.

[14] C.Morgan. Programming from specifications. 1990.

[15] Sylvain Conchon and Évelyne Contejean. The Alt-Ergo automatic theorem prover, 2008. http://alt-ergo.lri.fr/.

[16] Leonardo de Moura and Nikolaj Bjørner. Z3, An Efficient SMT Solver. http://research.microsoft.com/projects/z3/.

[17] Leonardo de Moura and Bruno Dutertre. Yices : An SMT Solver. http://yices.csl.sri.com/.

[18] David Detlefs, Greg Nelson, and James B. Saxe. Simplify : a theorem prover for program checking. *J. ACM*, 52(3) :365–473, 2005.

[19] E.W. Dijkstra. A discipline of programming. 1976.

[20] Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of C programs. In Jim Davies, Wolfram Schulte, and Mike Barnett, editors, *6th International Conference on Formal Engineering Methods*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29, Seattle, WA, USA, November 2004. Springer.

[21] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177, Berlin, Germany, July 2007. Springer.

[22] K.R.M. Leino and P. Müller. Object invariants in dynamic contexts. In *ECOOP*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer-Verlag, 2004.

[23] K.R.M. Leino and P. Müller. A verification methodology for model fields. In *ESOP*, volume 3924 of *Lecture Notes in Computer Science*, pages 115–130. Springer-Verlag, 2006.

[24] Claude Marché. Jessie : an intermediate language for Java and C verification. In *PLPV '07 : Proceedings of the 2007 workshop on Programming Languages meets Program Verification*, pages 1–2, Freiburg, Germany, 2007. ACM.

[25] Claude Marché. Towards modular algebraic specifications for pointer programs : a case study. In Hubert Comon-Lundh, Claude Kirchner, and Hélène Kirchner, editors, *Rewriting, Computation and Proof*, volume 4600 of *Lecture Notes in Computer Science*, pages 235–258. Springer, 2007.

[26] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in

JML. *Journal of Logic and Algebraic Programming*, 58(1–2) :89–106, 2004. http://krakatoa.lri.fr.

[27] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, March 2000.

[28] J.M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9 :287–306, 1987.

[29] Matthew Parkinson. Class invariants : The end of the road ? http://www.cl.cam.ac.uk/~mjp41/InvariantsPositionPaper.pdf.

[30] R.W.Floyd. Assigning meanings to programs.

[31] Jan Smans Wolfram Schulte, Songtao Xia and Frank Piessens. A glimpse of a verifying c compiler.