Sémantique et analyse de programmes au delà des valeurs numériques

David Monniaux

CNRS / VERIMAG

Mardi 6 avril 2010

Plan

Composition parallèle

Les *threads* $P_1, \ldots, P_n$ s'exécutent en parallèle, noté $P_1 \| \ldots \| P_n$.

Chaque *thread* P_i a

- un ensemble d'états Σ_i
- un état initial $\sigma_i^{(0)} \in \Sigma_i$
- une relation de transition $\rightarrow_i$

L'ensemble des états du programme parallèle est $\Sigma = \Sigma_1 \times \ldots \Sigma_n$, un état du programme parallèle est $(\sigma_1, \ldots, \sigma_n)$, et la relation de transition est

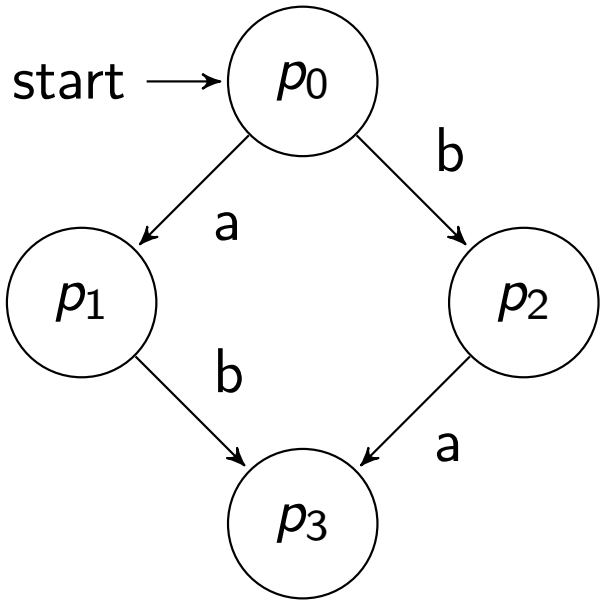
$$(\sigma_1, \ldots, \sigma_n) \rightarrow (\sigma'_1, \ldots, \sigma'_n) \iff \exists i \; \sigma_i \rightarrow_i \sigma'_i \wedge \forall j \; i \neq j \Rightarrow \sigma_j = \sigma'_j$$

Synchronisations

- Si chaque relation de transition ne dépend que de l'état local, alors on pourrait analyser chaque programme indépendamment.
- Le vrai problème est si la relation de transition dépend d'un état global (= état des autres *threads*) — s'il y a des **interactions** entre threads.

Remarque banale : si des parties des relations de transition σ_1 et σ_2 ne "regardent" pas l'état de l'autre *thread*, alors on peut faire "commuter" les transitions correspondantes.

Commutation



Inutile de considérer les deux **entrelacements** *ab* et *ba* vu qu'ils ont le même résultat.

Mutex

Synchronisation assurée par **exclusion mutuelle** : si un *thread* essaye d'acquérir un *mutex* déjà acquis par un autre, alors il doit attendre que l'autre l'ait rendu.

Revient à supprimer certaines des possibilités dans la relation de transition produit.

- Si plusieurs *threads* attendent que le même *mutex* se libère, lequel est réactivé ? **Non-déterminisme.**
- Encore faut-il savoir que c'est le même *mutex* : et s'ils sont désignés via des pointeurs ?

Analyses possibles

- *Deadlock-freedom* : en aucun cas, on ne peut se retrouver dans une s ituation où un *thread* ne peut redémarrer (A bloqué par B bloqué par C bloqué par A, etc.).
- *Race conditions* : une *race condition* est une situation où la sortie d'un calcul dépend du hasard du non-déterminisme avec un conflit d'accès à la mémoire.

Par exemple, deux threads exécutent compteur++. Ça s'interprète comme :

```
int tmp = compteur;
tmp = tmp + 1;
compteur = tmp;
```

Si les deux *threads* lisent tmp en même temps, le compteur n'est incrémenté qu'une fois !

Accès en mémoire partagée

Deux solutions :

- Analyser systématiquement les **race conditions** et refuser les programmes pour lesquels on ne peut justifier leur absence.
- Admettre que certaines opérations (lecture ou écriture d'un mot de 32 bits en mémoire) sont **atomiques** : elles s'exécutent en une fois. Une exécution de programme est donc vue comme une succession d'opérations atomiques, un **entrelacement** des actions des différents *threads* (cf produit de systèmes de transition).
  Utile car exiger des verrous systématiques est coûteux ! (P.ex. pour *garbage collectors* parallèles.)

Malheureusement ce deuxième modèle est faux sur de nombreuses architectures multi-cœurs / multi-processeurs !

Difficultés de l'analyse

- Pour connaître les accès en mémoire partagée, il faut une analyse d'alias entre les *threads*.
- Pour faire une analyse, il faut connaître le flot de contrôle (la séquence des instructions exécutées).
- Pour connaître le flot de contrôle, il faut une analyse des *mutex*. Certains sont accédés par pointeurs.
- Le flot de contrôle dépend des tests, qui dépendent des valeurs en mémoire y compris en mémoire partagée.

Un peu un problème d'œuf et de poule !

Une idée : raffinements successifs

Approche proposée par Aarti Gupta (NEC) :

- On analyse chaque *thread* séparément avec une vision très conservatrice de ce qui est partagé ou non (les variables locales à une procédure dont on ne prend pas l'adresse ne sont pas visibles des autres *threads*). Ce qui vient des autres *threads* est considéré comme n'importe quoi.
- On en déduit des choses sur les *mutex* et ce qui est éventuellement partagé.
- On raffine ce que l'on sait sur le flot de contrôle.
- On réanalyse. On en sait plus sur le contenu des variables...
- On raffine ce que l'on sait sur le flot de contrôle. Etc.

Anecdotes

Plusieurs sondes spatiales américaines ont eu des problèmes de *deadlocks* ou *race conditions*.

Quand on a trouvé un *race condition*, rajouter un verrou n'est pas forcément une bonne idée :

- Le verrou pourra causer un *deadlock*.
- Il est possible que la *race condition* soit rare.

D'où le développement de méthodes formelles d'analyse (*bounded model checking*...).

Recherche effective des *deadlock*

On peut faire une sorte de *bounded model checking* abstrait et rechercher des chemins de taille bornée qui créent des *deadlocks* et *race conditions*.

On abstrait violemment le programme (pas de prise en compte des valeurs numériques, p.ex.).

Cette technique

- Ne garantit pas de trouver tous les problèmes (on pourrait avoir une *race condition* à profondeur $n+1$, on ne la trouvera pas à profondeur n).
- Peut signaler des problèmes qui n'existent pas en réalité (on n'a pas pris en compte les valeurs du programme, donc éventuellement on prend en compte des exécutions qui n'existent pas en vrai).

Exemple

```
if (nbthreads > 0) {
  lock(ptr->lck);
}
ptr->counter++;
if (nbthreads > 0) {
  unlock(ptr->lck);
}
```

```
if (nondet()) {
  lock(ptr->lck);
}
read(ptr->counter);
write(ptr->counter);
if (nondet()) {
  unlock(ptr->lck);
}
```

- Si on ne tient pas compte de la valeur de la variable nbthreads et du fait qu'elle est non nulle dès qu'il y a plusieurs *threads*, on signale une *race condition* sur $*\text{ptr} = 0$.
- Si on ne tient pas compte que le booléen nbthreads > 0 a la même valeur dans les deux tests, on peut signaler un problème inexistant de déséquilibre de lock() et unlock().

Moralité

- Le modèle est fortement complexifié par les *threads*.
- Les **accès mémoires sans verrouillage explicite**, c'est **risqué, compliqué, difficile et délicat** et la sémantique en multi-processeurs ou multi-cœurs dépend finement du modèle mémoire de l'architecture machine (souvent non documenté précisément).
  cf POPL09
- Les pointeurs complexifient grandement l'analyse des *threads* vu qu'on ne sait pas syntaxiquement où est la mémoire partagée.

Plan

Modèle mémoire simple

On essaye de modéliser un nombre grand voire non borné de cellule mémoires utilisées dans le programme à analyser par un nombre fini, assez petit, de cellules mémoires abstraites.

Chaque cellule mémoire abstraite peut représenter plusieurs cellules mémoires concrètes.

- *May-alias* : deux pointeurs peuvent éventuellement désigner la même cellule mémoire.
- *Must-alias* : deux pointeurs désignent forcément la même cellule mémoire.

Les tableaux

Choix de modélisation possibles :

1. On ignore les tableaux (lecture donne $\top$, écriture ne fait rien à part vérifier les bornes).
2. Tous les éléments du tableau correspondent à la même variable abstraite.
3. Si le tableau a une taille fixée, on distingue toutes les cases (donc une variable abstraite pour $t[0]$, une pour $t[1]$ etc.).
4. On partitionne dynamiquement le tableau pour représenter des propriétés plus compliquées (sujet actif de recherche).

Le choix 3 est le plus précis, mais est coûteux.

Exemple pratique

On distingue toutes les cases, chacune abstraite par un intervalle :

```
int t[6] = {0, 0, 0, 0, 0, 0};
int i = bidule();
if (i < 0 || i > 3) i=0;
/* i dans 3..5 */
t[i] = 3;
assert(t[i] == 3);
```

Après t[i] = 3, les cases 0 à 2 contiennent $[0, 2]$, les cases 3 à 5 contiennent $[0, 3]$. La relecture de t[i] donne $[0, 3]$, donc l'analyseur n'arrive pas à prouver que l'assertion n'est pas violée !

(Quand on fait une écriture x = e, si x désigne un *may-alias* alors on doit **ajouter** la valeur e aux cases mémoires concernées... et non remplacer leur valeur !)

Comment récupérer la situation

On fait une **analyse d'alias** qui détecte quand deux expressions désignent forcément la même expression, et on utilise ce résultat :

```
int t[6] = {0, 0, 0, 0, 0, 0};
int i = bidule();
if (i < 0 || i > 3) i=0;
/* i dans 3..5 */
tmp = 3;
t[i] = 3;
assert(tmp == 3);
```

Et là on prouve que l'assertion n'est pas violée !

Quand ça se passe mal

Si on a fait une analyse très imprécise, p peut pointer sur de très nombreuses cases abstraites (bien plus que si l'analyse était précise) et donc a analyser $*p = 0$ peut invalider beaucoup d'informations !

Phénomène de « blob » : on finit par avoir des pointeurs qui pointent éventuellement sur une grande partie des variables du programme (plutôt, on ne peut prouver qu'elles ne pointe pas sur ces variables). Que faire ?

- Si on applique correctement $*p = 0$ alors on va rajouter 0 dans des tas de variables non concernées, d'où éventuellement imprécisions supplémentaires, fausses alarmes etc. (e.g. si $*p$ est de type pointeur, alors on va rajouter le pointeur null, donc ensuite avertissements d'utilisation de pointeurs nuls).
- Sinon, on peut, **incorrectement mais pragmatiquement**, sauter $*p = 0$.

Allocation dynamique

Dans un langage avec malloc() :

- On peut envoyer toutes les cases allouées par |malloc()| dans la même case abstraite. Mauvais.
- On peut distinguer les |malloc()| par ligne de programme :

```
char *buf1 = malloc(10);
char *buf2 = malloc(40);
```

On distinguera le bloc pointé par buf1 du bloc pointé par buf2.

Cela suffit-il ?

Mon malloc à moi

De nombreux programmeurs ou bibliothèques « enveloppent » le malloc() :

```
void *mymalloc(size_t l) {
  void *p = malloc(l);
  if (p == NULL) {
    fprintf(stderr, "Out␣of␣memory\n");
    exit(1);
  }
}
```

Avec la méthode plus haut, tous les malloc() viennent de la même ligne (dans mymalloc()).
Solution : détecter ce cas de figure.

Pointeurs

Dans chaque case abstraite modélisant un pointeur, stocker l'ensemble des cases où elle peut pointer.

```
char *p = malloc(10);
char *q = malloc(40);
char *r;
if (toto()) r=p; else r=q;
```

Arithmétique de pointeurs

- Dans les langages comme Java ou Caml, une référence pointe sur un objet et non sur une partie d'un objet.
- En C, un pointeur peut pointer au milieu d'une **struct** ou d'un tableau.

Modélisation des pointeurs par : bloc de base + déplacement (entier).

Plan

Définition des alias

Deux expressions e_1 et e_2 désignant des données en mémoire sont dites des *alias* l'une de l'autre si elles désignent la même valeur.
P.ex. dans :

```
int t[10];
int *x = t+3;
```

Les expressions $*x$ et `t[3]` sont des alias.

L'analyse d'alias consiste à calculer une **sur-approximation** de la relation d'alias (*may-alias*) et éventuellement une **sous-approximation** (*must-alias*).

Exemple : analyse de Steensgard

[Steensgard POPL 1996]
Analyse simple mais peu précise. Grosso modo :

- On partitionne les variables pointeurs.
- Au début, chaque variable etc. est seule dans sa classe.
- Dès que l'on voit une assignation x = y, on fusionne les classes de x et y.

À la fin, si deux variables sont dans des classes différentes, on sait qu'elles ne peuvent pointer sur le même objet.

Analyse d'échappement

On veut détecter quelles valeurs peuvent « s'échapper » d'une fonction, d'une procédure, d'un module, d'un *thread*...

```
int machin() {
  char *p = malloc(10);
  /* on calcule sur p sans appeler d'autre fonction
  free(p);
}
```

Le bloc alloué ne « sort » pas de la fonction, même s'il est apparemment dans le tas global $\Rightarrow$ on peut l'analyser comme une variable locale.

Exemple : suppression du verrouillage en Java

```
String toto() {
  StringBuffer buf = new StringBuffer("foobar");
  for(int i=0; i<1000; i++) {
    buf.append("*" + i);
  }
  return buf.toString();
}
```

Ce programme va verrouiller et déverrouiller le *mutex* associé au StringBuffer référencé par buf 1000 fois pour rien, car buf.append(...) est **synchronized**.

Différentes solutions

- Pour de la compilation Java hautement optimisée, on détecte statiquement des appels à des méthodes **synchronized** alors que l'objet ne s'échappe pas du *thread*, et on les remplace par des appels à des clones des méthodes ou des classes appelées, sans **synchronized**.
- Ceci demande une analyse non triviale !
- Solution plus simple : on a créé la classe StringBuilder (comme StringBuffer mais non synchronisée), et c'est au programmeur de réfléchir.

Allocation en pile

```
void toto(Vector3D u, Vector3D v) {
  Vector3D w = u.copy();
  w.translate(v);
  return w.norm();
}
```

Il est bête d'allouer w dans le tas alors que son usage est purement local. Si on détecte qu'il ne s'échappe pas, on peut l'allouer **en pile**.

Plan

Les nombres en virgule flottante

Correspondent aux type C **float** et **double**. Généralement mis en œuvre par flottants à la norme IEEE-754.

Points positifs :

- Les opérations flottantes IEEE-754 $\oplus$, $\ominus$, $\otimes$, $\oslash$, $\sqrt{}$ sont **déterministes** : le résultat est exactement fixé par la norme.
- L'erreur d'arrondi $|(x \oplus y) - (x + y)|$ peut être **bornée**.

Définition des flottants

Grossièrement, $\pm mantisse.2^{exposant}$, avec *mantisse* d'un nombre limité de bits. Le premier bit de mantisse est supposé égal à 1, on ne le stocke pas.

- IEEE-754 simple précision (souvent, type **float** du C) : 23 bits de mantisse, 8 bits d'exposant (y compris son signe), 1 bit de signe = 32 bits.
- IEEE-754 double précision (souvent, type **double** du C) : 52 bits de mantisse, 11 bits d'exposant (y compris son signe), 1 bit de signe = 64 bits.
- Intel 32 bits sous Linux ou Windows, type **long double** du C : 64 bits de mantisse, 15 bits d'exposant, 1 bit de signe = 80 bits.

En fait c'est un peu plus compliqué à cause des nombres dénormalisés, de $\pm\infty$, et de *not-a-number* (NaN, signale des erreurs).

Flottants, les difficultés

En fait, pas si déterministe et bien défini qu'on ne le croit :

- Les opérations flottantes ne sont **pas associatives**. En général $(x \oplus y) \ominus y \neq x$. Par exemple, $x = 1$, $y = 10^{10}$, $x \oplus y = 10^{10}$, $(x \oplus y) \ominus y = 0$.
- Certains compilateurs (p.ex. Intel icc) se permettent par défaut des optimisations par associativité, interdites par la norme. Pour gcc, ces optimisations ne sont permises qu'avec `-ffast-math`.
- Sur certaines architectures, il existe une instruction *fma* (*fused multiply-add*) combinant multiplication et addition. Autrement dit, $a \times b + c$ peut, suivant le contexte, être compilée comme $fma(a, b, c)$ ou $a \otimes b \oplus c$.
- Le cas ennuyeux des Intel 32 bits (i386, i486, Pentiums...), où le résultat du calcul peut dépendre de l'optimisation des variables en registres.

Exemple amusant pour WP

```
void do_nothing(double *x) { }
int main(void) {
  double x = 0x1p-1022, y = 0x1p100, z;
  do_nothing(&y);
  z = x / y;
  if (z != 0) {
    do_nothing(&z);
    assert(z != 0);
  }
}
```

0x1p−1022 est une notation C99 pour 2^{-1022}, 0x1p100 est 2^{100}.
assert(cond) signale une erreur à l'exécution si cond n'est pas vraie.

On pourrait croire que assert(z != 0) ne signale jamais d'erreur ?

Des différences subtiles

Avec gcc 4.0 sur Linux / Pentium 32 bits :

- Si compilé sans optimisation, x / y est calculé comme un **long double** (80 bits) non nul, puis converti en un **double** nul pour être stocké dans z, qui est rechargée de la mémoire pour le test z != 0. Le **if** n'est donc pas pris.
- Si compilé en un seul code source avec optimisation, gcc se rend compte que do_nothing() ne fait rien, propage les constantes, se rend compte que z est nul, donc que le **if** n'est pas pris et qu'au final main() ne fait rien. main() est donc compilée comme une fonction vide.

Un comportement peu intuitif

Si do_nothing() et main() sont dans deux codes sources séparés, gcc ne sait pas que do_nothing() ne fait rien, et en particulier pas qu'il ne change pas la variable passée en pointeur.
Le test z != 0 est fait sur un **long double** non nul gardé en registre. Mais pour l'appel do_nothing(), z est rechargée de la mémoire avec la valeur nulle. **L'assertion échoue.**

Les approches basées sur *weakest precondition* sans précautions particulières considèrent que les deux expressions z != 0 sont équivalentes et que donc l'assertion ne peut échouer.

On a un **problème de modélisation sémantique**.

Autre exemple, toujours sur Intel

```
static inline double f(double x) {
  return x/1E308;
}
double square(double x) {
  double y = x*x;
  return y;
}
int main(void) {
  printf("%g\n", f(square(1E308)));
}
```

1E308 note une valeur approchée de 10^{308}.

Suivant optimisation, ce programme affiche $+\infty$ ou une valeur approchée de 10^{308}.

Analyse par interprétation abstraite

- Pour les **intervalles**, on calcule $[l, h] \oplus [l', h']$ en calculant $l \oplus l'$ en arrondi vers le bas et $h \oplus h'$ en arrondi vers le haut.
- Pour les autres domaines (polyèdres etc.), dont les calculs sont définis en rationnels, on utilise le fait que $x \oplus y = x + y + \varepsilon$ et que ε est borné en fonction de $|x + y|$.

Moralité

- Il est dangereux de raisonner sur les flottants comme s'il s'agissait de réels.
- Il est dangereux de supposer que les compilateurs respectent la norme.
- Il est plus facile de définir la sémantique des flottants sur le code assembleur que sur le code source.
- Déboguer un programme en flottants peut être difficile. En général, on débogue mieux sur du code non optimisé, or l'optimisation change la sémantique.
- Changer de processeur, de compilateur, de système d'exploitation peut changer les résultats en flottants même si tout le monde est IEEE-754.
- Les méthodes par *weakest precondition* sont assez fragiles par rapport aux problèmes de sémantiques mal définies.
- Les méthodes par interprétation abstraite résistent mieux.

Plus de détails dans [Monniaux TOPLAS '08].

Plan

Les entiers machine sont bornés

Il est plus simple de raisonner comme si les entiers étaient $\mathbb{Z}$. Mais en C, les types entiers sont bornés !

Une addition entre deux **unsigned int** qui dépasse la valeur maximale du type $(2^n - 1)$ est calculée **modulo 2^n** (en pratique, $n = 16$ ou 32).

Une addition entre deux **signed int** qui sort des bornes rend un **résultat indéfini** (en pratique, calcule modulo 2^n).

2 solutions :

- On interdit les dépassements (cas de la méthode B).
- On les permet suivant la politique de l'utilisateur, et on gère correctement les modulos.

Quelques problèmes de normalisation

La norme du langage C ne précise pas la taille des types entiers (elle donne des tailles minimales).

Sur certaines architectures (microcontrôleurs 16 bits), **int** est de 16 bits et **long** de 32 bits.

Sur certaines architectures courantes (Intel et PowerPC 32 bits), **int** est de 32 bits, **long** est de 32 bits (et **long long** de 64 bits).

Sur d'autres architectures courantes (Intel 64 bits), **int** est de 32 bits, **long** et **long long** de 64 bits.

- Toute analyse doit donc être paramétrée par les données de l'architecture utilisée.
- On ne peut pas simplement se contenter de tester sur PC Pentium avant d'embarquer sans précaution sur microcontrôleur.

Le C est plein de pièges

```
short toto(short x) {
  return (x * 3) / 5;
}
```

Les **short** font 16 bits. Que retourne toto(30000) ?

Le C est plein de pièges

```
short toto(short x) {
  return (x * 3) / 5;
}
```

Les **short** font 16 bits. Que retourne toto(30000) ?

Première hypothèse : $(3 \times 30000) \bmod 2^{16}$ donne 24464, divisé par 5 donne 4892.

Le C est plein de pièges

```
short toto(short x) {
  return (x * 3) / 5;
}
```

Les **short** font 16 bits. Que retourne toto(30000) ?

Première hypothèse : $(3 \times 30000) \bmod 2^{16}$ donne 24464, divisé par 5 donne 4892.

Non ! 3 est promu en **int**, $3 \times 30000 = 90000$ divisé par 5 donne 18000.

Dans les opérations arithmétiques, les **short** sont promus en **int** même si l'opération a lieu entre deux **short**.

En revanche, les **int** ne sont pas promus en **long** même si un dépassement est possible.

Plan

Quelques avantages du typage objet à la Java

- Typage propre et strict. Pas de conversions pointeurs / entiers, pas de **void** $*$. Transtypage contrôlé vers les classes filles.
- On ne peut prendre de références que sur les objets, pas sur des parties d'objets. En C, on peut prendre un pointeur sur un champ de structure.
- L'**encapsulation** garantit que les champs **private** ne sont pas modifiés par les méthodes des autres classes.
- Deux objets de classes différentes ne peuvent être aliasés.

Invariant de classe

Propriété I :

- Vraie à la sortie des constructeurs (ou seulement des constructeurs et fonctions publiques).
- Maintenue par les fonctions publiques (y compris celles des classes dérivées... voir mécanismes de protection).

Un invariant de classe est forcément vrai entre les appels des fonctions publiques.

Exemple : tampon circulaire

```
class Tampon {
  int i;     double t[];
  Tampon(int n) {
    i = 0;
    t = new double[n];
  }
  void ajouter(double x) {
    t[i] = x;
    i++;
    if (i >= t.length) i=0;
    /* il y en manque */
  }
}
```

On prouve facilement l'invariant de classe $0 \leq i < t.length$.

Algorithme

Algorithme (simplifié) pour les invariants de classe :

- calcul de point fixe surapproximé (élargissements etc.)
- initialisation par sortie des constructeurs
- itérations par passage des méthodes publiques

Plan

Correction ?

- Analyseurs « test syntaxique ».
- Analyseurs non corrects qui suivent le flot de contrôle pour quelques propriétés. E.g. Coverity.
- Analyseurs partiellement corrects. E.g. Microsoft Device Driver Verifier.
- Analyseurs corrects (sauf bug et quelques finesses). E.g. PolySpace, Astrée

Si on renonce à la correction, on peut supprimer des fausses alarmes.

E.g. si l'analyseur ne comprend pas des instructions, si le programme utilise des pointeurs trop aliasés, on saute les instructions !

Deux usages différents

- **Trouver des bugs**. On peut se permettre d'ignorer des bugs (incorrection) mais on veut passer sur du code pénible (C sans restrictions, pointeurs) avec peu de fausses alarmes.
  Exemple : Coverity
- **Prouver leur absence**. On doit être rigoureux. On risque de produire plus de **fausses alarmes**.
  - Trier les fausses alarmes par « vraisemblance » ?
  - Spécialiser l'analyseur pour une classe de programmes ?

Les questions à poser au fournisseur

- Quelle classe de bugs l'analyseur détecte-t-il ?
- Les détecte-t-il tous ? (Correction)
- Produit-il des fausses alarmes ? (Incomplétude)
- Quelles suppositions fait-il sur le programme, la plate-forme ?
- Modèle-t-il correctement les entiers machine, les flottants, la mémoire partagée ?