Raisonnement séquentiel et concurrent

Marc Shapiro, UPMC-LIP6 & Inria
NMV 2017–2018

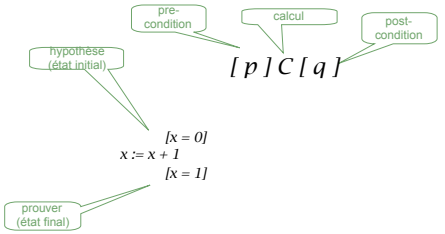
Raisonnement par invariants
Raisonnement séquentiel
Concurrence et cohérence
Rely-Guarantee
Liste chaînée concurrente

Logique de Hoare appliquée aux objets
Précondition, postcondition
Invariant de structure
Relation d'abstraction
Prouver ces relations
Eiffel, Spec#, ESC/Java
Vérification à l'exécution
assert

A Primer on Hoare Logic

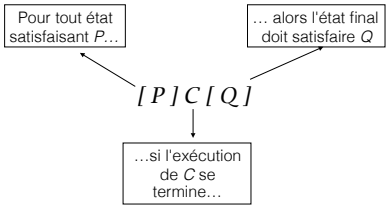
- A language of assertions describing property of the state at different program points
- Program variables can be used in assertions
- Invariants: Assertions that are true of all the states in a piece of code (e.g. loop invariant, global invariant, etc.)

Logique de Hoare



Pré-condition ; calcul ⇒ post-condition

Logique de Hoare



Composition de triplets pour prouver des programmes complexes

$$\{Pre\}\ c_0;\ \{P\}\ c;\ \{Q\}\ c_1;\ \{Post\}$$

A simple Hoare Logic Proof

$$\{\mathtt{x} = X \wedge \mathtt{y} = Y\}$$
$$\mathtt{aux} = \mathtt{x};$$
$$\boxed{\{\mathtt{x} = X \wedge \mathtt{y} = Y \ \wedge \mathtt{aux} = X\}}$$
$$\mathtt{x} = \mathtt{y};$$
$$\boxed{\{\mathtt{x} = Y \wedge \mathtt{y} = Y \ \wedge \mathtt{aux} = X\}}$$
$$\mathtt{y} = \mathtt{aux};$$
$$\{\mathtt{x} = Y \wedge \mathtt{y} = X\}$$

A simple Hoare Logic Proof

$\{x \neq 0\}$
$y := x * x$
$\{y = x^2 \wedge x \neq 0\}$
$y := y/x$
$\{y = x\}$

Check these examples

$\{x = 3\}\ x := x + 1\ \{x \leq 0\}$
$\{x = 3\}\ x := x + 1\ \{x \geq 0\}$
$\{x = 3\}\ x := x + 1; y := x\ \{y \geq 0\}$
$\{\exists X, x = X \wedge y = X + 1\}\ x := x + 1; y := x\ \{x = y\}$
$\{x = 0\}$ if $true$ then $x := x + 1$ else $x := x - 1$ fi $\{x = 1\}$
$\{x = 0\}$ if $x = 5$ then $x := x + 1$ else $x := x - 1$ fi $\{x = 1\}$
$\{x = 0\}$ while $true$ do $x := x + 1$ od $\{false\}$

Hoare Logic Rules

$\{P\}\ skip\ \{P\}$ no-op

$\{P[x \leftarrow e]\}\ x := e\ \{P\}$ assign

$$\frac{\{P\}\ c_0\ \{R\} \quad \{R\}\ c_1\ \{Q\}}{\{P\}\ c_0; c_1\ \{Q\}}$$ sequence

$$\frac{\{P \wedge b\}\ c_0\ \{Q_0\} \quad \{P \wedge \neg b\}\ c_1\ \{Q_1\}}{\{P\}\ \text{if } b \text{ then } c_0 \text{ else } c_1 \text{ fi } \{b \Rightarrow Q_0 \wedge \neg b \Rightarrow Q_1\}}$$ conditional

while $$\frac{\{P \wedge b\}\ c\ \{P\}}{\{P\}\ \text{while } b \text{ do } c \text{ od } \{\neg b \wedge P\}}$$

$$\frac{\{P\}\ c\ \{Q\} \quad P' \Rightarrow P \quad Q \Rightarrow Q'}{\{P'\}\ c\ \{Q'\}}$$ strengthening

Quelques succès

SeL4 : noyau
Astrée : code de navigation Airbus
RATP métro automatique
CompCert : compilateur C
Infer : cycle de développement Facebook
Failles de sécurité
NASA : Pathfinder
Millions de ligne de code séquentiel

SeL4 : conforme à sa spécification (la partie gestion de mémoire virtuelle non prouvée, tout le reste prouvé).
Airbus : C sans malloc / pas de division par zéro, de débordement de tableau, de buffer overflow.
CompCert : preuve que le code généré respecte la sémantique formelle du C

Attention, ça reste difficile (SeL4 : plusieurs thèses). Si on ne peut pas prouver c'est peut-être un bug, mais difficile de trouver la source.

Astrée fait la preuve d'absence d'erreur à l'exécution dans du code embarqué C (donc sans malloc) : dépassement de capacité en entier ou flottant lors des cacluls ou des conversions (overflow), opération arithmétique invalide en entier (division par zéro, shift, etc.) ou en flottant (division par zéro), erreurs d'arithmétique de pointeur ou de déréférencement invalide, dépassement de tableau, assert qui échouent, data-race et deadlock sur des programmes multi-thread (sans création dynamique de thread).

L'application principale est l'analyse de code avionique critique synchrone. Le programme est figé et correct ; le but de l'analyse est de prouver formelle l'absence d'erreur à l'exécution, en modifiant l'analyseur.

Il faut compter environ deux ans de développement de l'analyseur pour avoir des abstractions assez fines pour prouver que le code est correct (i.e., supprimer les fausses alarmes en améliorant l'analyseur) sur une gamme de logiciels (A340). Environ autant pour une autre gamme, plus complexe (A380). Peu de modifications nécessaires ensuite pour maintenir le zéro fausse alarme quand le code analysé évolue en gardant des caractéristiques similaires.

L'extension d'Astrée aux logiciels concurrents analyse des programmes de un ou deux million de lignes embarqués avioniques. Il y a encore des fausses alarmes (contrairement à l'analyse sur le code synchrone), donc pas une preuve formelle de sûreté. C'est acceptable car le code est moins critique. Par ailleurs, l'extension n'est pas utilisée en production chez Airbus. AbsInt, qui industrialise Astrée synchrone et concurrent a d'autres clients industriels avec de grosses applications, mais c'est difficile d'avoir des informations dessus.

Liste chaînée séquentielle

Spécification

Création : assure *invariant*

Méthode :

- suppose *invariant* vrai
- + *pré-condition* sur les arguments
- laisse *invariant* vrai
- + *post-condition* décrit effet de bord

Invariant de structure : structure interne de la donnée

Pas d'interférence :

- encapsulation
- ramasse-miettes

Spéc : facilite la compréhension du code
contrat clair entre les différentes paries du code

Objets encapsulés : il suffit de regarder le code de la classe

Si pas de RM attention aux effets de bord entre allocation et désallocation

Abstraction: ensemble d'entiers

Pre		Post
$v \in \mathbb{Z}$	$r \coloneqq E.add\ (v)$	$v \in E \wedge r = (v \notin old(E))$
$v \in \mathbb{Z}$	$r \coloneqq E.remove\ (v)$	$v \notin E \wedge r = (v \in old(E))$
$v \in \mathbb{Z}$	$r \coloneqq E.contains\ (v)$	$r = (v \in E)$

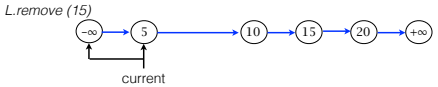
Précondition : vrai au début ; si mise en œuvre correcte ==> post-condition vraie à la fin

Abstraction: ensemble d'entiers

Pre	Abstraction	Post
$v \in int32^-$	$r \coloneqq E.add\ (v)$	$v \in E \wedge r = (v \notin old(E))$
$v \in int32^-$	$r \coloneqq E.remove\ (v)$	$v \notin E \wedge r = (v \in old(E))$
$v \in int32^-$	$r \coloneqq E.contains\ (v)$	$r = (v \in E)$

$head.value = -2^{31}$
$tail.value = 2^{31}-1$
$v \in \text{int32}^- =]\ -2^{31},\ 2^{31}-1[$

Réalisation : Liste chaînée d'entiers

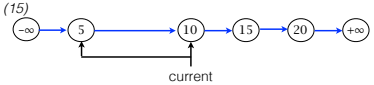


invariant de boucle
current.value < v

La même chose qu'avant, en réservant des valeurs faisables pour head et tail, donc v ne peut pas avoir ces valeurs
int32-- = entiers 32 bits moins les marqueurs de début et de fin

Réalisation : Liste chaînée d'entiers

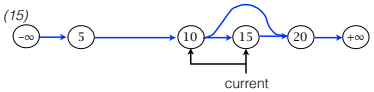
L.remove (15)



invariant de boucle
current.value < v

Réalisation : Liste chaînée d'entiers

L.remove (15)

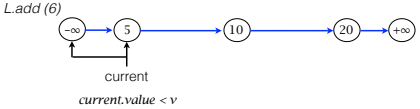


arrêt
¬ *current.value < v*

tail.value = +∞ ⇒ algorithme se termine

Réalisation : Liste chaînée d'entiers

L.add (6)

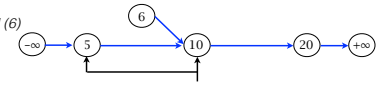


current.value < *v*

head, *tail* ⇒ l'insertion au début / à la fin n'est pas un cas spécial

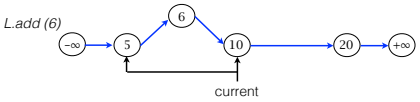
Réalisation : Liste chaînée d'entiers

L.add (6)



arrêt

Réalisation : Liste chaînée d'entiers

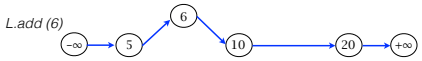


Invariant de structure :

- $head.value = -\infty \wedge tail.value = +\infty$
- $\forall n \neq tail$: $n.next$ nœud valide
  $\wedge\ n.value < n.next.value$

Invariant de structure : assertion sur l'état correct de l'objet, doit être maintenu par toute méthode
Vrai au début ==> vrai à la fin

Réalisation : Liste chaînée d'entiers



Invariant de structure :

- $head.value = -\infty \wedge tail.value = +\infty$
- $\forall n \neq tail$: $n.next$ nœud valide
  $\wedge\ n.value < n.next.value$

Pour simplifier la mise en œuvre je prends des valeurs représentables pour le premier et dernier.
Le système de type Java assure qu'on pointe vers un objet de type Node, donc il suffit de vérifier que le pointeur est non nul.

Relation d'abstraction

Spécification = abstraction, vue externe
- Ensemble *E*
- opérations : pre, post-condition

Réalisation = vue concrète, interne
- Liste *L*
- Invariant de structure

Relation d'abstraction
- La réalisation doit satisfaire la *spécification*
- $v \in]HEADVAL, TAILVAL[\Rightarrow$
  $v \in E \Leftrightarrow \exists n \in Node: L.head \rightsquigarrow^* n$
  $\wedge\ n.value = v$

Mise en œuvre séquentielle correcte

Spécification d'une classe :
- Précondition, postcondition
- Invariant de structure
- Relation d'abstraction
- À l'entrée et à la sortie de chaque méthode

Mise en œuvre correcte :
- Prouver ces relations
- Eiffel, Spec#, ESC/Java

Un pis-aller : vérification à l'exécution
- assert
- test

"Implements" n'est pas français

$m \rightsquigarrow^* n$ = existe chemin de *m* vers *n*

assert

Vérification de prédicat à l'exécution

- C, C++: assert (p);
- Java: assert p;
- Exclu en mode production

Avantages / inconvénients :

- ☺ Expression booléenne quelconque
- ☺ Test inclus avec le code
- ☺ Documentation
- ☹ Que du test
- ☹ Ne protège pas en production

À UTILISER !!!!

Voir listing SeqInt.java ci-joint

assert à l'entrée et à la sortie

```
class Truc {
  public boolean rep () {
    ... // tester invariant de structure
    }
  public bool meth1 (int x, String y) {
    assert rep();
    assert x <= y.size(); // arguments OK
    try {
      ... // mise en œuvre
    } finally {
      assert return_OK (...); // retour OK
      assert rep ();
    }
}
```


Java : finally assure que les assertions de sortie seront toujours exécutées, quelle que soit la façon dont on sort de la méthode (fin, return, ou exception).

Concurrence et cohérence

Concurrence ⟹ interférence

[x = 0]
x := x + 1
~~*[x = 1]*~~

x partagé entre
plusieurs fils d'exécution

Pré-condition ; calcul
+ *non-interférence*
⇒ post-condition

Ex.:"Aucun fil concurrent ne modifie x"

Correctness: fundamental issue with concurrency is *interference*

Non-interference:

- Owicki-Gries: no concurrent thread violates the precondition (in this case, that *x == 0)*
- Simplified: no conurrent thread writes *x*

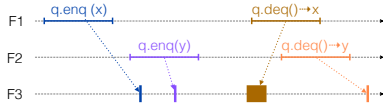
L'encapsulation ne suffit plus

Cohérence

Réalisation concurrente, comportement séquentiel
- Définir l'ordre dans lequel les opérations sont observées

Ex.: Cohérence *à terme* : lorsqu'il n'y a pas d'opérations en cours, tous les fils observent le même état

Cohérence séquentielle

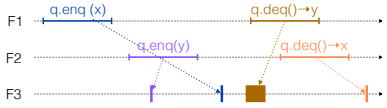


Effets observables des opérations d'un objet :
- dans un même ordre séquentiel
- respectant la spécification séquentielle
- et respectant leur ordre d'apparition dans le programme *(program order)*

p.deq()→x; p.enq(x)
interdit

Non compositionnel ! Si on observe deux objets à cohérence séquentielle, le résultat n'est pas forcément à cohérence séquentiel

Cohérence séquentielle

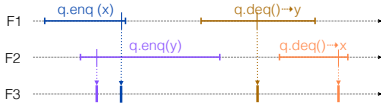


Effets observables des opérations d'un objet :

- dans un même ordre séquentiel
- respectant la spécification de l'objet
- et respectant leur ordre d'apparition dans le programme *(program order)*

Même ordre : ordre total
p, q: objets "file d'attente" (queue)

Linéarisabilité



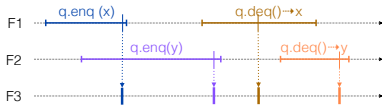
Effets observables des opérations d'un objet :

- dans un même ordre séquentiel
- respectant la spécification de l'objet
- ordre total *(≈instantanément)*
- entre l'appel et le retour

Exhiber le point de linéarisation

Si deux opérations se recouvrent, leur ordre d'observation est non-déterministe : je peux choisir l'ordre qui m'arrange !
Nous en profiterons dans l'algorithme de pile (Elimination Stack)

Linéarisability

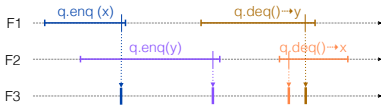


Effets observables des opérations d'un objet :

- dans un même ordre séquentiel
- respectant la spécification de l'objet
- ordre total *(≈instantanément)*
- entre l'appel et le retour

Exhiber le point de linéarisation

Linéarisability



Effets observables des opérations d'un objet :

- dans un même ordre séquentiel
- respectant la spécification de l'objet
- instantanément
- entre l'appel et le retour

Point de linéarisation

Si deux opérations se recouvrent, leur ordre d'observation est non-déterministe : je peux choisir l'ordre qui m'arrange !
Nous en profiterons dans l'algorithme de pile (Elimination Stack)

Objets corrects

- *Sûreté :*
  - Conforme spécification séquentielle
    - Pré-condition, post-condition
    - Invariant de structure
    - Relation d'abstraction
  - Exhiber point de linéarisation
    - abstraction conforme spécif séquentielle

  ⇒ Non-interférence entre fils (Rely-Guarantee)

- *Vivacité :*

	Bloquant	*Non bloquant*
∃ *appel termine*	sans étreinte mortelle	sans verrou
∀ *appels terminent*	sans famine	sans attente

Bloquant ⇒ Amdahl

Programmes concurrents

De Hoare à Rely-Guarantee

Hoare and concurrency

$$\frac{\{P\}\ c_0\ \{R\} \qquad \{R\}\ c_1\ \{Q\}}{\{P\}\ c_0; c_1\ \{Q\}}\ \text{sequence}$$

- Sequence rule: proving a sequential program reduces to proving each individual step
- Not true for concurrent programs! Interference at ";"
- Between any atomic steps

Et les pg concurrents ?

$$\frac{\{P\}\ c_0\ \{R\} \qquad \{R\}\ c_1\ \{Q\}}{\{P\}\ c_0; c_1\ \{Q\}}\ \text{sequence}$$

La preuve d'un programme séquentiel se réduit à la preuve de chaque pas individuellement

Ce n'est plus vrai pour les programmes concurrents :

Interférence possible entre chaque deux pas atomiques

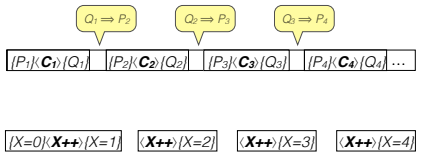
Sûreté et concurrence

Object correct / sûreté :

- Assertions séquentielles
  - Pré-condition, post-condition
  - Invariant de structure
  - Relation d'abstraction
  - Preuve par *rely-guarantee*
- Point de linéarisation

La mise en œuvre concurrente est équivalente à la spécification séquentielle

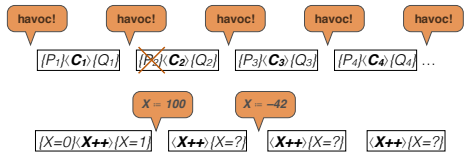
Modèle séquentiel



La preuve d'un programme séquentiel se réduit à la preuve de chaque étape

Programme = suite d'actions atomiques $\langle C_i \rangle$

Modèle concurrent



Impossible de raisonner si on ne sait pas ce que fait l'environnement

Rely-Guarantee

Logique pour raisonner sur les programmes concurrents

≈ Logique de Hoare + processus + *non-interférence*

Rely = interférence que mon processus accepte de la part des autres

Guarantee = ce que mon processus promet aux autres

Permet de raisonner de façon séquentielle sur chaque processus isolément

Plus complexe que Hoare

L'environnement est susceptible de faire n'importe quel effet de bord entre deux pas atomiques
havoc = dévastation, chamboulement

Rely-Guarantee

Pour chaque processus:

$[P_i, R_i]\ \mathbf{C_i}\ [G_i, Q_i]$

au moment de l'invocation

≈Hoare + parallélisme : non-interférence

- Hypothèses (processus *i*) :
  - Précondition : P_i
  - *Rely* : à chaque pas d'exécution : R_i

à chaque pas atomique

- En déduire (processus *i*) :
  - *Guarantee* : à chq pas : G_i
  - État final : Q_i

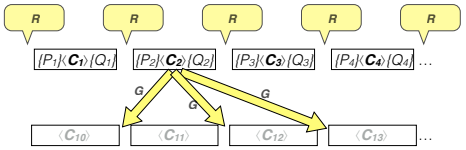
au retour

Par processus : preuve séquentielle ≈classique

Global : $\forall j \neq i : guar_j \Rightarrow rely_i$

« Si tu ne tires pas, je ne tirerai pas »

Modèle R–G



Rely = interférence que mon processus accepte de la part des autres

Guarantee = ce que mon processus promet aux autres

Si pré-condition *stable* par rapport au *Rely*, on peut raisonner

Programme = suite d'actions atomiques ⟨C_i⟩

Exemples

Séquentiel

{X=0} **t := X; t := t+1; X := t** *{X=1}*

Concurrent

[X=0, ID(X)] **t := X; t := t+1; X := t** *[X=1, MOD(X)]*

||

[Y=0, ID(Y)] **t := Y; t := t+2; Y := t** *[Y=2, MOD(Y)]*

Notation

x, t : locales au processus
X, T : partagés entre processus
ID(X) : une action ne modifie pas *X* : *X=old(X)*
MOD(X) : une action modifie au plus *X*
MOD(X) ≙ ∀ *T≠X, ID(T)*
Preserve(P) ≙ *old(P)* ⟹ *P*

Chaque précondition (y compris les intermédiaires) du code concurrent sont stables/non modifiées par rapport à son Rely
Chaque rely est garanti par le Guarantee de l'autre processus
Cet exemple montre que les données sont disjointes

$i \neq j \Longrightarrow (RS_i \cup WS_i) \cap (RS_j \cup WS_j) = \varnothing$

alors on peut raisonner sur chaque processus indépendamment (ouf !)

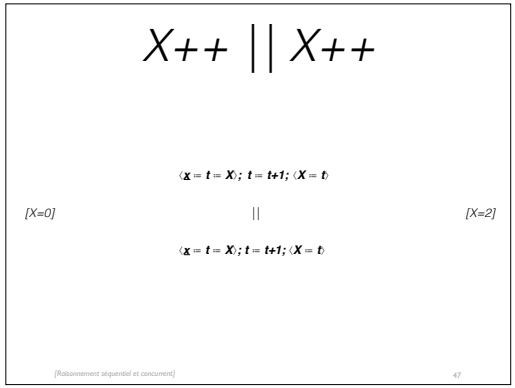
X++ || X++

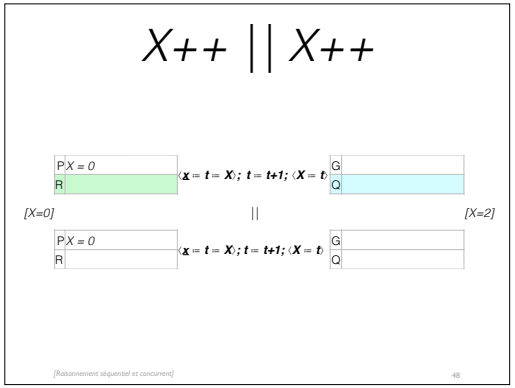
X++

[X=0] || *[X=2]*

X++

Deux processus concurrents incrémentent de 1 la variable partagée *X,* initialement 0. On s'attend à une valeur finale de 2. Vérifions.

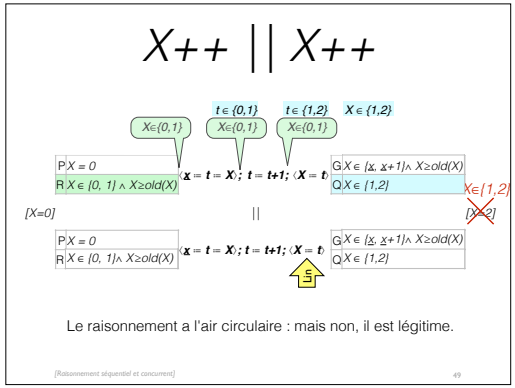


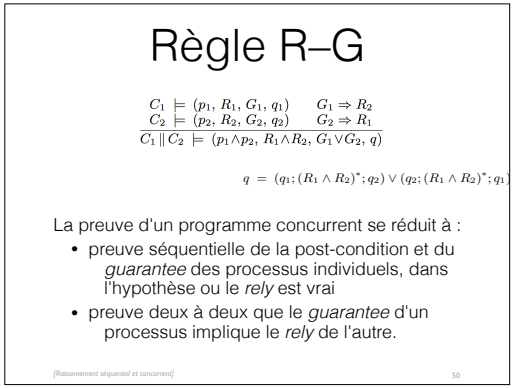


Voici les pas élémentaires des deux processus. Quel *rely* ? peut-on établir le *guarantee* et la post-condition ?

Supposons que le 1er processus puisse s'appuyer sur X∈{0,1} (le 2è soit n'a rien fait, soit a terminé avant moi).

Je reporte le rely entre chaque pas. Je peux en déduire (post-condition) la valeur de t à chaque étape. Je garantis de ne pas modifier X. Dans ce cas, ma valeur finale est $X \in \{1,2\}$ et mon *Guar* $X \in \{0,1\}$ (plus précisément : $X \in \{\underline{x}, \underline{x}+1\} \wedge X \geq old(X)$ où $\underline{x}$ = valeur lue au début).
Idem de l'autre côté.
Le raisonnement a l'air circulaire mais non, il est légitime! Prouvé. “Si tu ne tires pas, je ne tire pas.” [Abadi & Lamport TOPLAS 1995]
Mais attention la valeur finale est 1 ou 2, et non forcément 2! Car X++ n'est pas atomique.





Supposons que le 1er processus puisse s'appuyer sur X∈{0,1} (le 2è soit n'a rien fait, soit a terminé avant moi).

Je reporte le rely entre chaque pas. Je peux en déduire (post-condition) la valeur de t à chaque étape. Je garantis de ne pas modifier X. Dans ce cas, ma valeur finale est $X \in \{1,2\}$ et mon *Guar* $X \in \{0,1\}$ (plus précisément : $X \in \{\underline{x}, \underline{x}+1\} \wedge X \geq old(X)$ où $\underline{x}$ = valeur lue au début).
Idem de l'autre côté.
Le raisonnement a l'air circulaire mais non, il est légitime! Prouvé. “Si tu ne tires pas, je ne tire pas.” [Abadi & Lamport TOPLAS 1995]
Mais attention la valeur finale est 1 ou 2, et non forcément 2! Car X++ n'est pas atomique.

Intuitivement
- Post-condition: comme C1 suivi de C2 (ou l'inverse) avec interférence entre les deux
- Guarantee : disjonction. Par exemple : MOD(X)∨MOD(y) = MOD(X,Y)

TAS, getAndSet

```
TAS (L, v) { // hardware
    atomic {
        int tmp = L;
        L = V;
        return tmp;
    }
}
```

P: true *R: true*	***r* := *TAS(L, v)***	*G: MOD(L)* *Q: r=old(L) ∧ L = v*

Lin = point de linéarisation

Exclusion mutuelle

P: L.owner = null *R: true*	***lock(L)***	*G: MOD(L)* *Q: L.owner = thistd*
P: L.owner = thistd *R: true*	***unlock(L)***	*G: MOD(L)* *Q: L.owner = null*

Variable partagée *X* protégée par verrou *L*. Plusieurs processus ‖ :

P: L.owner = null	***…; lock(L); …; X := thisthr; …; unlock(L)***	*G: L.owner≠thistd ⇒ ID(X)*
R: L.owner = thistd ⇒ ID(X)		*Q: X = thistd*

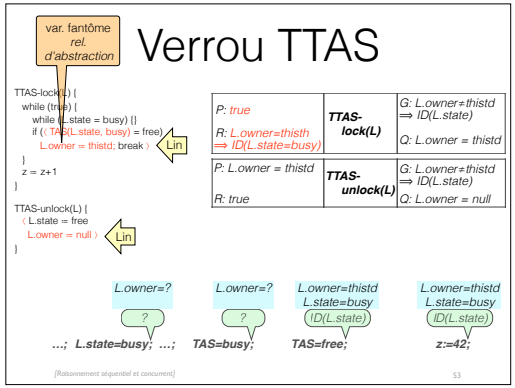
L.owner=thistd *ID(X)* ***…; lock(L);***

L.owner=thistd *X=thistd* *ID(X)* ***…; X := thisthr;***

L.owner=thistd *X=thistd* *ID(X)* ***…;***

L.owner=null *X=thistd* *ID(X)* ***unlock(L)***

thistd = ID of this thread
La discipline de verrouillage a été explicitée
Grâce à celle-ci il n'y a pas eu de mauvaise interférence : X a la bonne valeur à la fin.
Il y a bien eu de la « bonne » interférence entre lock/unlock.



z≔z+1 ne sert à rien ici, c'est juste pour montrer le point de linéarisation.
La spéc du verrou mentionne L.owner; or il n'est pas dans le code !
Si on ajoute une affectation à L.owner, elle ne sera pas atomique avec le TAS... Comment s'en sortir ?
Réponse : L.owner est une variable fantôme : elle n'existe que pour le raisonnement.
Puisqu'elle n'existe pas en machine, elle ne prend aucun temps, on peut donc décider qu'elle est atomique avec l'instruction qui précède (ou celle qui suit) !!!
[Nota : dans ce cas précis l'atomicité n'est pas vraiment nécessaire ; voyez-vous pourquoi ?]