Cours de Model Checking

Leçon 4 : Software Model Checking

Sébastien Bardin

CEA-LIST, Laboratoire de Sûreté Logicielle

sebastien.bardin@cea.fr
http://sebastien.bardin.free.fr/

Plan

Rappels

Model Checking

Technique de vérification automatique de systèmes réactifs

Ingrédients

- $\mathcal{M}$ = système de transitions
- φ = formule de logique temporelle (LTL, CTL, CTL*, etc.)
- MC = est-ce que $\mathcal{M} \models \varphi$?

Algorithmes déjà vus

- algos standards (LTL : automates – CTL : marquage)
- algos optimisés (LTL : on-the-fly, etc. – CTL : symbolique)

Comment attaquer des systèmes infinis, voir du logiciel ?

Rappel : Model Checking Symbolique

Ingrédients

- domaine de valeurs symboliques D représentant des *ensembles* de configuration
  - $[\![]\!] : D \mapsto 2^Q$
- calculs symboliques sur D copiant les calculs ensemblistes sur 2^Q
  - $\bot$ et $\top$ tels que $[\![\bot]\!] = \emptyset$ et $[\![\top]\!] = 2^Q$
  - $[\![d \sqcup d']\!] = [\![d]\!] \cup [\![d']\!]$
  - $[\![d \sqcap d']\!] = [\![d]\!] \cap [\![d']\!]$
  - $[\![\neg d]\!] = 2^Q \setminus [\![d]\!]$
  - $d \sqsubseteq d'$ ssi $[\![d]\!] \subseteq [\![d']\!]$
  - $d = \bot$ ssi $[\![d]\!] = \emptyset$
- calcul de pre et post symboliques (ici : toutes les transitions d'un coup)
  - $[\![post^\#(d)]\!] = post([\![d]\!])$

Avec ça : on refait le calcul d'accessibilité, version ensembliste

[On peut en fait faire tout CTL*]

Rappel : Model Checking Symbolique (2)

Calcul symbolique d'accessibilité (d_I tq $[\![d_I]\!]$ = états initiaux)

$d := d_I$
Tant que $post^{\#}(d) \not\sqsubseteq d$ **faire**
  $d := post^{\#}(d) \sqcup d$
Fin Tant que
retourner d

Remarque : ici juste besoin de $\sqcup, post^{\#}, \sqsubseteq$
. CTL et CTL* ont besoin des autres opérateurs

Rappels : sur le model checking symbolique

Le model checking symbolique ouvre de nombreuses possibilités !!

Les ensembles peuvent être infinis $\{(x, y, z).x + y \geq z\}$

- model checking de systèmes infinis
- quelques beaux résultats (pushdown systems, automates temporisés, réseaux de Petri)

Les ensembles peuvent être approchés $x \in [0..100]$ pour coder $\{0, 65, 100\}$

- liens avec l'interprétation abstraite
- un exemple utile : predicate abstraction

Le software model checking se base extensivement sur ces idées

Quelques points clés

- Challenge : problème indécidable !!
  - . idée 1 : maîtriser les abstractions
    - yes/? — ?/no — yes/no/loop
  - . idée 2 : identifier des sous-classes utiles
- Méthode : se ramener fortement à la logique
  - . cadre général pour des domaines symboliques
  - . progrès dans les solveurs : classes expressives et efficaces (en pratique)

Things like even software verification, this has been the Holy Grail of computer science for many decades but now in some very key areas, for example, driver verification we're building tools that can do actual proof about the software and how it works in order to guarantee the reliability.

- Bill Gates (2002)

Plan

Sémantique d'un programme

Cas simple : juste des variables entières a, b, c, d, ...

Sémantique opérationnelle

- une configuration du système :
  . c : Var $\mapsto \mathbb{N}$
- une affectation modifie l'état mémoire :
  . ex : z := a+b
  . c $\xrightarrow{z:=a+b}$ c' ssi c' = c[z $\leftarrow$ c[a] + c[b]]
- un branchement teste l'état mémoire

Sémantique relationnelle

- sémantique des instructions = des relations sur $C \times C$
- lien entre variables avant et après l'instruction
- ok aussi pour des langages plus compliqués (mémoire, pointeurs, allocation dynamique, etc.)

Plan

Calcul symbolique via formules

Domaine symbolique basé sur des formules logiques $(d = \varphi)$

. ex : $x \geq a + b \wedge y == load(Memory, z) + 1 \wedge ...$

- $[\![d]\!]$ = ensemble des configurations c tq $c \models d$
- union $\sqcup$: $\vee$
- empty(d) : unsat(d)
- $d \sqsubseteq d'$: valid$(d \Rightarrow d')$
- calcul de $post^{\#}$? De manière générale : on donne à chaque instruction t une relation logique $\psi_t(in, out)$
  - . $post_t^{\#}(\varphi(x)) \triangleq \exists x.\varphi(x) \wedge \psi_t(x, x')$
  - . on peut souvent faire des constructions plus simples

Calcul de $post^{\#}$ (1)

Loc	Instruction
0	input(y,z)
1	w := y+1
2	x := w + 3
3	if (x < 2 * z) (branche True)
4	if (x < z) (branche False)

Prédicat du chemin (entrées Y_0 et Z_0)

$\top$

Calcul de $post^{\#}$ (1)

Loc	Instruction
0	input(y,z)
1	w := y+1
2	x := w + 3
3	if (x < 2 * z) (branche True)
4	if (x < z) (branche False)

Prédicat du chemin (entrées Y_0 et Z_0)

$\top \wedge W_1 = Y_0 + 1$

Calcul de $post^{\#}$ (1)

Loc	Instruction
0	input(y,z)
1	w := y+1
2	x := w + 3
3	if (x < 2 * z) (branche True)
4	if (x < z) (branche False)

Prédicat du chemin (entrées Y_0 et Z_0)

$\top \wedge W_1 = Y_0 + 1 \wedge X_2 = W_1 + 3$

Calcul de $post^{\#}$ (1)

Loc	Instruction
0	input(y,z)
1	w := y+1
2	x := w + 3
3	if (x < 2 * z) (branche True)
4	if (x < z) (branche False)

Prédicat du chemin (entrées Y_0 et Z_0)

$$\top \wedge W_1 = Y_0 + 1 \wedge X_2 = W_1 + 3 \wedge X_2 < 2 \times Z_0$$

Calcul de $post^{\#}$ (1)

Loc	Instruction
0	input(y,z)
1	w := y+1
2	x := w + 3
3	if (x < 2 * z) (branche True)
4	if (x < z) (branche False)

Prédicat du chemin (entrées Y_0 et Z_0)

$$\top \wedge W_1 = Y_0 + 1 \wedge X_2 = W_1 + 3 \wedge X_2 < 2 \times Z_0 \wedge X_2 \geq Z_0$$

Calcul de $post^{\#}$ (1)

Loc	Instruction
0	input(y,z)
1	w := y+1
2	x := w + 3
3	if (x < 2 * z) (branche True)
4	if (x < z) (branche False)

Prédicat du chemin (entrées Y_0 et Z_0)

$$\top \wedge W_1 = Y_0 + 1 \wedge X_2 = W_1 + 3 \wedge X_2 < 2 \times Z_0 \wedge X_2 \geq Z_0$$

Alternative :

$$\text{let } W_1 \triangleq Y_0 + 1 \text{ in}$$
$$\quad \text{let } X_2 \triangleq W_1 + 3 \text{ in}$$
$$\quad\quad X_2 < 2 \times Z_0 \wedge X_2 \geq Z_0$$

Calcul de $post^{\#}$ (2)

attention : introduire une nouvelle variable logique à chaque nouvelle utilisation d'une variable du programme

- les "variables" du programme C peuvent être modifiées à chaque étape de l'exécution
- les "variables" logiques sont des inconnues, de valeur constante
- le renommage est nécessaire pour prendre en compte la dynamique de l'exécution
  - prédicat pour `x := x+1` ?
  - $X_{n+1} = X_n + 1$, plutôt que $X = X + 1$

Réfuter des invariants : k-BMC

Bounded model checking

- principe : utiliser l'approche précédente pour calculer une représentation exacte de $post^{\leq k}(\text{Init})$
  - . approche naive : énumérer les chemins et $\vee$
- propriétés
  - . permet de réfuter un invariant I quand $post^{\#\leq k}(\text{Init}) \wedge \neg\varphi_I$ est satisfiable
  - . par contre, ne peut pas prouver I ...
  - . remarque : la procédure est correcte et complète si les chemins du programme sont bornés (taille l), et $k \geq l$
- Efficacité
  - . attention à la taille de la formule !! [#chemins exponentiel]
  - . construction via forme ssa (partage), preprocessing, etc.
- variante : exécution symbolique
  - . résoud un seul chemin à la fois : formules simples
  - . MAIS problème de l'énumération de chemin

Alternative au BMC : exécution symbolique

input: a program P

output: a test suite TS covering all feasible paths of $Paths^{\leq k}(\mathtt{P})$

- pick a path $\sigma \in Paths^{\leq k}(\mathtt{P})$
- compute a *path predicate* φ_σ of σ [wpre, spost]
- solve φ_σ for satisfiability [smt solver]
- SAT(s)? get a new pair $< \mathrm{s}, \sigma >$
- loop until no more path to cover

Prouver des invariants : k-induction

k-Induction : prouver un invariant I avec un raisonnement borné

- Principe de base : on cherche à montrer que I est un invariant inductif
  - . invariant inductif : $\text{Init} \subseteq I$ et $post(I) \subseteq I$
  - . propriété : un invariant inductif est bien un invariant

  [invariant : $post^*(\text{Init}) \subseteq I$]
- Généralisation : on essaie de montrer que I est k-inductif
  - . $\text{Init}, post(\text{Init}), ..., post^{k-1}(\text{Init}) \subseteq I$,
  - . et : $post^k(I) \subseteq I$

facile à vérifier symboliquement ! [mais invariant initial rarement k-inductif]

Remarque : programmes à chemins bornés

- k-BMC est correct et complet pour k suffisant
  . car dans ce cas $post^{\leq k}(Init) = post^{*}(Init)$
- on peut faire un BMC itératif, en augmentant k
- on peut aussi entrelacer k-BMC et k-induction, pour essayer de terminer plus rapidement

À retenir : la vérification formelle est simple pour des systèmes infinis dont les chemins sont bornés et peu nombreux

Remarque : modélisation mémoire

Considérons l'instruction : `x := a + b`

Traduction 1 :

$$X_{n+1} = A_n + B_n$$

Remarque : modélisation mémoire

Considérons l'instruction : `x := a + b`

Traduction 1 :

$X_{n+1} = A_n + B_n$

Modèle mémoire sous-jacent : ensemble de variables $\{A, B, X, \ldots\}$

Remarque : modélisation mémoire

Considérons l'instruction : `x := a + b`

Traduction 1 :

$X_{n+1} = A_n + B_n$

Modèle mémoire sous-jacent : ensemble de variables $\{A, B, X, \ldots\}$

Bien mais ne pourra prendre en compte les pointeurs

Remarque : modélisation mémoire

Considérons l'instruction : `x := a + b`

Traduction 2 : ajout d'un état mémoire M

$$store(M, addr(X), load(M, addr(A)) + load(M, addr(B)))$$

Remarque : modélisation mémoire

Considérons l'instruction : `x := a + b`

Traduction 2 : ajout d'un état mémoire M

$store(M, addr(X), load(M, addr(A)) + load(M, addr(B)))$

Modèle mémoire sous-jacent : map $\{Addr_1 \mapsto A, Addr_2 \mapsto B, \ldots\}$

Remarque : modélisation mémoire

Considérons l'instruction : `x := a + b`

Traduction 2 : ajout d'un état mémoire M

$store(M, addr(X), load(M, addr(A)) + load(M, addr(B)))$

Modèle mémoire sous-jacent : map $\{Addr_1 \mapsto A, Addr_2 \mapsto B, \ldots\}$

ok pour les pointeurs, mais on ne peut écrire au milieu de `x` (respect du typage, modèle mémoire à la Java)

Remarque : modélisation mémoire

Considérons l'instruction : `x := a + b`

Remarque : modélisation mémoire

Considérons l'instruction : `x := a + b`

Traduction 3 : encodage de M au niveau octet (ici : 3 octets)

let tmpA = load(M,addr(A)) @ load(M,addr(A)+1) @ load(M,addr(A)+2)
and tmpB = load(M,addr(B)) @ load(M,addr(B)+1) @ load(M,addr(B)+2)
in
let nX = tmpA+tmpB
in

$$
\begin{array}{l}
\quad store(\\
\qquad store(\\
\qquad\quad store(M, addr(X), nX[0]), \\
\qquad\quad addr(X) + 1, nX[1]), \\
\qquad addr(X) + 2, nX[2])
\end{array}
$$

Remarque : modélisation mémoire

Considérons l'instruction : `x := a + b`

Traduction 3 : encodage de M au niveau octet (ici : 3 octets)

let tmpA = load(M,addr(A)) @ load(M,addr(A)+1) @ load(M,addr(A)+2)
and tmpB = load(M,addr(B)) @ load(M,addr(B)+1) @ load(M,addr(B)+2)
in
let nX = tmpA+tmpB
in

$$store(store(store(M, addr(X), nX[0]), addr(X)+1, nX[1]), addr(X)+2, nX[2])$$

ok pour du C ... mais la formule est complexe

Remarque : modélisation mémoire

Considérons l'instruction : `x := a + b`

PB ouvert : affiner automatiquement le niveau d'abstraction de la modélisation

Plan

Model Checking Abstrait

Ingrédients

- domaine de valeurs symboliques D représentant des *ensembles* de configuration
  - $[\![]\!] : D \mapsto 2^Q$
- calculs symboliques sur D copiant les calculs ensemblistes sur 2^Q, en permettant une surapproximation
  - $\bot$ et $\top$ tels que $[\![\bot]\!] = \emptyset$ et $[\![\top]\!] = 2^Q$
  - $[\![d \sqcup d']\!] \supseteq [\![d]\!] \cup [\![d']\!]$
  - si $d \sqsubseteq d'$ alors $[\![d]\!] \subseteq [\![d']\!]$
  - si $\text{empty}(d)$ alors $[\![d]\!] = \emptyset$
- calcul de post symbolique
  - $[\![post^\#(d)]\!] \supseteq post([\![d]\!])$

Avec ça : on refait le calcul d'accessibilité, version ensembliste

- pb1 : surapprox, ne peut conclure que yes/?
- pb2 : terminaison a priori pas garantie si chemins non bornés [mais possible]

Exemple classique de domaine

d = ensembles finis de formules conjonctives $\{\varphi_1, \ldots, \varphi_n\}$ dans une théorie fixée T

- $[\![d]\!] = \bigcup_{\varphi \in d} [\![\varphi]\!]$
- $d_1 \sqcup d_2$ = union ensembliste de d_1 et d_2
- empty(d) ssi $\forall \varphi \in d, \text{unsat}(d)$
- post approché : $\text{post}^{\#}(d) = \bigcup_{\varphi \in d} \alpha(\text{post}^{\#}(\varphi))$

  . α prend une formule logique ψ quelconque et retourne une formule logique $\psi' \in T$ tq $\psi \Rightarrow \psi'$. Idéalement, ψ' "proche" de ψ

  . remarque : minimiser en enlevant les φ' unsat
- test $\sqsubseteq$ approché :

  $\{\varphi_1, \ldots, \varphi_n\} \sqsubseteq \{\varphi'_1, \ldots, \varphi'_n\}$

  ssi $\forall i, \exists j$ tq $\varphi_i \Rightarrow \varphi_j$

[vs BMC : formules moins grosses (important si coût résolution > PTIME)]

Plan

Abstraction de prédicats

Problème du MC abstrait : quel domaine ?

Proposition : Predicate Abstraction

- paramètre : ensemble fini de prédicats $p_1, ..., p_n$ sur les configurations du programme
  - . ex : $(x \geq y + 1), a == *p, ...$
- domaine : (un ensemble de) valuations des p_i : $p_i \mapsto \mathcal{B}$
  - . ensemble fini de valeurs abstraites !
  - . on note l_i pour p_i ou $\neg p_i$
- sémantique : $[\![(l_1, ..., l_n)]\!] = \{c | c \models \wedge_i l_i\}$

On réutilise le cadre précédent

- $\sqcup$ [purement ensembliste], empty? [besoin de solveur]

Abstraction de prédicats (2)

- $\sqsubseteq$ de manière +/- approchée :
  - . cf slide 21 [besoin de solveur]
  - . $\subseteq$ au niveau des ensembles de valuations [no solveur]

- calcul de successeurs : $post^{\#}$: valuation $\mapsto 2^{\text{valuation}}$
  - . $post^{\#}(v) = \{v' | sat(v \wedge v')\} = \{v' | [\![v]\!] \cap [\![v']\!] \neq \emptyset\}$

Exemples


```
1:  input x,z ≥ 0
2:  y:=x+z
3:  z:=z+5
4:  x:=z+5
5:  assert(x ≥ 10)
```

Exemples

Prédicats : $x \geq 10$ [* : préd. vrai ou faux]


```
1:  input x,z ≥ 0          (*)
2:  y:=x+z                 (*)
3:  z:=z+5                 (*)
4:  x:=z+5                 (*)
5:  assert(x ≥ 10)         (*) BAD
```

Exemples

Prédicats : $x \geq 0$, $x \geq 10$ [* : préd. vrai ou faux]


```
1:  input x,z ≥ 0          (1,*)
2:  y:=x+z                 (1,*)
3:  z:=z+5                 (1,*)
4:  x:=z+5                 (*,*)
5:  assert(x ≥ 10)         (*,*) BAD
```

Exemples

Prédicats : $x \geq 0$, $z \geq 0$, $z \geq 5$, $x \geq 10$ [* : préd. vrai ou faux]

1:	`input x,z` ≥ 0	(1,1,*,*)
2:	`y:=x+z`	(1,1,*,*)
3:	`z:=z+5`	(1,1,1,*)
4:	`x:=z+5`	(1,1,1,1)
5:	`assert(x` ≥ 10`)`	(1,1,1,1) OK

Abstraction de prédicats (3)

Domaine très sympathique !

- se ramène au cas fini, donc terminaison assurée !
- mieux : on peut être très efficace
  - . se ramener complètement à booléens, sans solveur FOL
  - . et réutiliser la machinerie des BDDs
  - . à condition d'avoir précalculé la relation $post^{\#}$
    (solveurs FOL utilisés seulement pendant phase de précalcul)

Problèmes

- attention à la phase de pré-calcul !
- trouver les prédicats !

À propos de $post^{\#}$

Calcul efficace des successeurs possibles

- calcul plus lazy
  - . on ne calcule $post^{\#}(v)$ que si nécessaire + mise en cache
  - $\rightarrow$ aide bcp si peu de valuations atteintes
  - . mais coût de $post^{\#}(v)$ dépend tjs de #valuations
  - . une idée : précalculer les $l_i \Rightarrow l_j$ (quadratique) pour borner facilement les v' possibles pour $post^{\#}(v)$
  - . note : dépend encore de #valuations, mais dans le pire cas
- valuations plus générales : $p_i \mapsto \{1, 0, *\}$
  - . plus concis : $\{(0,1,1),(1,1,1)\}$ devient $\{(*,1,1)\}$
- approximer plus (abstraction cartésienne)
  - . remplacer $\{(0,1,0),(1,1,1)\}$ par $\{(*,1,*)\}$
  - . précalcul de $post^{\#}(v)$ dépend de #prédicats !!
    [revient à précalculer les $l_i \Rightarrow l_j$]

Trouver les prédicats

Counter Example Guided Refinement

- on lance un calcul de MC abstrait avec predicats $p_1, ..., p_n$
- supposons que le calcul de $post^{\#}$ permette de trouver une trace menant à une erreur
- attention : trace abstraite, au niveau booléen
- exécution symbolique sur cette trace pour confirmer l'erreur
- vrai erreur : algorithme retourne KO
- fausse alarme : déduire un (ou des) nouveau prédicat p_{n+1} qui assurera que le même cex ne sera pas retrouvé

  . ex : wp le long du chemin, pour localiser la perte de précision, et construire le(s) prédicat(s)
- recommencer une nouvelle predicate abstraction avec $p_1, ..., p_n, p_{n+1}$

Exemple (2)

Raffinement : [* : préd. vrai ou faux]

. prédicats : $x \geq 10$


```
1:  input x,z ≥ 0          (*)
2:  y:=x+z                 (*)
3:  z:=z+5                 (*)
4:  x:=z+5                 (*)
5:  assert(x ≥ 10)         (*) BAD
```

Exemple (2)

Raffinement : [* : préd. vrai ou faux]

. prédicats : $x \geq 10$

Maintenant : weakest precondition à partir de 15

1:	`input x,z` ≥ 0	(*)	[$z \geq 0$]
2:	`y:=x+z`	(*)	[$z \geq 0$]
3:	`z:=z+5`	(*)	[$z \geq 5$]
4:	`x:=z+5`	(*)	[$x \geq 10$]
5:	`assert(x` ≥ 10`)`	(*) BAD	

Exemple (2)

Raffinement : [* : préd. vrai ou faux]

. prédicats : $x \geq 10$

Maintenant : weakest precondition à partir de 15

On recommence avec les prédicats trouvés : $z \geq 0$, $z \geq 5$, $x \geq 10$

1:	input x,z $\geq$ 0	(1,*,*)
2:	y:=x+z	(1,*,*)
3:	z:=z+5	(1,1,*)
4:	x:=z+5	(1,1,1)
5:	assert(x $\geq$ 10)	(1,1,1) OK

Aller plus loin

- raffiner juste où il faut : lazy cegar
- diminuer le nombre de points d'abstraction/raffinement : large-block encoding
- meilleur choix de prédicat : interpolants
- intégration fine raffinement / calcul abstrait (pas de pré-calcul) : property-directed reachability

Exemples (3)

Prédicats locaux : [* : préd. vrai ou faux]

- l1 : $x \geq 0$, $z \geq 0$
- l2 : $x \geq 0$, $z \geq 0$
- l3 : $z \geq 5$
- l4 : $z \geq 5$, $x \geq 10$
- l5 : $x \geq 10$

1:	input x,z ≥ 0	(1,1)
2:	y:=x+z	(1,1)
3:	z:=z+5	(1)
4:	x:=z+5	(1,1)
5:	assert(x ≥ 10)	(1) OK

Exemples (3)

Prédicats locaux : [* : préd. vrai ou faux]

- l1 : $x \geq 0$, $z \geq 0$
- l2 : $x \geq 0$, $z \geq 0$
- l3 : $z \geq 5$
- l4 : $z \geq 5$, $x \geq 10$
- l5 : $x \geq 10$

1:	input x,z ≥ 0	(1,1)
2:	y:=x+z	(1,1)
3:	z:=z+5	(1)
4:	x:=z+5	(1,1)
5:	assert(x ≥ 10)	(1) OK

Se combine bien avec le raffinement de domaine !

Exemples (4)

Large-block encoding : [* : préd. vrai ou faux]

. `l2,l3,l4` considérés comme un seul block

. prédicats : `z`$\geq$ `0, x`$\geq$ `10`


```
1:   input x,z ≥ 0          (1,*)
2-4:   y:=x+z
       z:=z+5
       x:=z+5               (1,1)
5:   assert(x ≥ 10)         (1,1) OK
```

Plan

Bilan & comparaison

approach	yes	no	termination
BMC/DSE	✗	✓	✓
k-induction	✓	✗	✓
abstract model checking	✓	✗	✗
predicate abstraction	✓	✗	✓
cegar	✓	✓	✗

Quelques succès industriels

- Static Driver Verifier (2002, Microsoft) : vérification de drivers (internes, tiers)
  . cegar + interprétation abstraite + mc infini (pushdown systems)
- SAGE (2008, Microsoft) : "smart fuzzing" intensif de composants critiques
  . exécution dynamique symbolique
- Prover Technology (outils pour l'industrie ferroviaire)
  . BMC et k-induction
- TLA+ (??, Microsoft, utilisé par Amazon) : vérification de protocoles de Cloud
  . LTL + FOL, débogage par model checking concret, preuve par système interactif
- Cubicle (Uni. Orsay, cas d'étude Intel) : vérification de protocoles de cohérence de cache
  . cf suite du cours de Sylvain Conchon

Liens avec d'autres approches de vérification

WP : calcul de weakest precondition
AI : interprétation abstraite

- MC abstrait et AI : essentiellement un problème de curseur, entre développer les traces (précision) ou fusionner au maximum l'info (terminaison)
  - raison historique : AI plutôt sur invariance, MC plutôt sur safety
  - les 2 se rapprochent : trace partitioning (AI) et abstractions (MC)
- MC abstrait / AI et WP : un problème d'expressivité (domaines restreints ou FOL), et le problème de la gestion des boucles
  - raison historique : WP plutôt sur correction fonctionnelle, MC/AI plutôt sur erreur d'exécution / de séquencement
  - technos pour BMC et WP sont maintenant très proches ...

Liens avec le model checking de systèmes infinis

- PA : permet de gérer un ensemble infini de configurations (données)
  . mais ne peut rien pour une structure de contrôle infinie !
  . ex : fonctions récursives, # arbitraire de process
- Certaines classes étudiées pour le model checking ∞ gèrent un contrôle infini (mais des données finies)
  . pushdown systems : récursivité
  . réseaux de Petri : # arbitraire de process (via counting abstraction)
- On peut combiner dans certains cas !
  . SDV

Conclusion

- le software model checking fonctionne !
- attention, plus robuste pour le débug que pour la preuve
- tendance : se ramène de + en + à la logique
- remarque 1 : on a perdu la liveness ...
- remarque 2 : langages plus évolués (objets, fonctionnels) ou plus bas (asm) ?