La vérification par énumération explicite

Gérard Berry

Collège de France
Chaire Algorithmes, machines et langages
gerard.berry@college-de-france.fr

Cours 6, 06 avril 2016

Et séminaire de Stéphanie Delaune (ENS Cachan)

Grand merci à Hubert Garavel, Radu Mateescu et Robert de Simone (Inria)

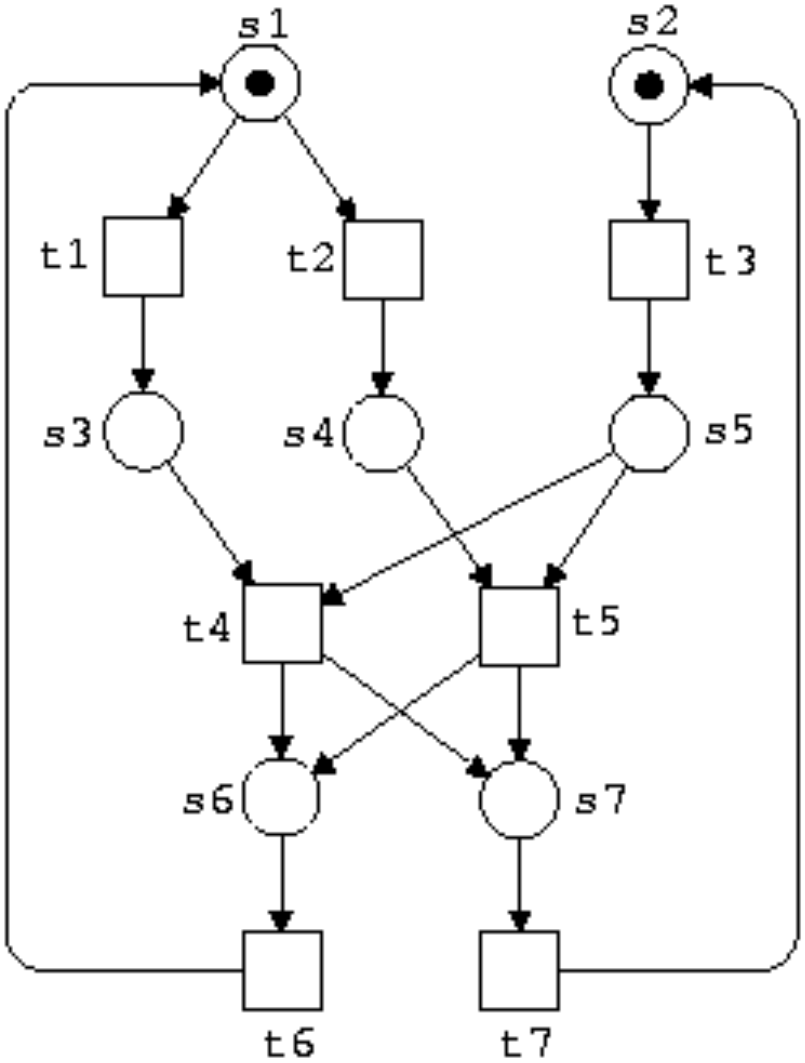
COLLÈGE DE FRANCE
1530

Model-checking explicite (ou énumératif)

- Naissance du model-checking dans les années 1980
  - CESAR par Queille et Sifakis (1982)
  - EMC par Clarke et Emerson

  → Prix Turing 2007 à Clarke, Emerson et Sifakis
- S'adresse à des modèles d'états finis, vérifiés par énumération explicite de leurs états et transitions
- Application dans ce cours : algorithmes distribués asynchrones
  - exclusion mutuelle, protocoles de communication, caches, votes, etc.
  - domaine où l'intuition est largement inopérante
- Précurseurs : Dijkstra, les réseaux de Petri et leurs systèmes de vérification

Réseaux de Petri

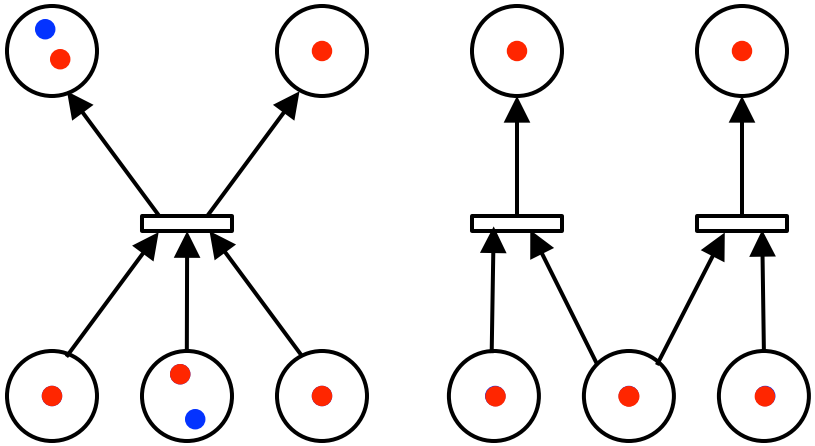


Source Univ. Stuttgart

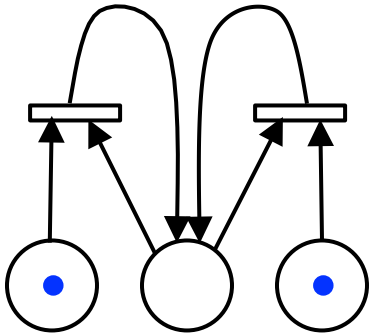
places → états
transitions → action
jetons → parallélisme,
conflit et choix

expression simple
des phénomènes clefs
du parallélisme

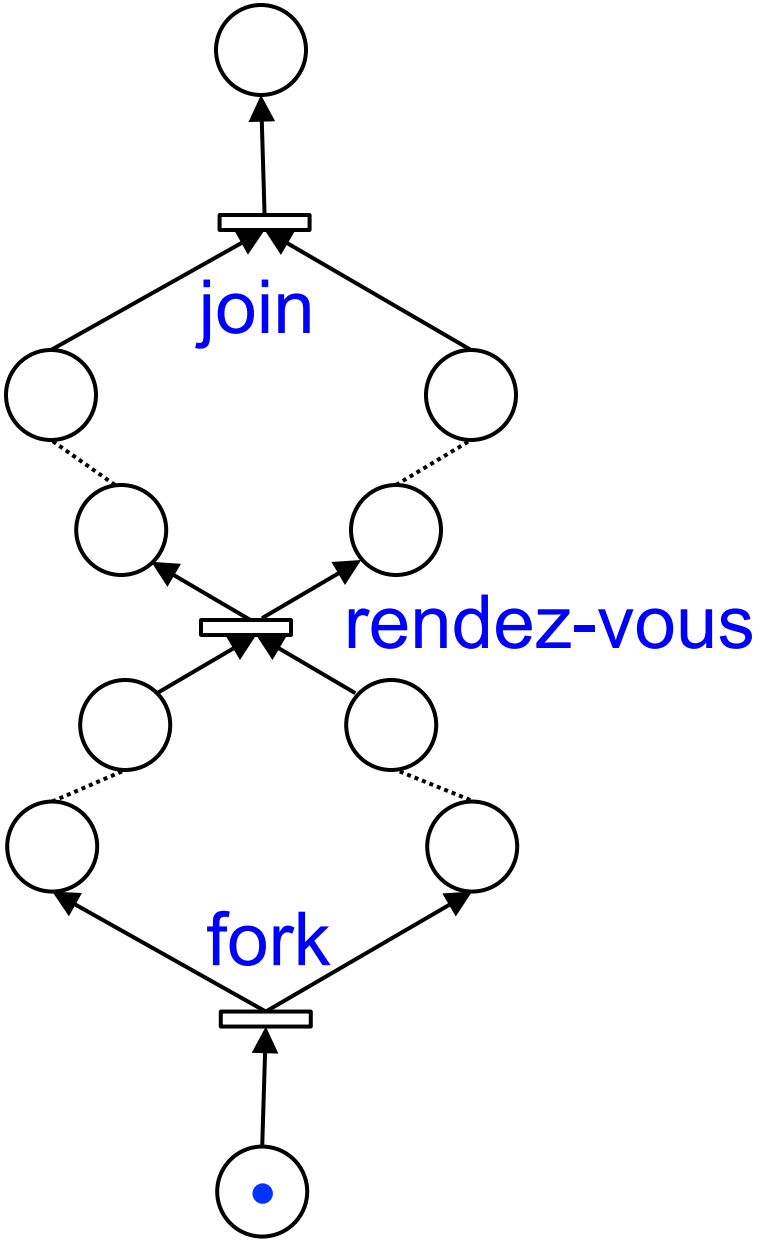
expressivité limitée et
manque de hiérarchie



transitions conflit



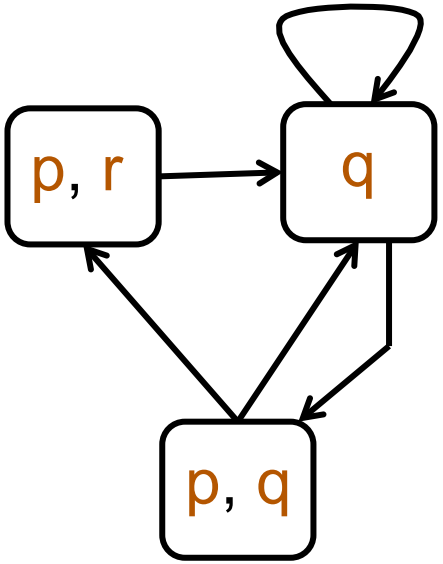
interblocage
(deadlock)



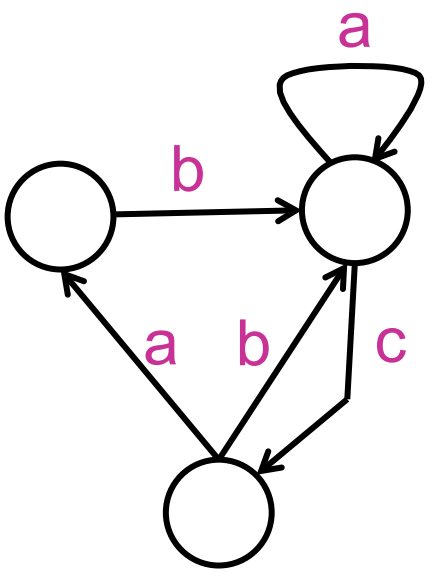
Trois types de modèles explicites pour la représentation des graphes de calcul

- Structures de Kripke : graphes orientés dont les états portent les propriétés
- Systèmes de transitions : graphes orientés dont les arcs portent les propriétés
- Systèmes mixtes : graphes orientés où états et arcs portent les propriétés

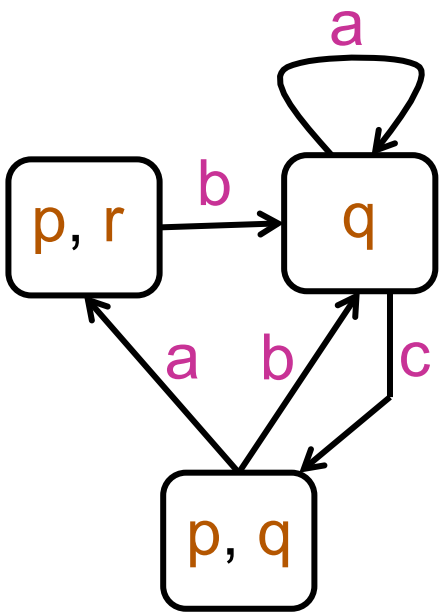
  (ex. automates temporisés, cf. cours 5 du 23 mars 2016)



structure
de Kripke



structure
de transitions



structure
mixte

Ces structures ne sont pas données explicitement,
mais implicitement par des spécifications parallèles.
Le modèle explicite est construit par énumération

Vérifieurs et propriétés

- Deux systèmes principaux : SPIN et CADP
- Beaucoup d'autres systèmes moins développés
- SPIN, G. Holtzmann, Bell Labs, débuts en 1991
  - système léger fondé sur le langage de spécification dédié Promela
  - processus d'états finis, logique temporelle linéaire (LTL) et équité (fairness)
  - accent sur légèreté et efficacité (style Unix)
- CADP, H. Garavel, R. Mateescu et. al, Inria Grenoble, 1986 →
  - fondé sur les calculs de processus communicants
  - système très riche en langages traités et modes de vérification
  - logique temporelle arborescente, bisimulation, etc.
  - traitement probabiliste (chaînes de Markov)
  - grande boîte à outils, accent sur la généralité et les performances

Et deux communautés d'utilisateurs actives, académiques et industriels

Agenda

1. SPIN
2. Calculs de processus et bisimulation
3. CADP
4. Conclusion

SPIN et Promela (G. Holtzmann)

- Développé au Bell Labs par Gerard Holtzmann dans les années 1990
- SPIN = Simple Promela Intepreter
- Promela = Process Meta Language
- Langage de spécification de systèmes parallèles asynchrones finis, style proche de la programmation classique.
- Explore la structure de Kripke engendrée par la spécification (l'information est dans les états)
- Logique temporelle linéaire (LTL) avec équité (fairness) traduite en automates de Büchi
- Vérification d'invariants et de progrès, détection de blocages, famines, sous conditions d'équité, etc.

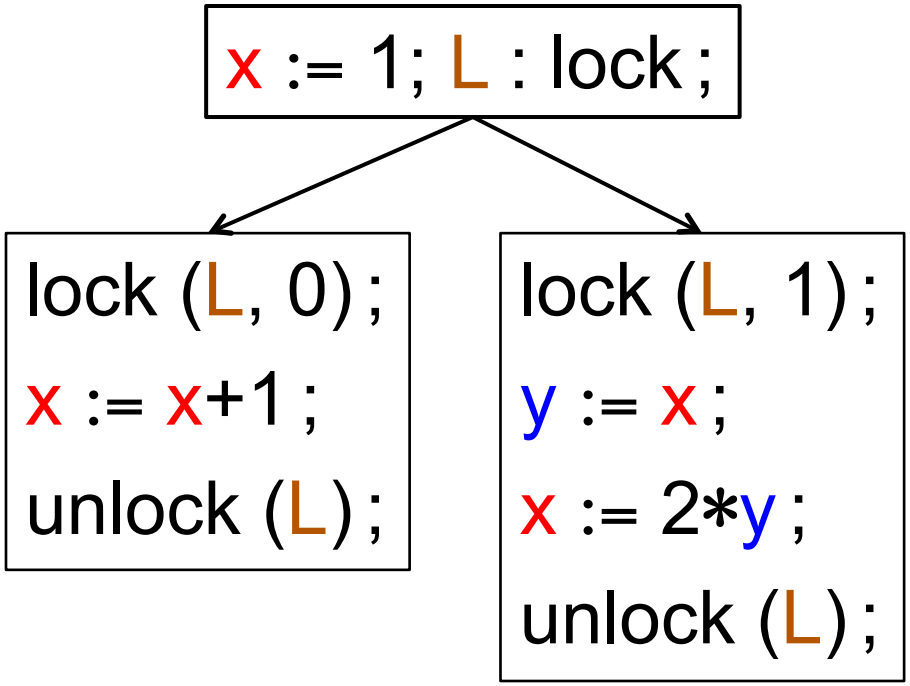
Ingrédients de Promela

- Types finis, variables classiques, affectation, tests, boucles
- Processus parallèles, créés statiquement ou dynamiquement (en nombre fini)
- Asynchronisme des processus, mais primitive d'atomicité de suite d'instructions (cf. *Test and Set*)
- Communication par variables partagées, buffers bornés, ou rendez-vous
- Tests bloquants
- Invariants, assertions et vivacité pour la vérification

Gros travail sur l'efficacité !

Exclusion mutuelle

- Assurer que « x := x+1 » et « y := x ; x := 2*y » sont exécutés de façon exclusive (ou atomique)
- Méthode : utiliser un verrou partagé pour délimiter des sections critiques



- verrou à 2 : TestAndSet
  Peterson
- verrou à n : Boulangerie
  L. Lamport

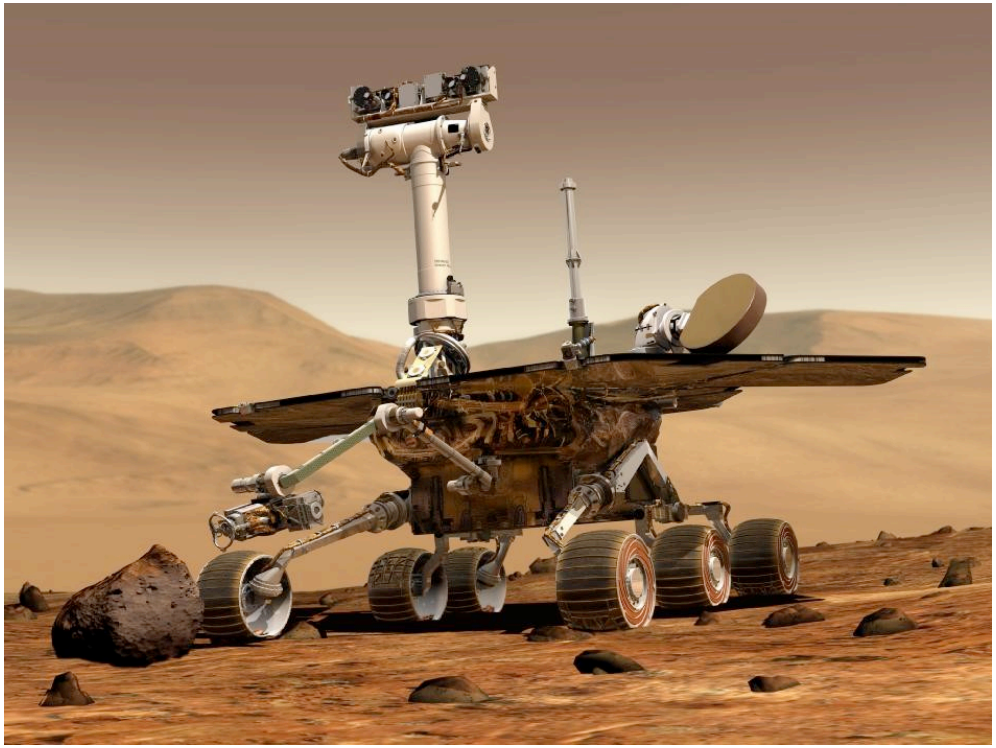
Algorithme de Peterson (1981) en SPIN

```
bool dem[2], vict; // booléens partagés
byte ncrit; // nombre de processus en section critique
active [2] proctype user ()  { // les deux processus
again: dem[ _pid] = 1;
        vict= _pid;
        (dem[1-_pid] == 0 || vict != _pid) →
        ncrit++; // critical section
        assert(ncrit == 1);
        ncrit--;
        dem[ _pid] = 0;
       goto again
}
```

Algorithme de Peterson en SPIN

```
$ spin -a peterson.pml
$ cc -o pan pan.c
$ ./pan
(Spin Version 6.4.5 -- 1 January 2016) + Partial Order Reduction
 Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid end states      +
State-vector 28 byte, depth reached 24, errors: 0
      40 states, stored
      27 states, matched
      67 transitions (= stored+matched)
       0 atomic steps
hash conflicts:        0 (resolved)
```

Le bug du robot martien PathFinder (1997)



Amarsissage parfait, mise en route parfaite,
premières photos historiques, mais au bout d'un moment,
pertes de données et reboots intempestifs.

Grattage de têtes à la NASA !

Le bug du robot martien PathFinder (1997)

- Toutes la nuit, les ingénieurs essaient de reproduire le bug au sol, en simulation avec trace de tous les événements
- Au petit matin, le dernier ingénieur debout le voit se produire
- Trois tâches High, Medium et Low veulent accéder au bus avec des priorités respectives haute, moyenne et basse
- Au blocage, la tâche Low a pris le bus; elle est active mais non exécutable car High s'est mise en attente du bus. La tâche Medium se met aussi en attente, et tout est bloqué !

Modélisation en SPIN (réduite à 2 tâches)

```
mtype = { free, busy, idle, waiting, running };
mtype H = idle; mtype L = idle; mtype mutex = free;
active proctype High() {
end: do
     :: H = waiting ;
     atomic { mutex == free -> mutex = busy } ;
     H = running ;
     // critical section
     atomic { H = idle; mutex = free }
 od
}
active proctype Low() provided (H == idle) {
{ end: do
     :: L = waiting ;
     atomic { mutex == free -> mutex = busy};
     L = running ;
     // critical section
     atomic { L = idle; mutex = free }
  od
}
```

SPIN à l'œuvre sur Pathfinder

```
$ spin –a pathfinder.pmc
$ cc –o pan pan.c
$ ./pan
pan:1: invalid end state (at depth 4)
pan: wrote pathfinder.pml.trail+

State-vector 28 byte, depth reached 5, errors: 1
        5 states, stored
        1 states, matched
        6 transitions (= stored+matched)
        2 atomic steps
```

Trace du deadlock

```
$ spin -t -p pathfinder.pml
  1:    proc  1 (low:1) pathfinder.pml:40 (state 1)
                [L = waiting]
  2:    proc  1 (low:1) pathfinder.pml:41 (state 2)
                [((mutex==idle))]
  3:    proc  1 (low:1) pathfinder.pml:41 (state 3)
                [mutex = busy]
  4:    proc  1 (low:1) pathfinder.pml:42 (state 5)
                [L = running]
  5:    proc  0 (high:1) pathfinder.pml:27 (state 1)
                  [H = waiting]
spin: trail ends after 5 steps
```

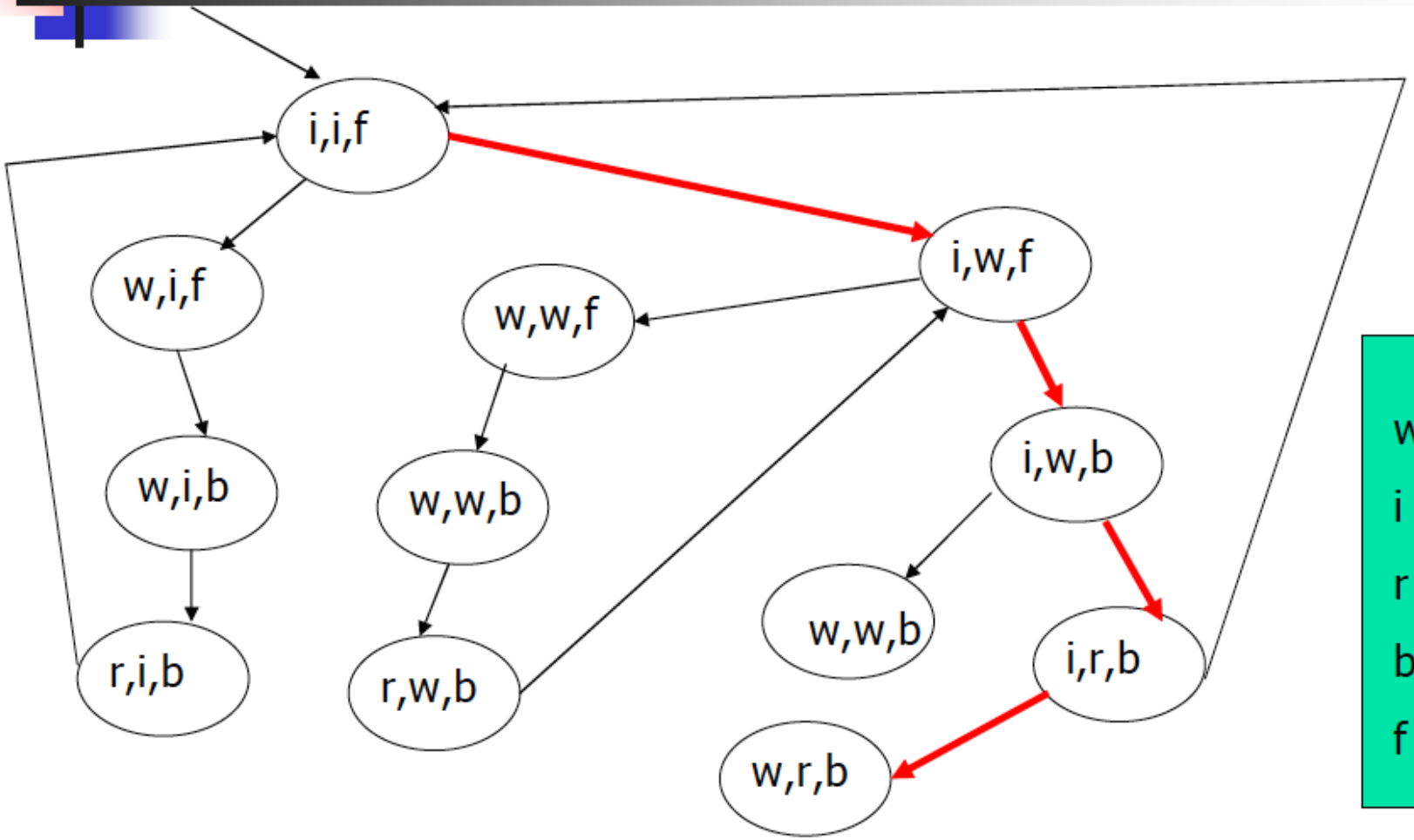
Explication du conflit

- Source du conflit
  - H valant idle, Low peut passer à L = wait
  - H valant toujours idle, Low peut passer à L = running
  - mais High peut passer librement à H = wait
- Deadlock :
  - puisque H = wait, Low n'est plus activable à cause de la clause « provided (H == idle) »
  - Donc Low ne peut plus rendre le verrou.
  - Ce qui fait que High et Low sont toutes deux bloquées !

Bug corrigé par téléchargement d'un patch (AR 15mn!)
Ils avaient eu la bonne idée de laisser l'OS en mode debug...

Le bug du robot martien PathFinder

State Space Graph



w ≡waiting

i ≡idle

r ≡running

b ≡busy

f ≡ free

source CADP

Progrès, LTL et équité

- Absence de blocage local : SPIN permet d'étiqueter des instructions par « progress: » pour vérifier que toute trace infinie passe infiniment souvent par une instruction étiquetée

- SPIN permet de valider des formules LTL (linéaire). Ces formules sont traduites en automates de Büchi, dans lesquels une trace doit passer infiniment souvent pas un état acceptant (facile).

- LTL permet d'exprimer des conditions d'équité infinie.
  Exemple : vérification en supposant l'ordonnanceur équitable

Vérifier ϕ en supposant que si p exécute infiniment souvent l'instruction étiquetée A, alors q exécute aussi infiniment souvent celle étiquetée B :

$((\Box\Diamond\ p@A)\Rightarrow(\Box\Diamond\ q@B)) \Longrightarrow \phi$

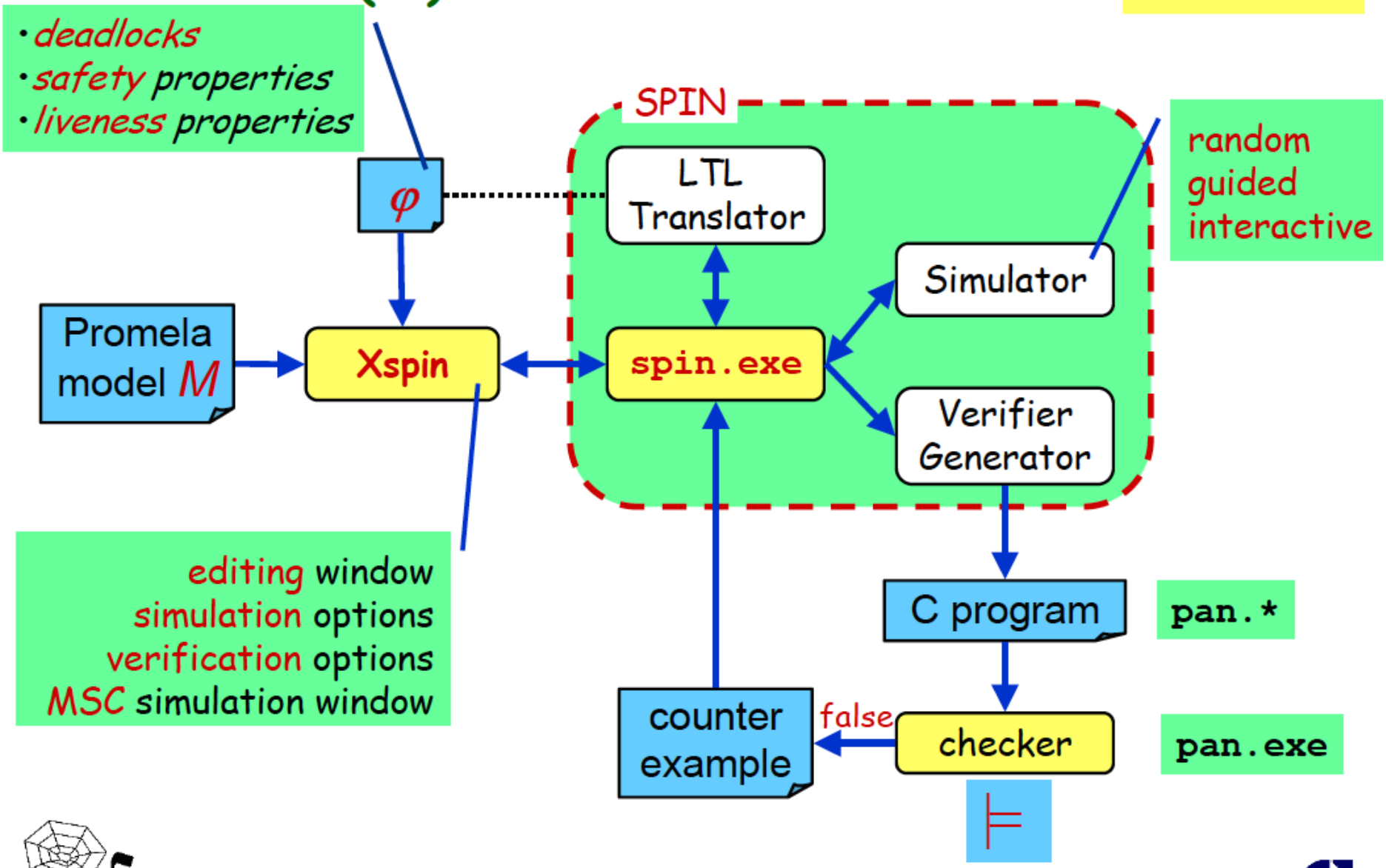
Vérification d'équité générale faible ou forte (weak / strong fairness)

- Equité faible : tout processus qui reste exécutable infiniment longtemps finira par être exécuté
- Equité forte: tout processus qui reste exécutable infiniment souvent finira par être exécuté

- SPIN traite l'équité faible (par recopie astucieuse des structures de Kripke)

(X)SPIN Architecture

$M \models \varphi$



Réduire la taille de l'exploration

L'exploration stocke des états, typiquement de ~1Kbits,
et explore les transitions entre états.

Méthodes d'accélération :

1. Réduction d'ordre partiel : quand des actions commutent, ne visiter qu'une des permutations
2. Abstraction par hashcode : hacher les états
   - exactement, pour les retrouver vite
   - ou approximativement, sans tester les collisions
   - ou sur 64 bits, eux-même re-hashés (*hascompact*, D. Peled)

- Partial Order Reduction (cont.)

Suppose the statements of P1 and P2 are all local.

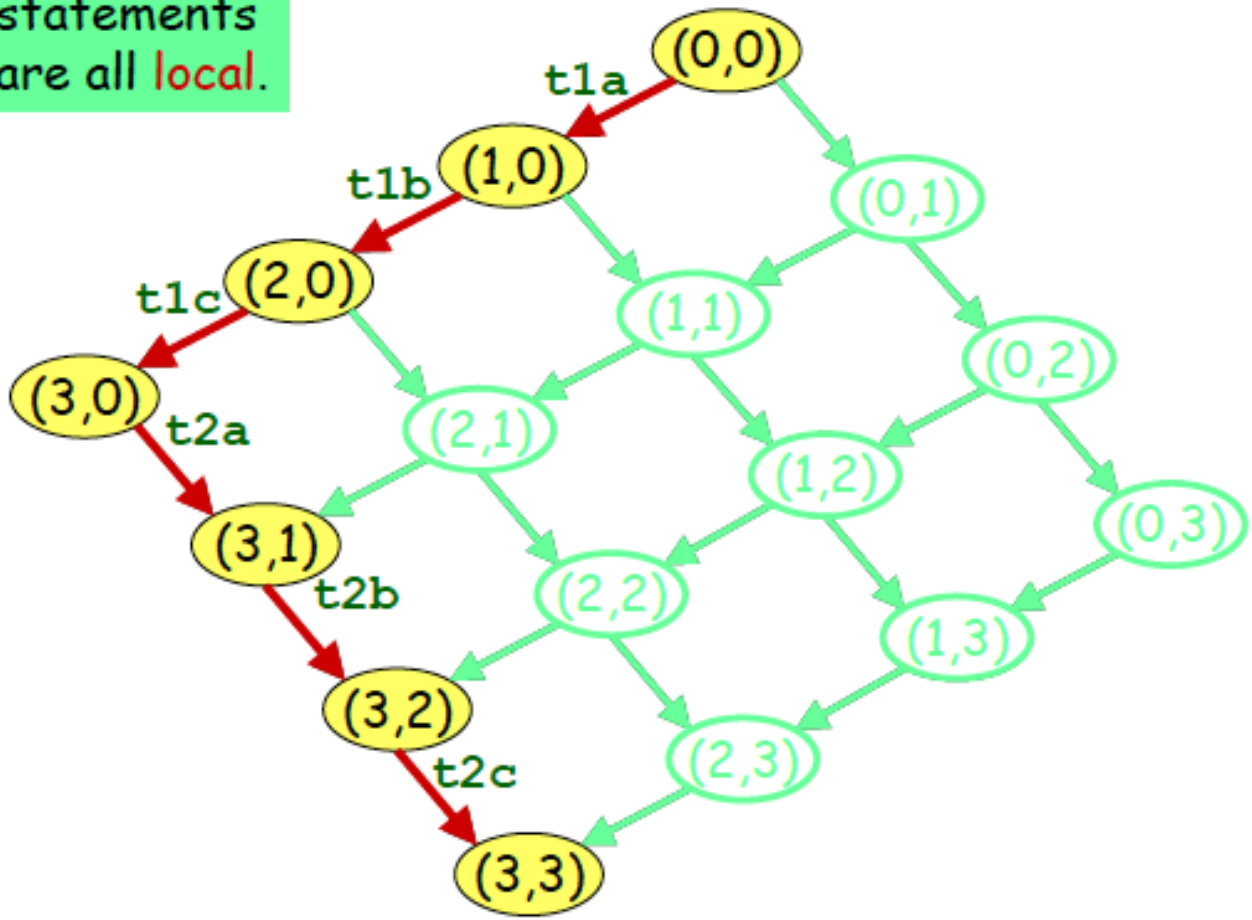
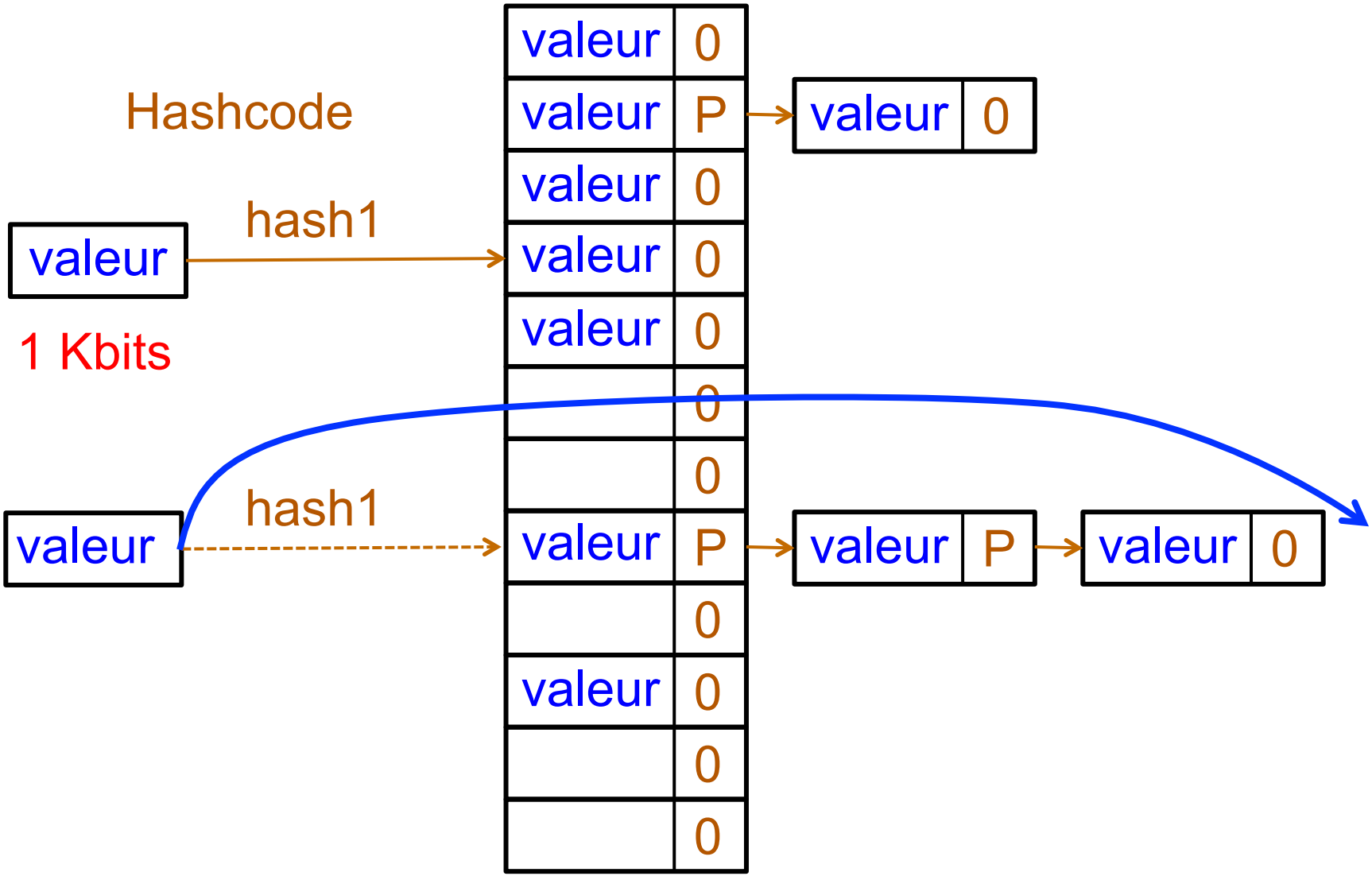


Table de hash classique



La mémoire se remplit trop vite !

Abstraction 1 : ne pas gérer les collisions !

- Quand une collision se produit, on l'ignore simplement, en faisant semblant d'avoir déjà vu le nouvel état
- Pas de faux positifs, plus efficace en mémoire, mais risque d'être fortement incomplet, laissant échapper des bugs

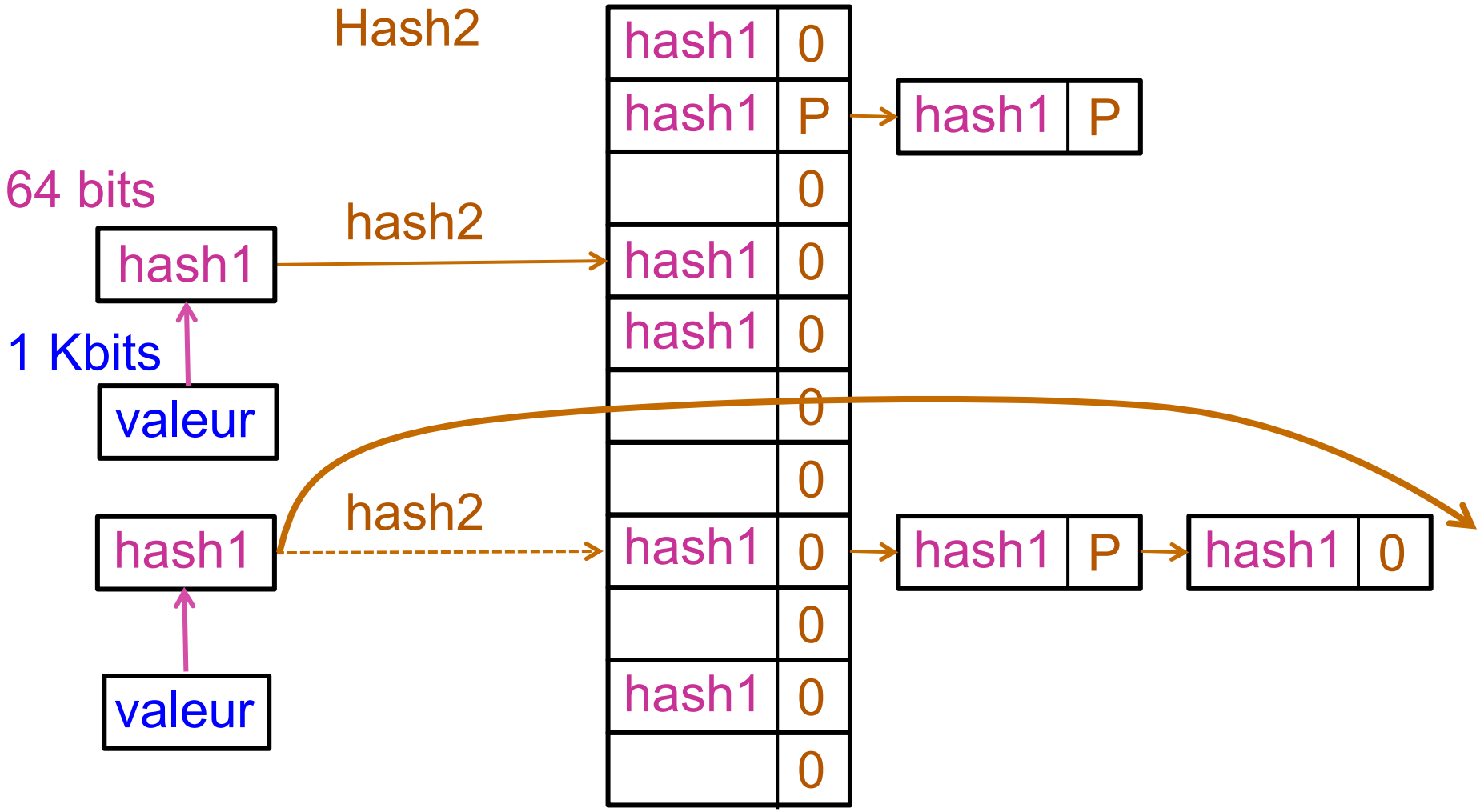
Abstraction 2 : Hacher sur 1 bit (ou 2, ou plus) !

- Hacher chaque état sur une adresse dans une table de bits et mettre à 1 le bit à cette adresse (1 bit par état, plus 1Kbits !)
- Pas de faux positifs, beaucoup plus efficace en mémoire, mais risque d'être assez incomplet
- Mettre à 1 deux bits avec deux fonctions de hash différentes (défaut pour Spin), ou la même chose avec plus de bits

➔ Nettement mieux, mais mémoire pas idéalement utilisée

➔ Rendement optimum pour ~20bits, mais temps prohibitif !

Meilleur : Hashcompact (P. Wolper)



Agenda

1. SPIN
2. Calculs de processus et bisimulation
3. CADP
4. Conclusion
5. Bonus : la machine chimique (rappel de 2009)

Calculs de processus : CCS (Meije, Lotos,..)

- Actions $A = \{a,b,...\}$, $A^- = \{a^-,b^-,\ldots\}$, τ (action invisible)

$$\alpha \in A \cup A^-,\ \ \alpha^- : a \leftrightarrow a^- ,\ \mu \in A \cup A^- \cup \{\tau\}$$

- Identificateurs de processus x, y, z
- Processus p, q, r,… définis par

0	inaction
μ.p	action
p | q	parallélisme
p + q	choix non-déterministe
p \ a	restriction
p[b/a]	renommage d'action
x	identificateur de processus
rec x = p	définition récursive

Sémantique Opérationnelle Structurelle (SOS – G. Plotkin)

$$\mu.p \xrightarrow{\mu} p$$

$$\frac{p \xrightarrow{\mu} p'}{p+q \xrightarrow{\mu} p'} \qquad \frac{q \xrightarrow{\mu} q'}{p+q \xrightarrow{\mu} q'}$$

$$\frac{p \xrightarrow{\mu} p'}{p|q \xrightarrow{\mu} p'|q} \qquad \frac{q \xrightarrow{\mu} q'}{p|q \xrightarrow{\mu} p|q'}$$

$$\frac{p \xrightarrow{\alpha} p' \quad q \xrightarrow{\alpha^-} q'}{p|q \xrightarrow{\tau} p'|q'}$$

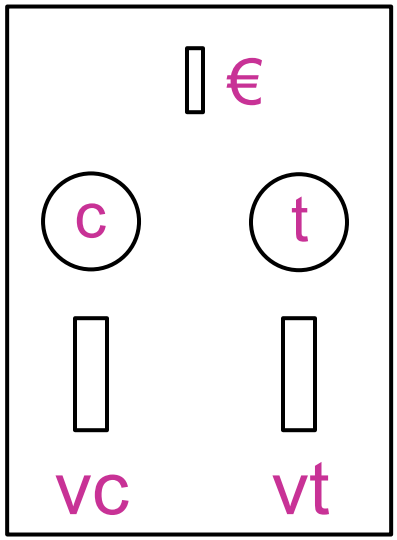
Sémantique Opérationnelle Structurelle

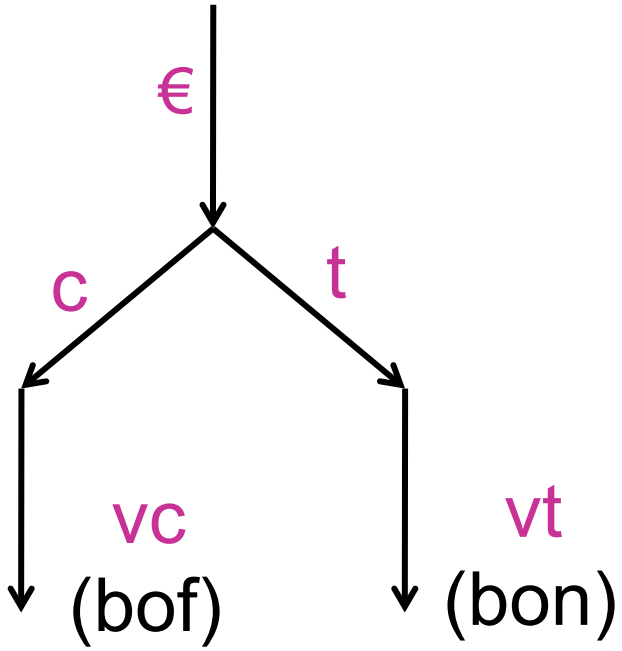
$$\frac{p \xrightarrow{a} p'}{p\,[\,b/a] \xrightarrow{b} p'\,[b/a]} \qquad \frac{p \xrightarrow{a^-} p'}{p\,[\,b/a] \xrightarrow{b^-} p'\,[b/a]}$$

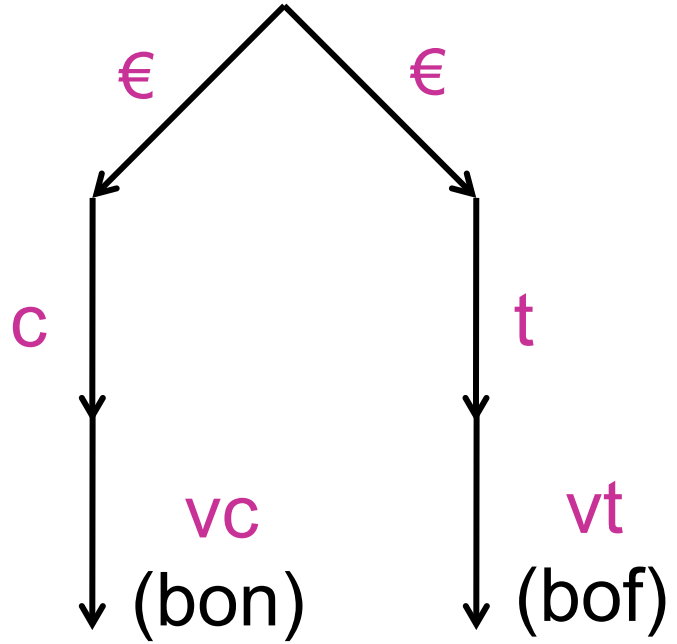
$$\frac{p \xrightarrow{\mu} p' \quad \mu \neq a, a^-}{p \setminus a \xrightarrow{\mu} p' \setminus a}$$

$$\frac{p \xrightarrow{\mu} p'}{\mathrm{rec}\ x{=}p \xrightarrow{\mu} p'\,[x \leftarrow \mathrm{rec}\ x{=}p]}$$

Machine à café anglaise ou italienne







Laquelle préférez vous?

Equivalentes pour la logique temporelle linéaire

(€.c.vc + €.t.vt)*

Exemples CCS

- Les machines à café :

  rec A = €⁻.(c⁻.vc.A + t⁻.vt.A)

  $A \xrightarrow{€^-} c^-.vc.A + t^-.vt.A$

  rec I = €⁻.c⁻.vc.I + €⁻.t⁻.vt.I

  $I \xrightarrow{€^-} c^-.vc.dist\_n$

  $I \xrightarrow{€^-} t^-.vt.dist\_n$

- Pile électrique standard :

  rec PileX = jus.PileX + τ.0

- Thm : la pile Wonder ne s'use que si l'on s'en sert :

  rec Wonder = jus.(Wonder + τ.0)

La bisimulation forte (R. Milner)

- A et I engendrent le même langage
  mais ont des comportements interactifs différents
- La bisimulation (forte) compare les possibilités, pas seulement les actions effectuées.

  C'est la plus grande équivalence vérifiant

$$p \approx q \text{ ssi } \begin{cases} p \xrightarrow{\mu} p' \Rightarrow \exists q'.\ q \xrightarrow{\mu} q' \text{ avec } q' \approx p' \\ q \xrightarrow{\mu} q' \Rightarrow \exists p'.\ p \xrightarrow{\mu} p' \text{ avec } p' \approx q' \end{cases}$$

$$A \xrightarrow{€^-} c^-.vc.dist\_d + t^-.vt.dist\_d \xrightarrow{t^-} vt.dist\_d$$

$$I \xrightarrow{€^-} c^-.vc.dist\_d \not\xrightarrow{t^-}$$

La bisimulation forte est une congruence

- La bisimulation forte exprime qu'il est impossible de distinguer deux graphes de transitions en les parcourant
- C'est une congruence pour les opérateurs du calcul :

  $p \approx p' \Longrightarrow \mu.p \approx \mu.p'$, $p \setminus a \approx p' \setminus a$, $p[b/a] \approx p'[b/a]$

  $\text{rec } x=p \approx \text{rec } x=p'$

  $p \approx p', q \approx q' \Longrightarrow p+q \approx p'+q'$, $p \mid q \approx p' \mid q'$

- Tout terme admet une forme normale unique, calculée efficacement en $n \log(n)$ grâce à un bel algorithme de Tarjan
- Applications pratiques :

  test d'égalité de processus → équivalence de spécifications

  réduction par bisimulation → preuves compositionnelles (CADP)

La bisimulation observationnelle

- Mais les processus font souvent des actions silencieuses τ issues de leurs communications internes. Or, $a.\tau.0$ et $a.0$ ne sont pas fortement bisimilaires, même s'ils se valent pour les observateurs extérieurs

- La bisimulation observationnelle $\approxeq$ compare des processus vus d'un observateur externe, pour qui une chaîne d'actions internes n'est pas observable

- Elle se définit comme bisimulation forte pour l'observation

$$p \overset{\alpha}{\rightsquigarrow} p' \quad = \quad p \overset{\tau^*.\alpha.\tau^*}{\dashrightarrow} p'$$

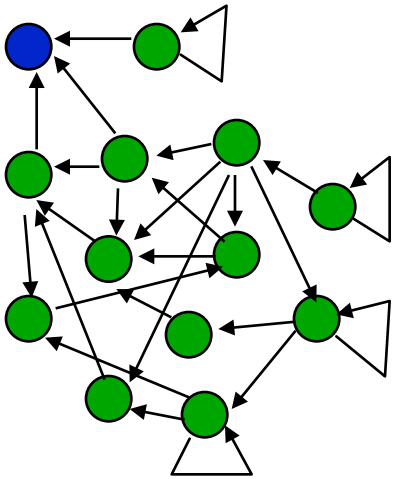
- Forme normale unique, réduction efficace, mais pas congruence

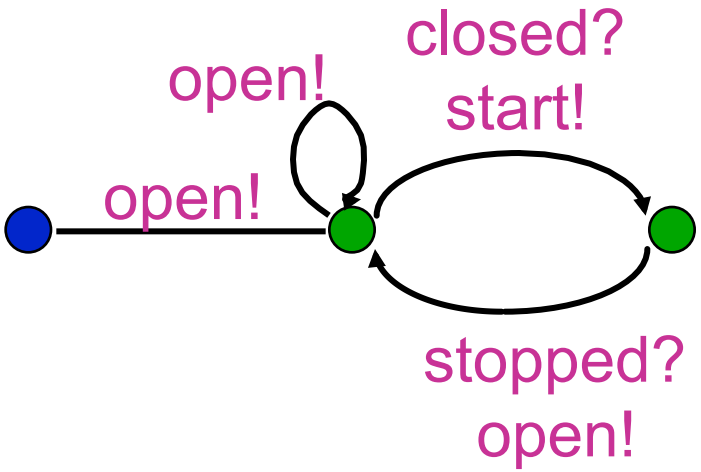
car $a.0 \approxeq \tau.a.0$ mais $a.0+b.0 \not\approxeq \tau.a.0+b.0$

Vérification visuelle de l'ascenseur

Sûreté : l'ascenseur ne peut pas voyager la porte ouverte

Réduction par bisimulation observationnelle
en cachant tout sauf start!, open!, stopped?, closed?





Non-interférence de services

rec *cycler* = a^-.*start*.(*b*.*nil* | *end*.*cycler*)

rec *scheduler_2* = local *a1*, *a2* *in*

(*a1*.0

| *cycler* [*a1* / *a*, *a2* / *b*, *start1* / *start*, *end1* / *end*]

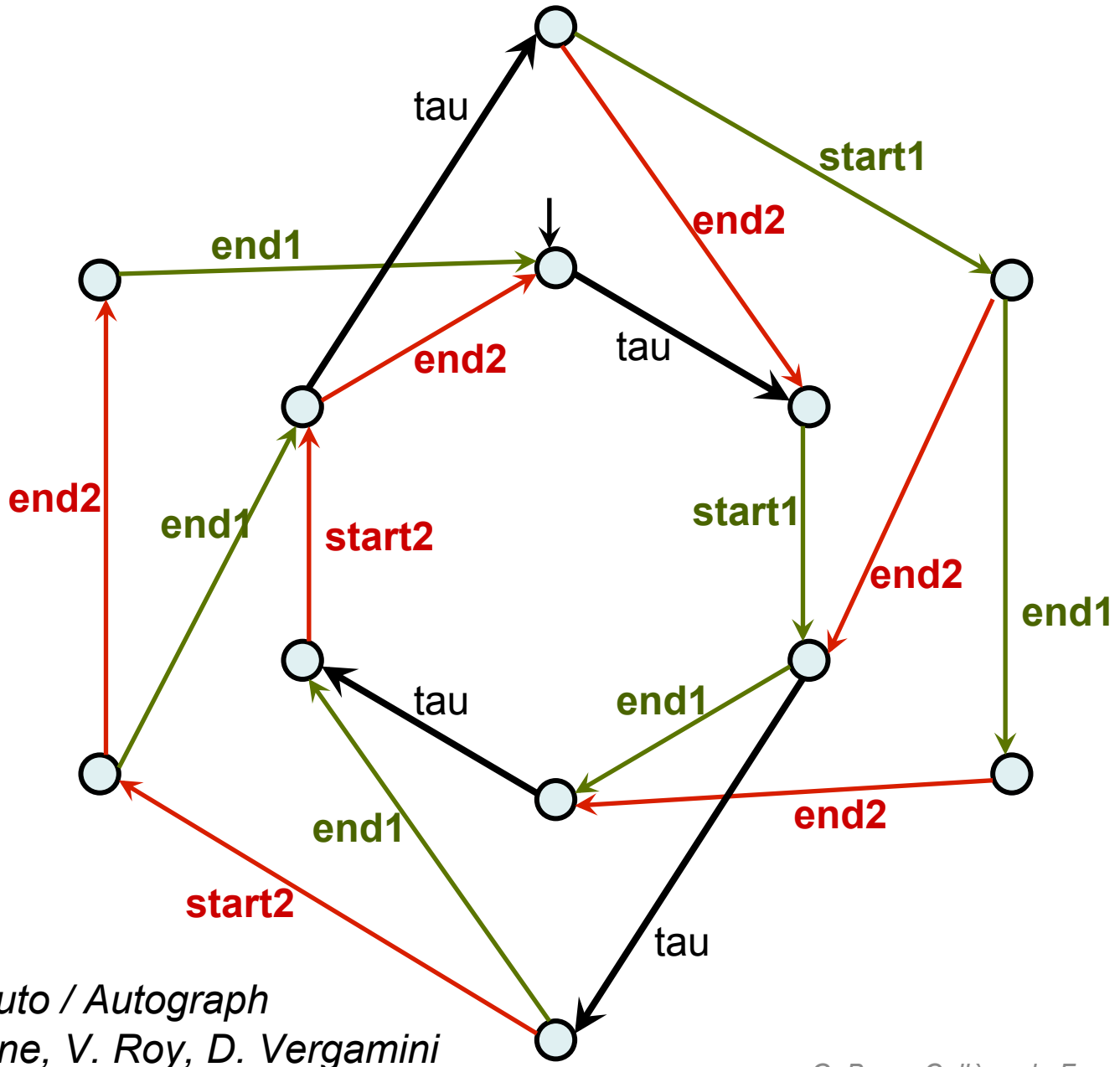
| *cycler* [*a2* / *a*, *a1* / *b*, *start2* / *start*, *end2* / *end*])

1. sûreté : les deux *cycler* n'interfèrent pas
   si on ignore *start2* et *end2*, le premier *cycler*
   marche comme s'il était tout seul
2. vivacité : on a une alternance infinie de *start1* et *start2*

→ réduire par bisimulation obervationnelle

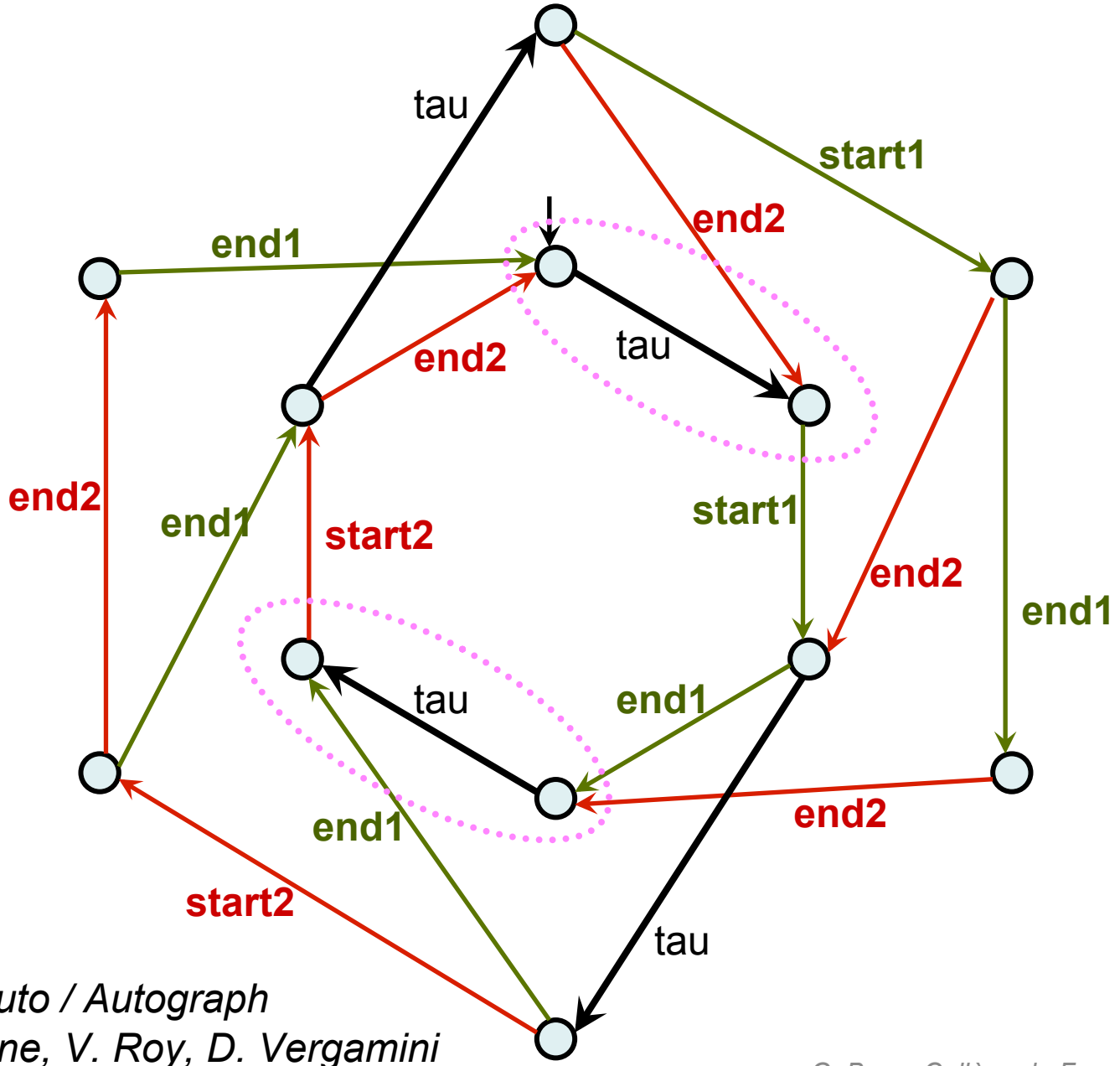
1. *scheduler_2* \ *start2*, *end2*
2. *scheduler_2* \ *end1*, *end2*

Scheduler2 expansé



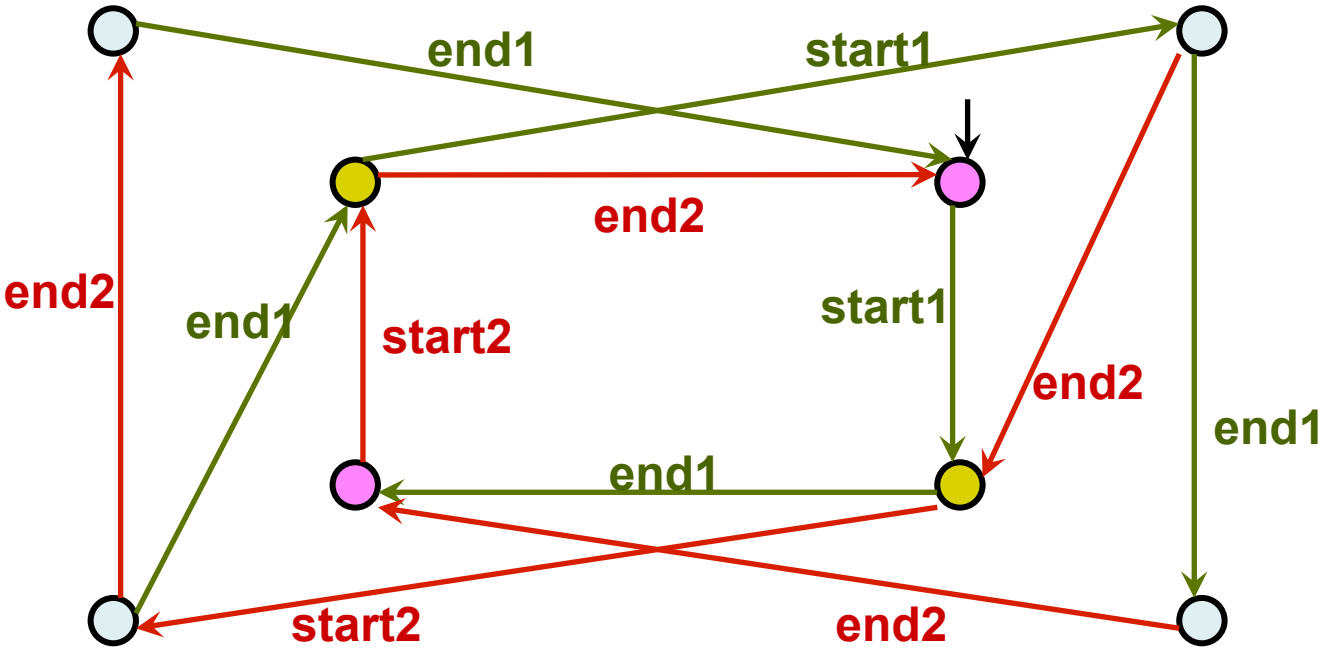
Auto / Autograph
R. de Simone, V. Roy, D. Vergamini

Réductions dans Scheduler2



Auto / Autograph
R. de Simone, V. Roy, D. Vergamini

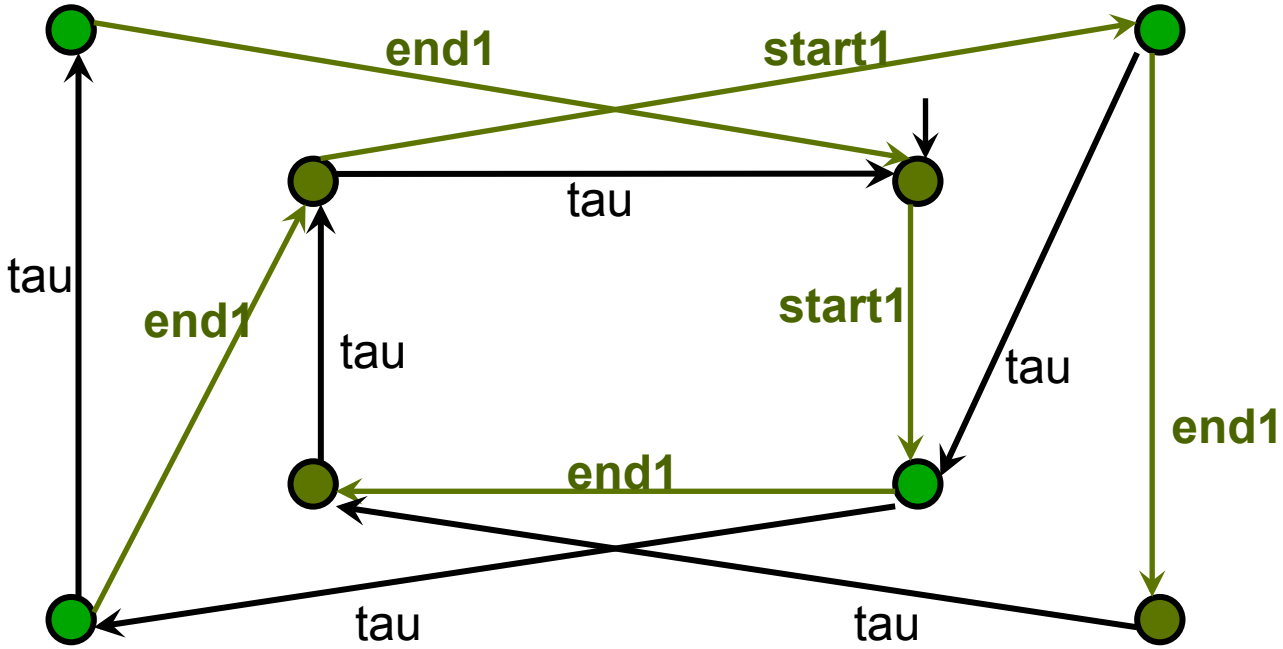
Scheduler2 réduit

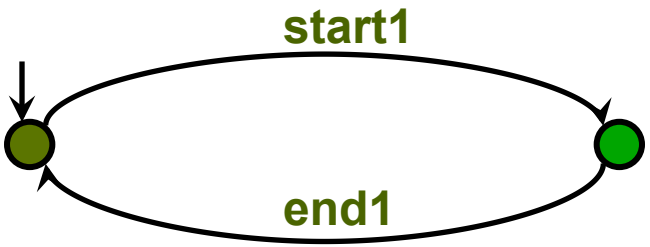


Auto / Autograph

R. de Simone, V. Roy, D. Vergamini

Masquage de start2 et end2 puis réduction

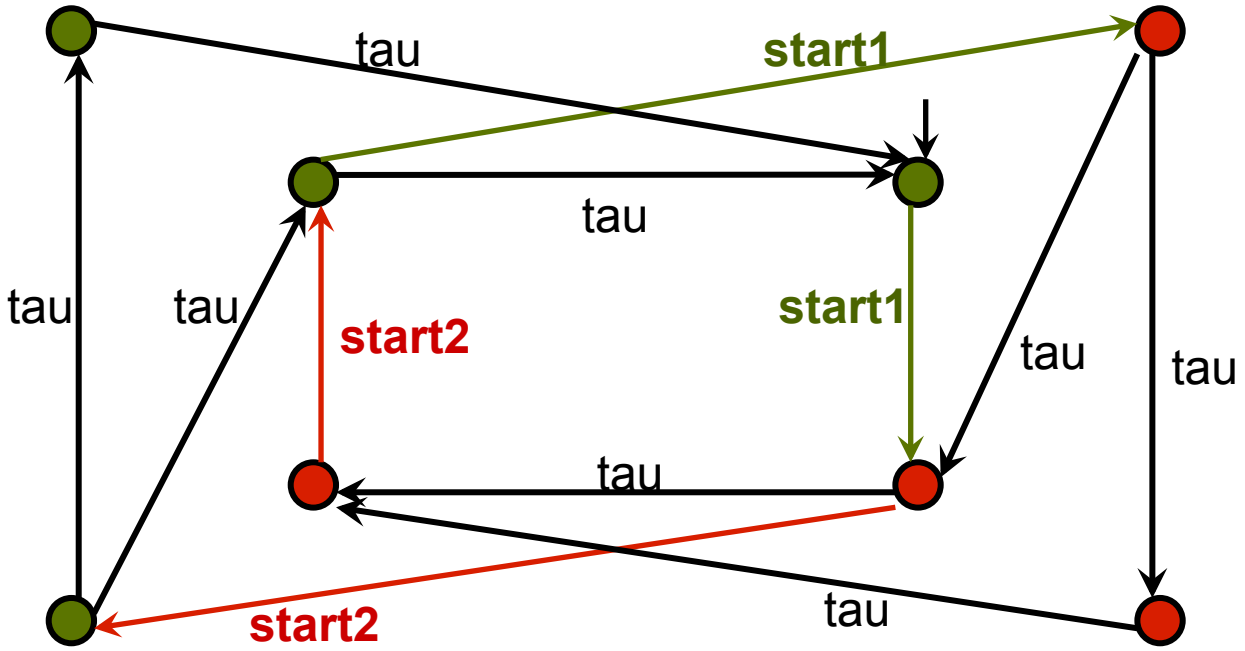


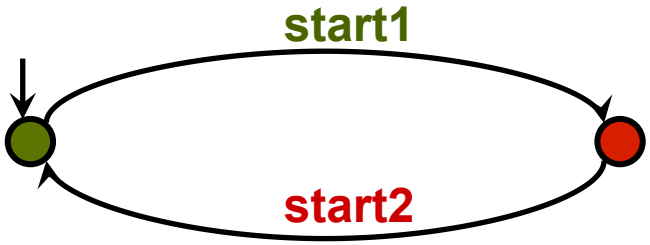


non-interférence prouvée !

Auto / Autograph
R. de Simone, V. Roy, D. Vergamini

Masquage de end1 et end2 puis réduction





vivacité prouvée !

Auto / Autograph
R. de Simone, V. Roy, D. Vergamini

Agenda

1. SPIN
2. Calculs de processus et bisimulation
3. CADP
4. Conclusion
5. Bonus : la machine chimique (rappel de 2009)

CADP : Garavel, Mateescu, et.al.

Construction and Analysis of Distributed Processes

- Un système de vérification énumérative très riche
  - calculs de processus, bisimulation, logique temporelle arborescente
  - beaucoup de langages disponibles, traduits en un formalisme interne unique (BES = Boolean Equation Systems), avec retour vers le source
  - nombreux modules (45) pour la simulation, la génération de modèles, la réduction par bisimulation, la vérification d'équivalences, la vérification de formules temporelles CTL, la vérification compositionnelle, l'évaluation de performances, etc., coordonnés par un langage de scripts

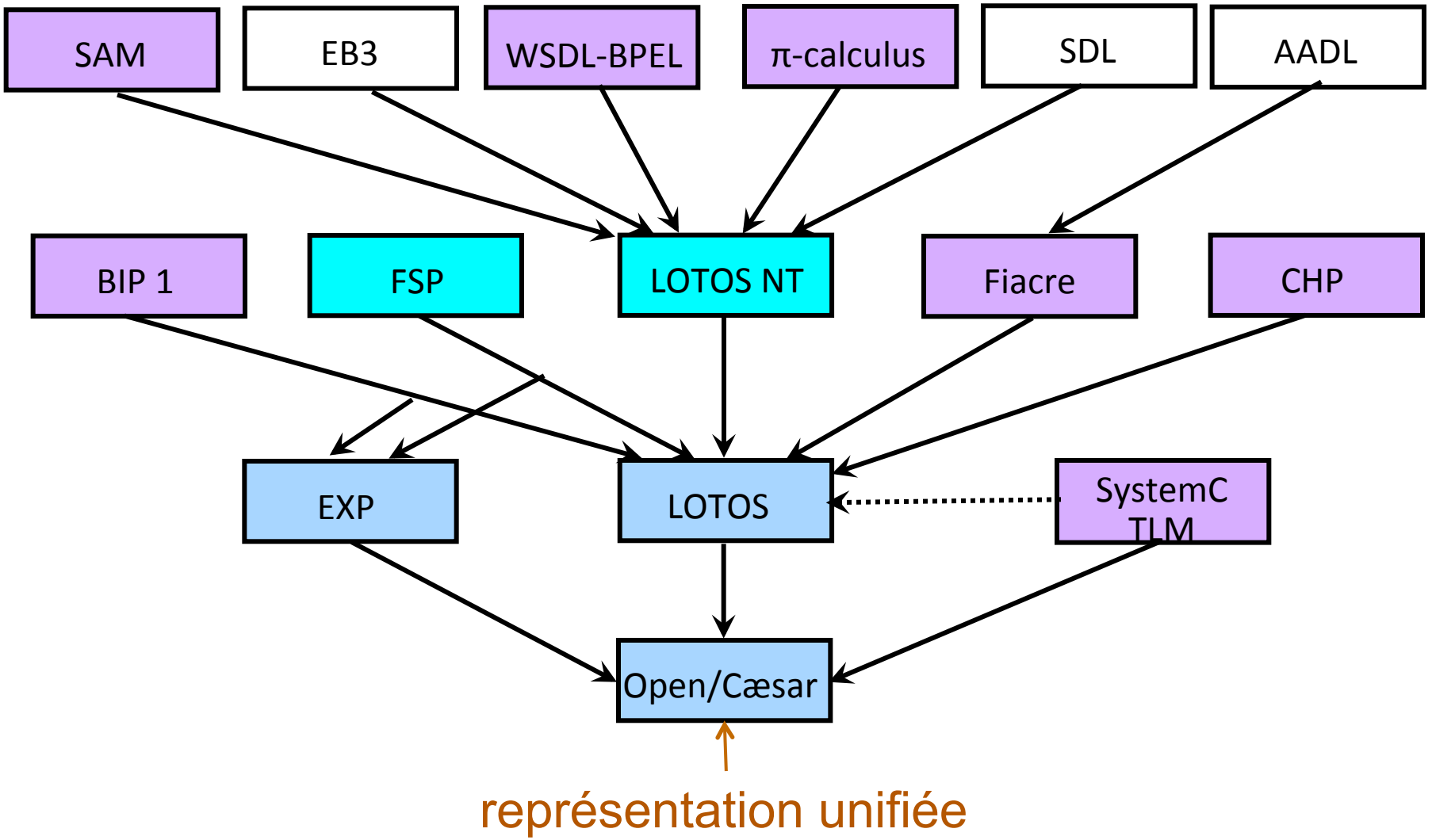
CADP : Garavel, Mateescu, et.al.

Construction and Analysis of Distributed Processes

- Un système de vérification énumérative très riche
  - génère du C pour l'efficacité. Construction de modèles exhaustive (sur disques) ou on the fly (comme SPIN)
  - fort accent sur l'efficacité le passage à l'échelle : ordres partiels, hashcodes, parallélisation, etc.
  - nombreuses applications, y compris non-académiques : algos distribués (Airbus), systèmes de fichiers, mémoire virtuelle répartie, HPC (Bull), SoCs (ST Micro), configuration et reconfiguration, etc.

Gratuit pour les académiques

Langages traités par CADP



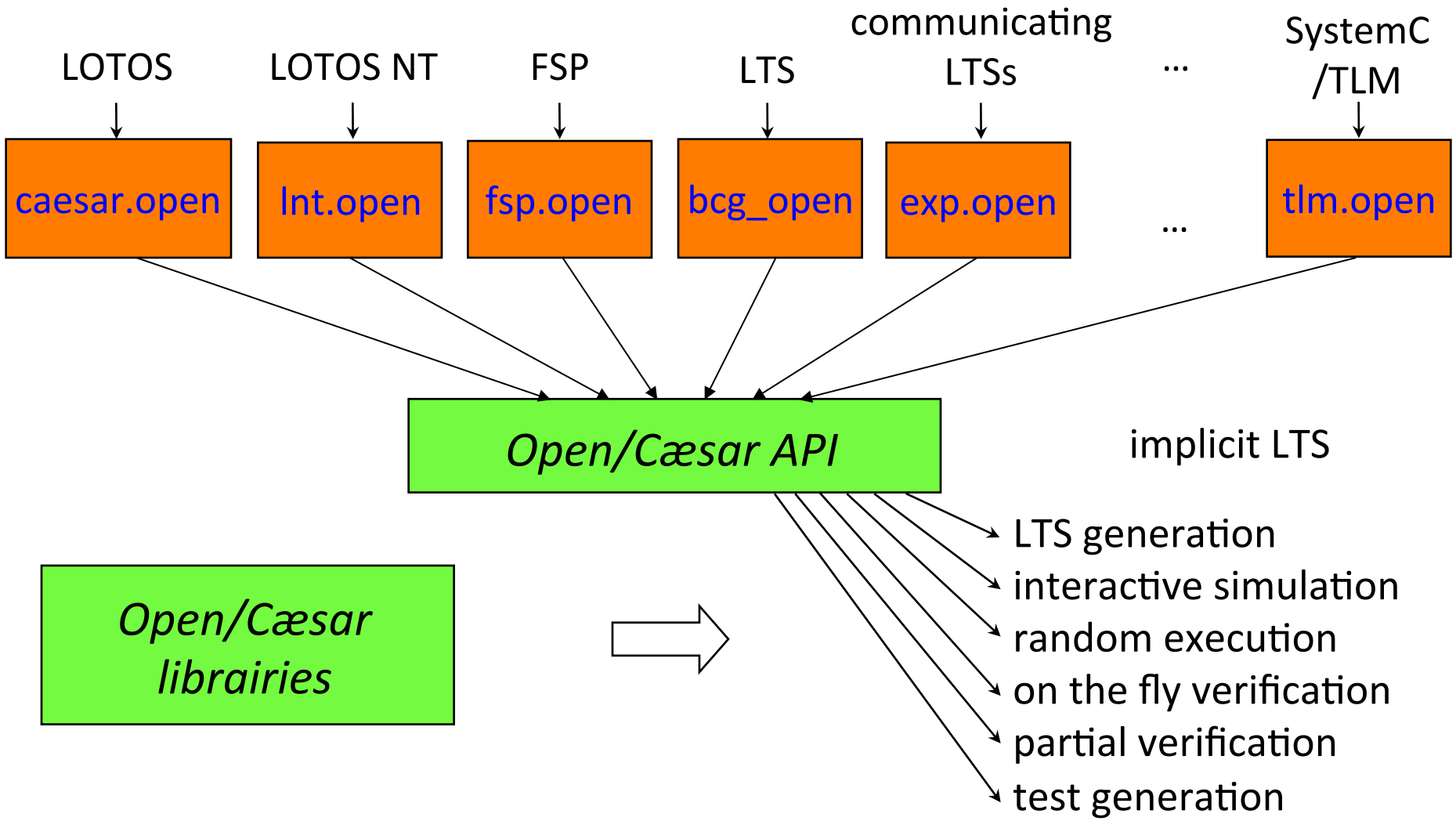
De LOTOS à Lotos NT (LNT)

- Lotos : langage normalisé issu de CCS (processus) avec passages de valeurs
  - contrôle : calcul de processus à la CCS
  - données : types abstraits ACP (axiomes et règles de réécritures)

- Problème de LOTOS (et des calculs de processus classiques)
  - le préfixage a.p d'un processus par une action est trop asymétrique et rend l'écriture d'une séquence p;q compliquée et peu naturelle
  - les formalismes de contrôle de données sont trop disjoints

- LOTOS NT
  - extension de Lotos avec des types prédéfinis et des constructions syntaxiques plus riches ressemblant plus à la programmation classique,
  - langage bien meilleur à l'utilisation
  - traduction interne en LOTOS dans CADP pour la vérification

Traitement des langages : Open / Cæsar



source CADP

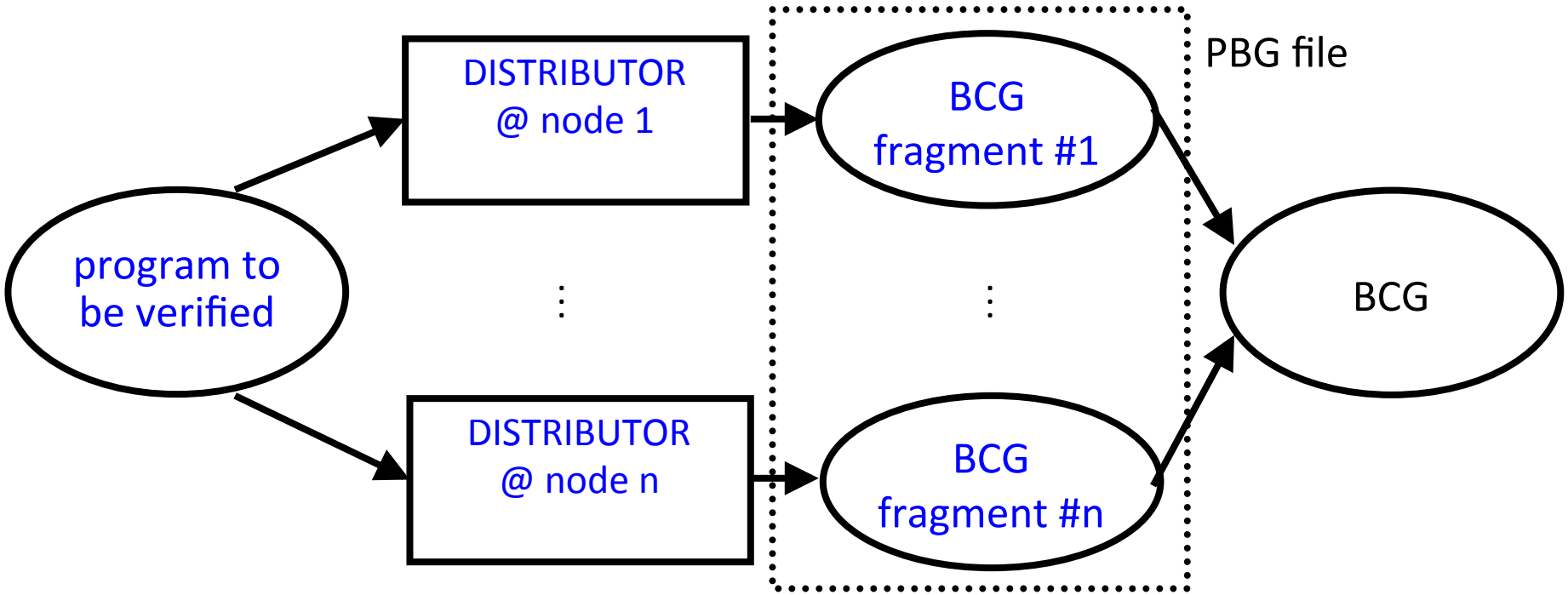
Modules de traitement énumératif

- EXECUTOR: random walk
- OCIS: interactive simulation (graphical)
- GENERATOR: exhaustive LTS generation
- REDUCTOR: LTS generation with reduction
- PROJECTOR: LTS generation with constraints
- TERMINATOR: Holzmann's bit-space algorithm
- EXHIBITOR: search paths defined by reg. expr.
- EVALUATOR: evaluation of mu-calculus formulas
- TGV: test sequence generation
- DISTRIBUTOR: distributed state space generation
- CUNCTATOR: Markov chain steady-state simulator
- …

source CADP

Distributor : parallélisation du calcul

- Génération distribuée de l'espace d'états
- Réduction locales d'ordre partiel
- Réduction locales par bisimulation



source CADP

Agenda

1. SPIN
2. Calculs de processus et bisimulation
3. CADP
4. Conclusion
5. Bonus : la machine chimique (rappel de 2009)

Conclusion

- La vérification explicite (énumérative) est excellente pour les algorithmes distribués asynchrones
- Elle est complémentaire des méthodes implicites (BDDs, SAT, SMT), qui n'aiment pas trop l'asynchronisme
- Elle est aussi complémentaire des méthodes logiques comme TLA+ de Lamport sur les mêmes sujets

La prochaine fois (13 avril 2016) :
- démo de vérification en CADP
- démo de vérification liée à Esterel en Why + SMT

Bibliographie

Edmund Clarke
The Birth of Model Checking
Dans *25 years of Model Checking,* Springer, 2008.

J-P. Queille et J. Sifakis
Specification and verification of concurrent systems in CESAR,
International Symposium on Programming, 1982.

G.J. Holzmann
The Spin Model Checker: Primer and Reference Manual
Addison-Wesley, 2006
http://spinroot.com

P. Wolper et D. Leroy
Reliable Hashing without Collision Detection
Proc. Computer Aided Verification, Springer LNCS 697, pp. 59-70, 1993

Bibliographie

R. Milner

Communication and Concurrency

Prentice Hall (1989)

http://CADP.inria.fr

V. Roy, R. de Simone

Auto/Autograph

Proc. Computer Aided Verification Conf., LNCS 531, pp. 65-75 (2005)

H. Garavel, F. Lang, R. Mateescu, W. Serwe

CADP 2011 : A Toolbox for the Construction and Analysis of Distributed Processes

Int J Softw Tools Tech Transfer (2013) 15:89-107

http://CADP.inria.fr