Prouver les programmes

La vérification de modèles (Model-Checking)

Gérard Berry

Collège de France

Chaire Algorithmes, machines et langages

gerard.berry@college-de-france.fr

Cours 5, Paris, 25/03/2015

Suivi du séminaire de Véronique Cortier (LORIA Nancy)

COLLÈGE DE FRANCE 1530

Les devises de Leslie Lamport

- A system specification consists of a lot of elementary mathematics glued together with a tiny bit of temporal logic

- Unfortunately, the computer science departments in many universities apparently believe that fluency in C++ is more important than a sound education in elementary mathematics. So, some readers may be unfamiliar with the math needed to write specifications.

- If exposure to C++ has not destroyed your ability to think logically, you should have no trouble filling any gaps in your mathematics education

Le Model-Checking (systèmes d'états finis)

- Naissance dans les années 1980, dans un milieu très différent de celui de la preuve formelle :
  - protocoles de communication (les précurseurs)
  - circuits électroniques
  - algorithmes distribués
  - programmes réactifs et temps-réel

- Des idées-clefs développées indépendamment en France et aux USA → Prix Turing 2007
  - J-P. Queille et J. Sifakis* à Grenoble
  - E. Clarke* et E. Emerson* à CMU et U. Texas

- Deux grand principes : l'exploration systématique des exécutions, explicite ou implicite (symbolique), et l'expression des propriétés à prouver en logique temporelle (A. Pnueli*)

- De nombreux systèmes efficaces, dont plusieurs industriels

Agenda

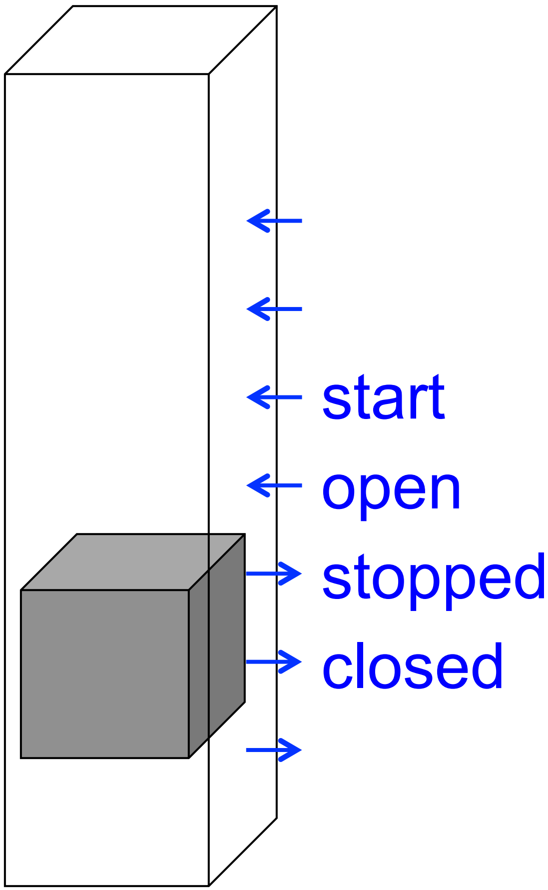
1. Les logiques temporelles
2. Les systèmes de transitions et la bisimulation
3. La vérification par observateurs
4. Les algorithmes de vérification explicite
5. Le Sudoku en calcul booléen

Agenda

1. Les logiques temporelles
2. Les systèmes de transitions et la bisimulation
3. La vérification par observateurs
4. Les algorithmes de vérification explicite
5. Le Sudoku en calcul booléen

Propriétés de sécurité (safety)

L'ascenseur ne peut pas voyager la porte ouverte

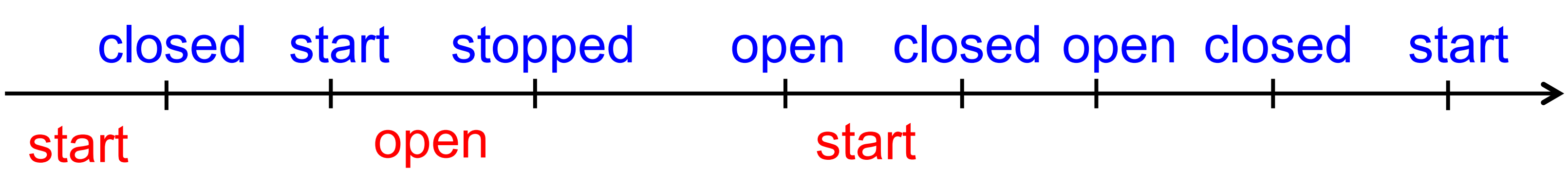


<u>A vérifier</u>

1. Au départ, l'ascenseur ferme sa porte avant de démarrer
2. Après un start, open ne doit jamais arriver jusqu'à stopped
3. Après un open, start ne doit jamais arriver jusqu'à closed

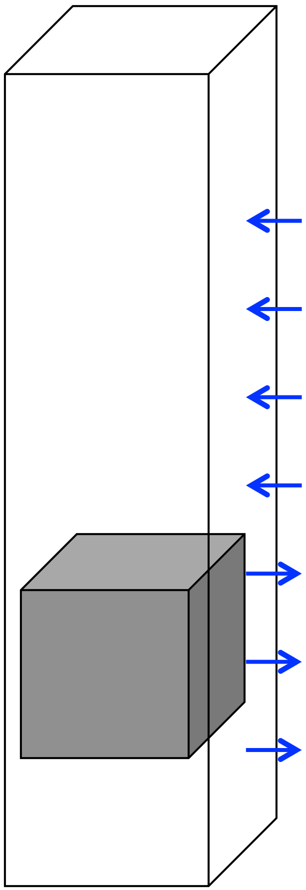
<u>Condition d'environnement</u>

l'ascenseur est initialement arrêté, porte ouverte



Propriétés de vivacité (liveness)

L'ascenseur ramasse les passagers qui l'appellent



<u>A vérifier</u>

Après call[i], l'ascenseur finit par s'arrêter à l'étage i et y ouvre sa porte avant de redémarrer

<u>Conditions d'environnement</u>

Quand il monte (rep. descend), l'ascenseur parcourt tous les étages dans l'ordre de la direction donnée, jusqu'à l'ordre d'arrêt (qui doit nécessairement arriver)

Parler des exécutions

- Structures de données : trois grands choix
  - graphes d'états étiquetés par des prédicats, appelés structures de Kripke
  - graphes d'états-transitions, avec transitions étiquetées
  - graphes mixtes états / transitions étiquetés
- Définition des prédicats d'états / transitions
  - simples symboles non interprétés p, q, etc.
  - ou prédicats sur le contrôle et les données d'un programme
  - ou prédicats sur les horloges d'un système temporisé
- Définition des propriétés à vérifier (sûreté ou vivacité)
  - formules de logique temporelle linéaire ou arborescente
  - formules de μ-calcul
  - propriétés définies par des observateurs
  - équivalences ou raffinement de comportements

Structure de Kripke

$K = (S, I, T, P, L)$

S : ensemble d'états (*states*)

$I \subset S$: ensemble d'états initiaux

$T \subset S \times S$: relation de transition totale à gauche

$\forall s \in S.\ \exists s' \in E.\ sTs'$ $\rightarrow$ chemins infinis pour T

$P = \{p, q, ...\}$: ensemble de prédicats atomiques

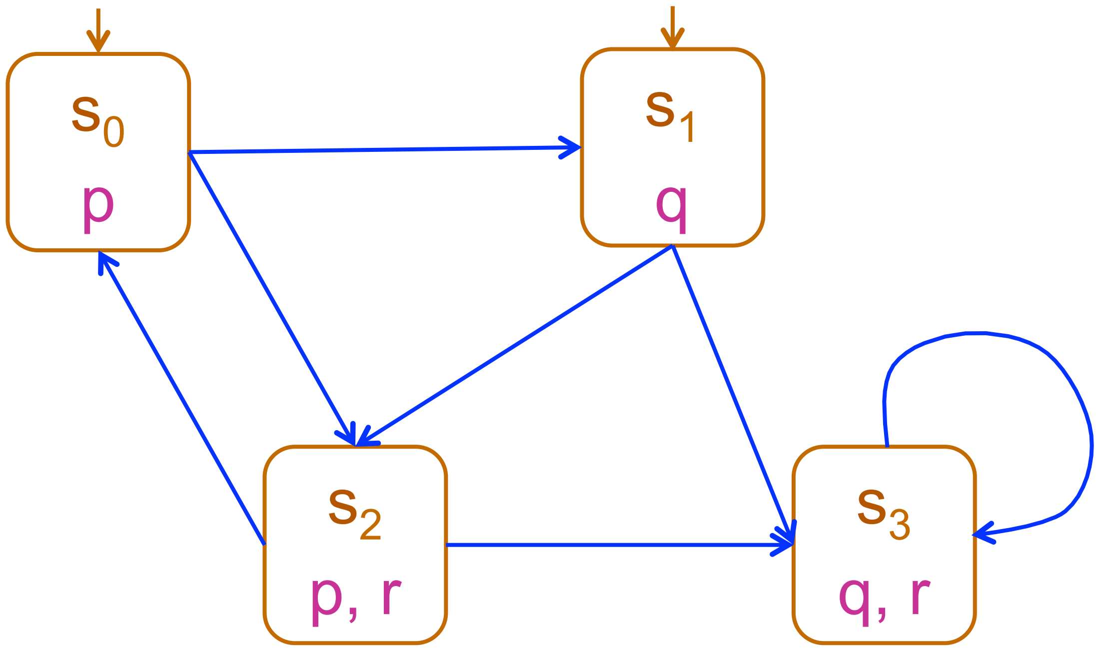
$L : S \rightarrow 2^P$: fonction d'étiquetage

$L(s) = \{p, r, \ldots\}$ ensembles des prédicats vrais en s

chemin : $\pi = s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots$

suffixe : $\pi[n] = s_n \rightarrow s_{n+1} \rightarrow s_{n+2} \rightarrow \ldots$

Exemple de structure de Kripke



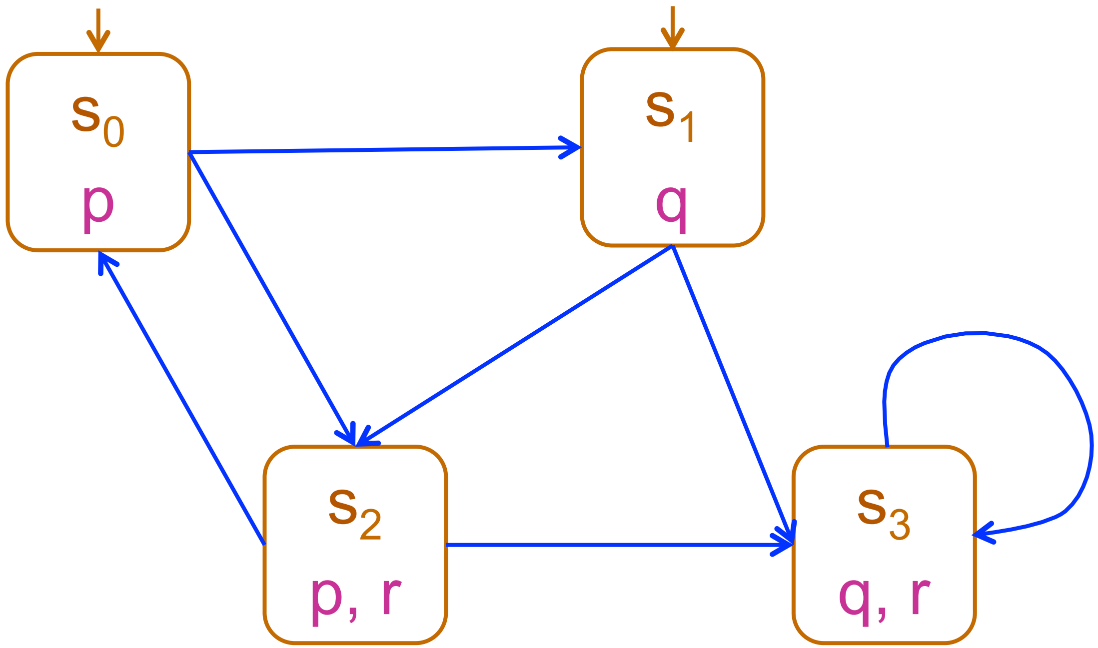
Propriétés satisfaites :

P1 : p et q ne sont jamais vraies en même temps

P2 : tout q est immédiatement suivi d'un r

P3 : tout chemin infini depuis un état initial atteint r

Exemple de structure de Kripke



Propriétés non satisfaites :

P4 : tout p est immédiatement suivi d'un q

P5 : tout chemin infini depuis un état initial atteint p

*La logique temporelle CTL**

Φ : formule d'états

ϕ : formules de chemins

$\Phi := \bot \mid \top \mid p \mid \neg\Phi \mid \Phi \wedge \Phi' \mid \Phi \vee \Phi' \mid \Phi \Rightarrow \Phi' \mid \Phi \Leftrightarrow \Phi'$
$\quad \mid A\phi \mid E\phi$

$\phi := \Phi \mid \neg\phi \mid \phi \wedge \phi' \mid \phi \vee \phi' \mid \phi \Rightarrow \phi' \mid \phi \Leftrightarrow \phi'$
$\quad \mid G\phi \mid F\phi \mid X\phi \mid \phi U \phi'$

- Intuition
  - $A\phi$: tous les chemins partant de l'état donné satisfont ϕ
  - $E\phi$: il existe un chemin partant de l'état donné qui satisfait ϕ
  - Φ : l'état initial du chemin donné satisfait Φ
  - $G\phi$: tous les suffixes du chemin donné satisfont ϕ
  - $F\phi$: il existe un suffixe du chemin donné qui satisfait ϕ
  - $X\phi$, i.e., next ϕ : le premier suffixe du chemin donné satisfait ϕ
  - $\phi U \phi'$: le chemin donné satisfait ϕ jusqu'à satisfaire ϕ', ce qu'il doit faire

*La logique temporelle CTL**

Φ : formule d'états

ϕ : formules de chemins

Φ := $\bot \mid \top \mid p \mid \neg\Phi \mid \Phi\wedge\Phi' \mid \Phi\vee\Phi' \mid \Phi\Rightarrow\Phi' \mid \Phi\Leftrightarrow\Phi'$
$\mid A\phi \mid E\phi$

ϕ := $\Phi \mid \neg\phi \mid \phi\wedge\phi' \mid \phi\vee\phi' \mid \phi\Rightarrow\phi' \mid \phi\Leftrightarrow\phi'$
$\mid G\phi \mid F\phi \mid X\phi \mid \phi U \phi'$

Interprétation $K,s \vDash \Phi$ ou simplement $s \vDash \Phi$ pour une structure de Kripke K et un état s

$s \vDash p$: le prédicat p est vrai en s

$s \vDash \neg\Phi$: $s \nvDash \Phi$, $s \vDash \Phi\wedge\Phi'$: $s \vDash \Phi$ et $s \vDash \Phi'$, etc.

$s \vDash A\ \phi$: tous les chemins partant de s vérifient ϕ

$s \vDash E\ \phi$: il existe un chemin partant de s qui vérifie ϕ

*La logique temporelle CTL**

Φ : formule d'états

ϕ : formules de chemins

$\Phi := \bot \mid \top \mid p \mid \neg\Phi \mid \Phi\wedge\Phi' \mid \Phi\vee\Phi' \mid \Phi\Rightarrow\Phi' \mid \Phi\Leftrightarrow\Phi'$
$\mid A\phi \mid E\phi$

$\phi := \Phi \mid \neg\phi \mid \phi\wedge\phi' \mid \phi\vee\phi' \mid \phi\Rightarrow\phi' \mid \phi\Leftrightarrow\phi'$
$\mid G\phi \mid F\phi \mid X\phi \mid \phi U \phi'$

<u>Interprétation</u> $\pi \vDash \phi$ pour K et un chemin $\pi = s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots$

$\pi \vDash \Phi$: $s_0 \vDash \Phi$,

$\pi \vDash \neg\phi$: $\pi \nvDash \phi$, $\pi \vDash \phi\wedge\phi'$: $\pi \vDash \phi$ et $\pi \vDash \phi'$, etc.

$\pi \vDash G\,\phi$: ssi $\forall n.\ \pi[n] \vDash \phi$, avec $\pi[n] = s_n \rightarrow s_{n+1} \rightarrow \ldots$

$\pi \vDash F\,\phi$: ssi $\exists n.\ \pi[n] \vDash \phi$

$\pi \vDash X\,\phi$: ssi $\pi[1] \vDash \phi$

$\pi \vDash \phi\ U\ \phi'$ ssi $\exists n.\ (\forall m<n.\ \pi[m] \vDash \phi) \wedge \pi[n] \vDash \phi'$

*La logique temporelle CTL**

Φ : formule d'états

ϕ : formules de chemins

$\Phi := \bot \mid \top \mid p \mid \neg\Phi \mid \Phi\wedge\Phi' \mid \Phi\vee\Phi' \mid \Phi\Rightarrow\Phi' \mid \Phi\Leftrightarrow\Phi'$
$\quad \mid A\phi \mid E\phi$

$\phi := \Phi \mid \neg\phi \mid \phi\wedge\phi' \mid \phi\vee\phi' \mid \phi\Rightarrow\phi' \mid \phi\Leftrightarrow\phi'$
$\quad \mid G\phi \mid F\phi \mid X\phi \mid \phi U \phi'$

nommage et notations équivalentes :

pout tout chemin : $\Box\phi \equiv A\phi$

il existe un chemin : $\Diamond\phi \equiv E\phi$

pour tout suffixe : $\Box\phi \equiv G\phi$

il existe un suffixe : $\Diamond\phi \equiv F\phi$

prochain suffixe (next) : $\bigcirc\phi \equiv X\phi$

*Complexité de CTL**

- CTL* est très riche, car les modalités peuvent se combiner arbitrairement
- Complexité théorique :
  - satisfiabilité : pour une formule, existe-t-il une structure de Kripke K la satisfaisant ? 2-ExpTime-complet ☹
  - vérification : étant donné K, s et Φ, quel est le coût de la vérification de K,s $\models \Phi$?

    Linéaire en K, PSpace-complet en Φ
- Rappel : pour taille n
  - linéaire en n : coût en $O(n)$
  - polynomial en n : coût en $O(n^k)$
  - PSpace-complet : mémoire en $O(n^k)$ temps supposé en $O(k^n)$
  - 2ExpTimeComplete : temps en $O(2^{2^n})$

La logique LTL : Linear Temporal Logic

- Formules seulement sur les chemins, formules d'états réduites aux prédicats p

$$\phi := p \mid \neg\phi \mid \phi\wedge\phi' \mid \phi\vee\phi' \mid \phi\Rightarrow\phi' \mid \phi\Leftrightarrow\phi'$$
$$\mid \mathrm{G}\,\phi \mid \mathrm{F}\,\phi \mid \mathrm{X}\,\phi \mid \phi\,\mathrm{U}\,\phi'$$

$$\phi := \ldots$$
$$\mid \square\,\phi \mid \lozenge\,\phi \mid \bigcirc\,\phi \mid \phi\,\mathrm{U}\,\phi$$

- Interprétation comme en CTL*, sur chaque chemin
- Limitation : pas de prédicats d'états non-triviaux,
  donc pas de quantification sur les chemins

Avantages : simple, intuitive, applicabilité assez générale
algorithmes raisonnablement efficaces
Vérifieurs : SPIN, MurΦ, Cospan, SMV, NuSMV, etc.

L'ascenseur en LTL

- De l'état initial, l'ascenseur doit fermer sa porte avant de partir

  $\square(\text{init} \Rightarrow (\neg \text{open}\ \mathbf{U}\ \text{start})$

- Après start, jamais open jusqu'à stopped

  $\square(\text{start} \Rightarrow \bigcirc(\neg \text{open}\ \mathbf{U}\ \text{stopped}))$

- Après open, jamais start jusqu'à closed

  $\square(\text{open} \Rightarrow \bigcirc(\neg \text{start}\ \mathbf{U}\ \text{closed}))$

- Vivacité : l'ascenseur ramasse ses passagers: il est toujours vrai que s'il est appelé à l'étage i, l'ascenseur finira par s'arrêter à cet étage et y ouvrira sa porte avant de repartir

  $\square(\text{call}[i] \Rightarrow \Diamond(\text{stopped}[i] \wedge (\neg \text{start}\ \mathbf{U}\ \text{open}))$

L'algorithme de Dekker

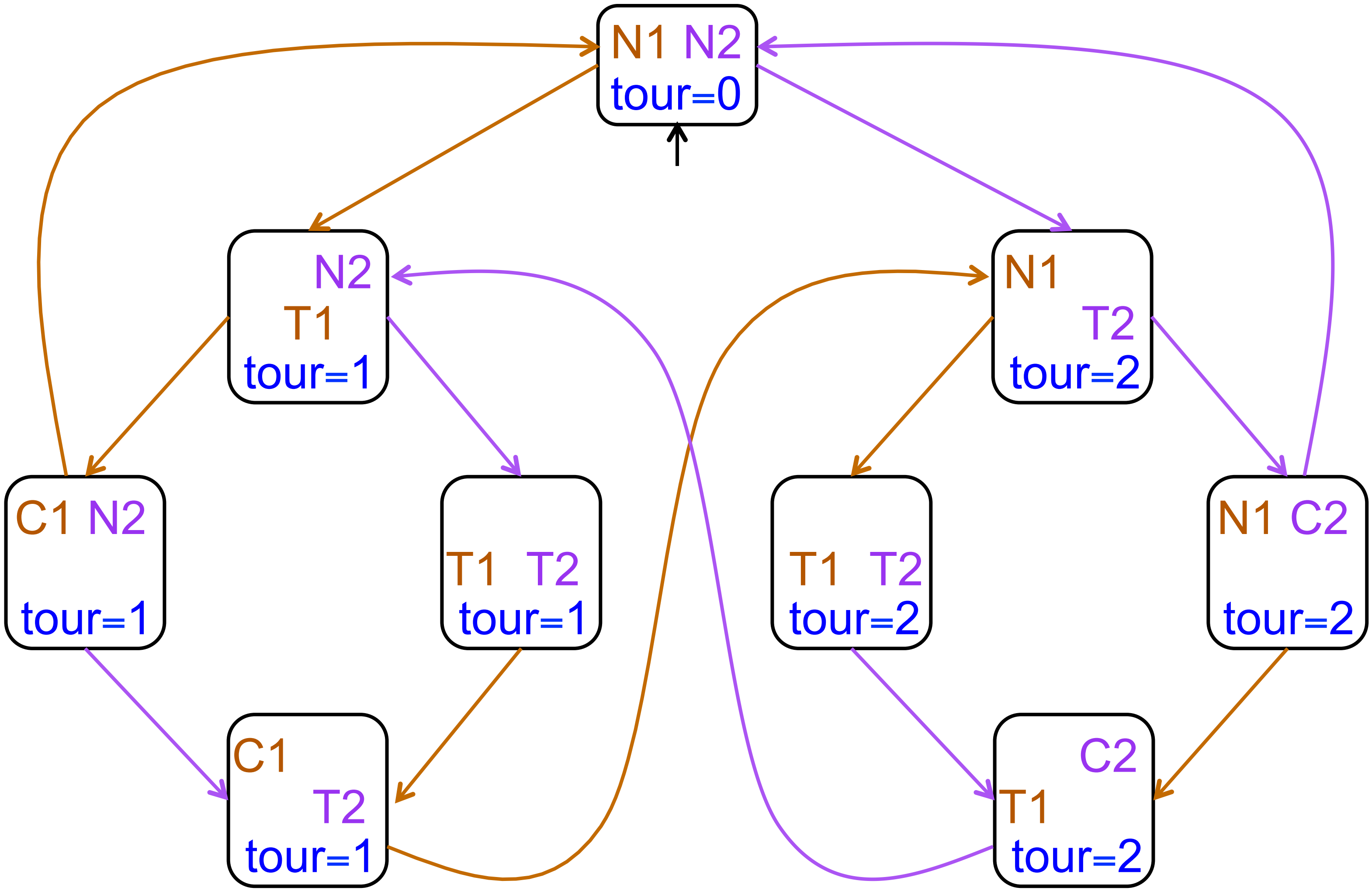
Deux processus J1 et J2 essaient de prendre une ressource R. L'algorithme de Dekker donne un arbitrage équitable:

1. J1 et J2 ne sont jamais ensemble dans leur section critique
2. Chacun des deux obtient une infinité de fois la section critique s'il la demande une infinité de fois

N1 : J1 est dans une section non critique
T1 : J1 essaie d'entrer en section critique
C1 : J1 est en section critique
T2 : visible en lecture seule

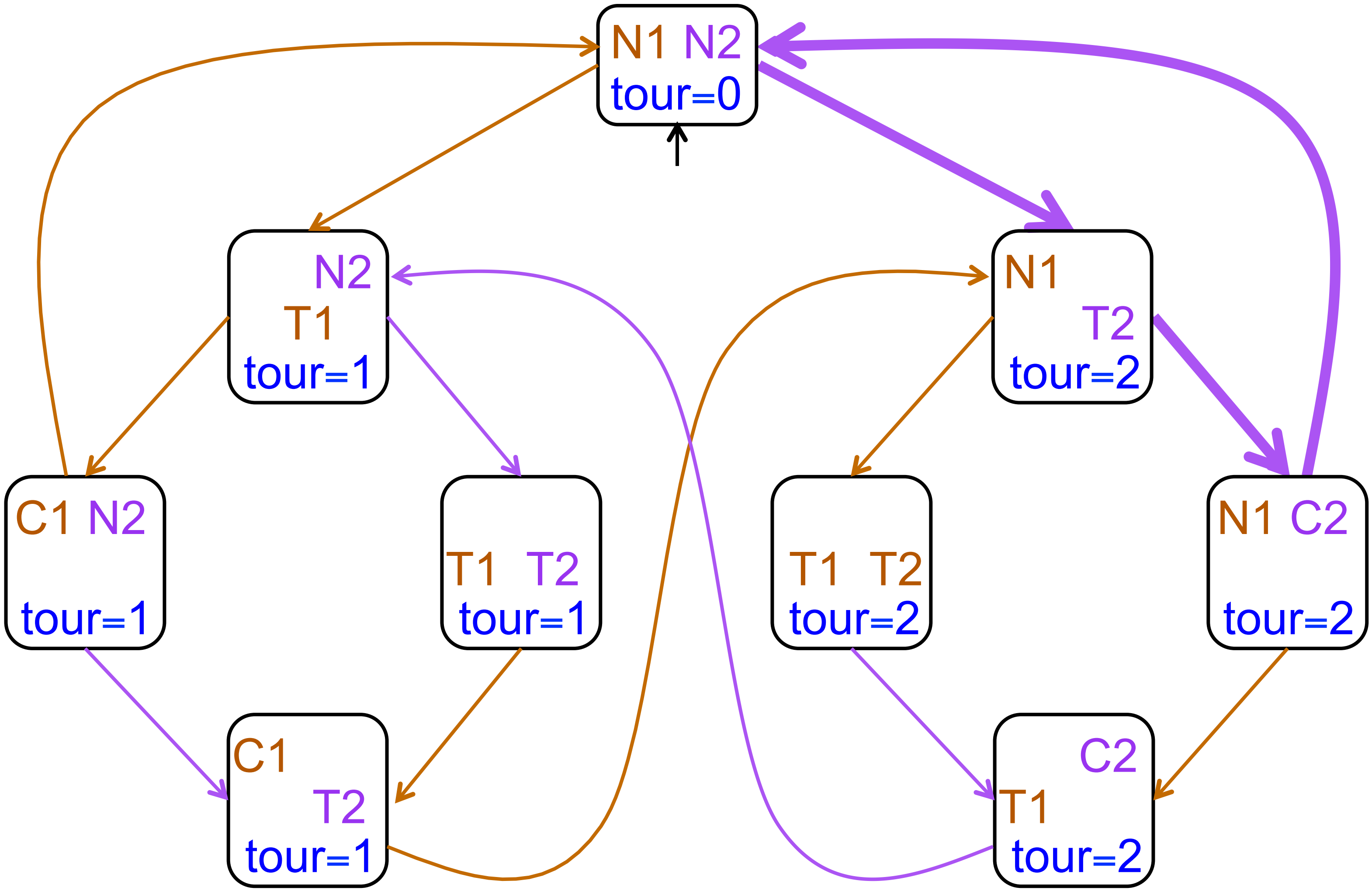
N2 : J2 est dans une section non critique
T2 : J2 essaie d'entrer en section critique
C2 : J2 est en section critique
T1 : visible en lecture seule

tour = 0, 1, 2 : variable partagée en lecture / écriture

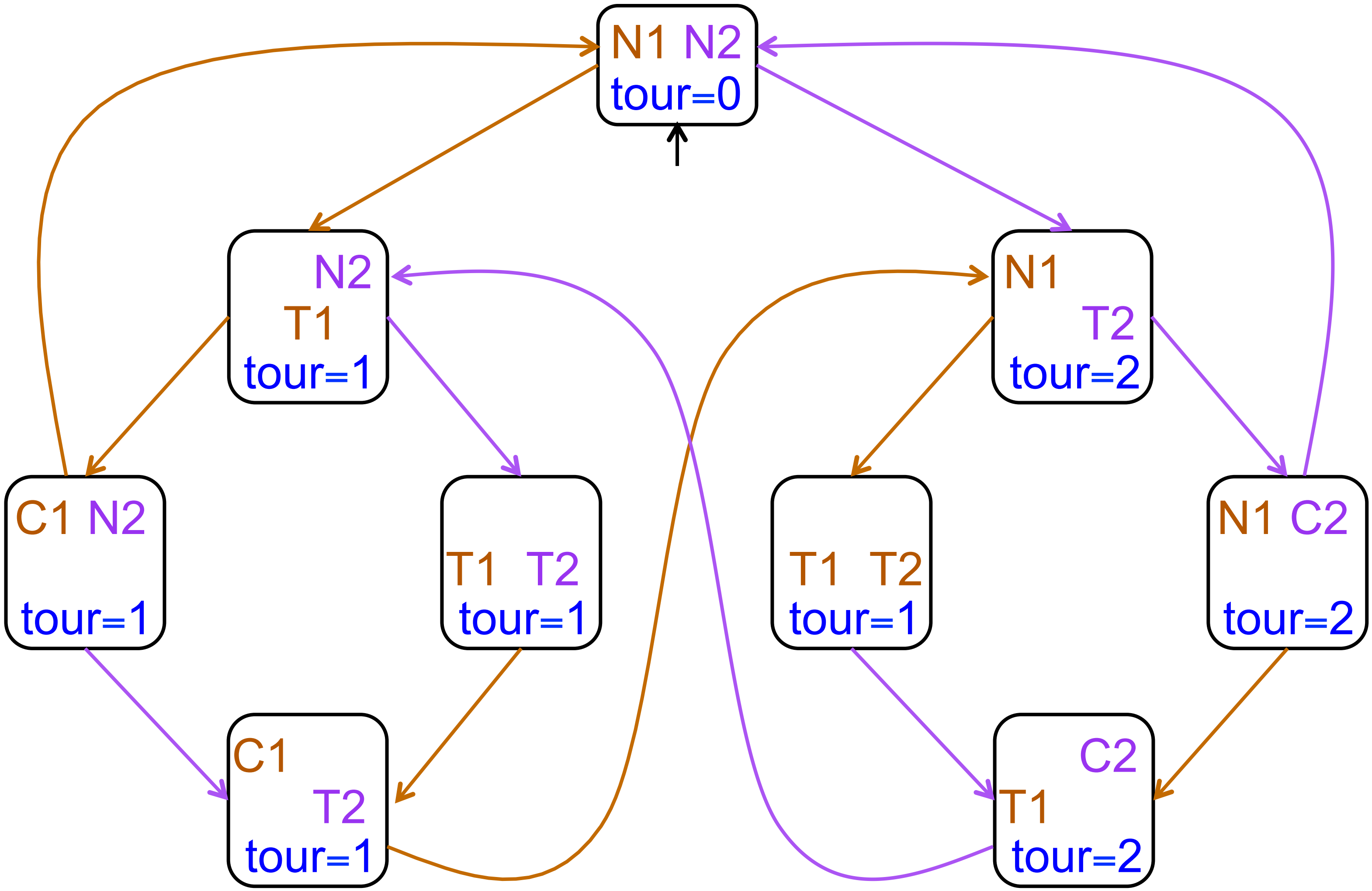


$\Box\neg(C1 \wedge C2)$: Pas de conflit de section critique

VRAI : simple vérification dans chaque état



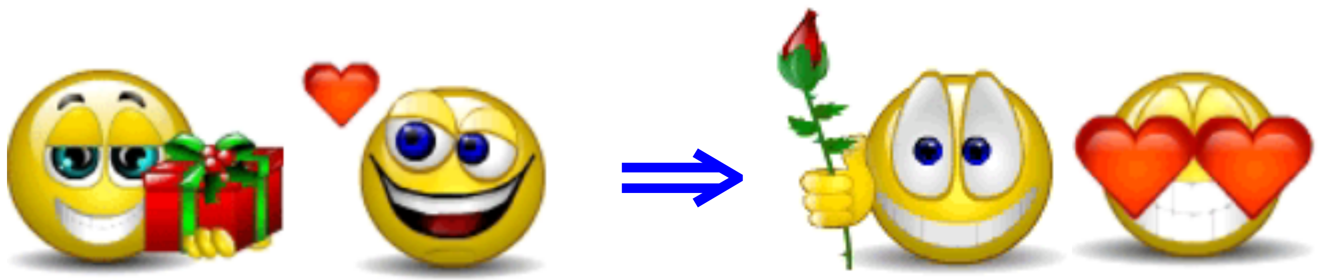
□◇C1 : J1 entre infiniment souvent en section critique
FAUX à cause de la boucle en haut à droite



$\square(\Diamond T1 \Rightarrow \Diamond C1)$: J1 entre infiniment souvent en section critique s'il essaie infiniment souvent

Strong fairness : VRAI mais bien plus subtil !

Amour un jour, amour toujours



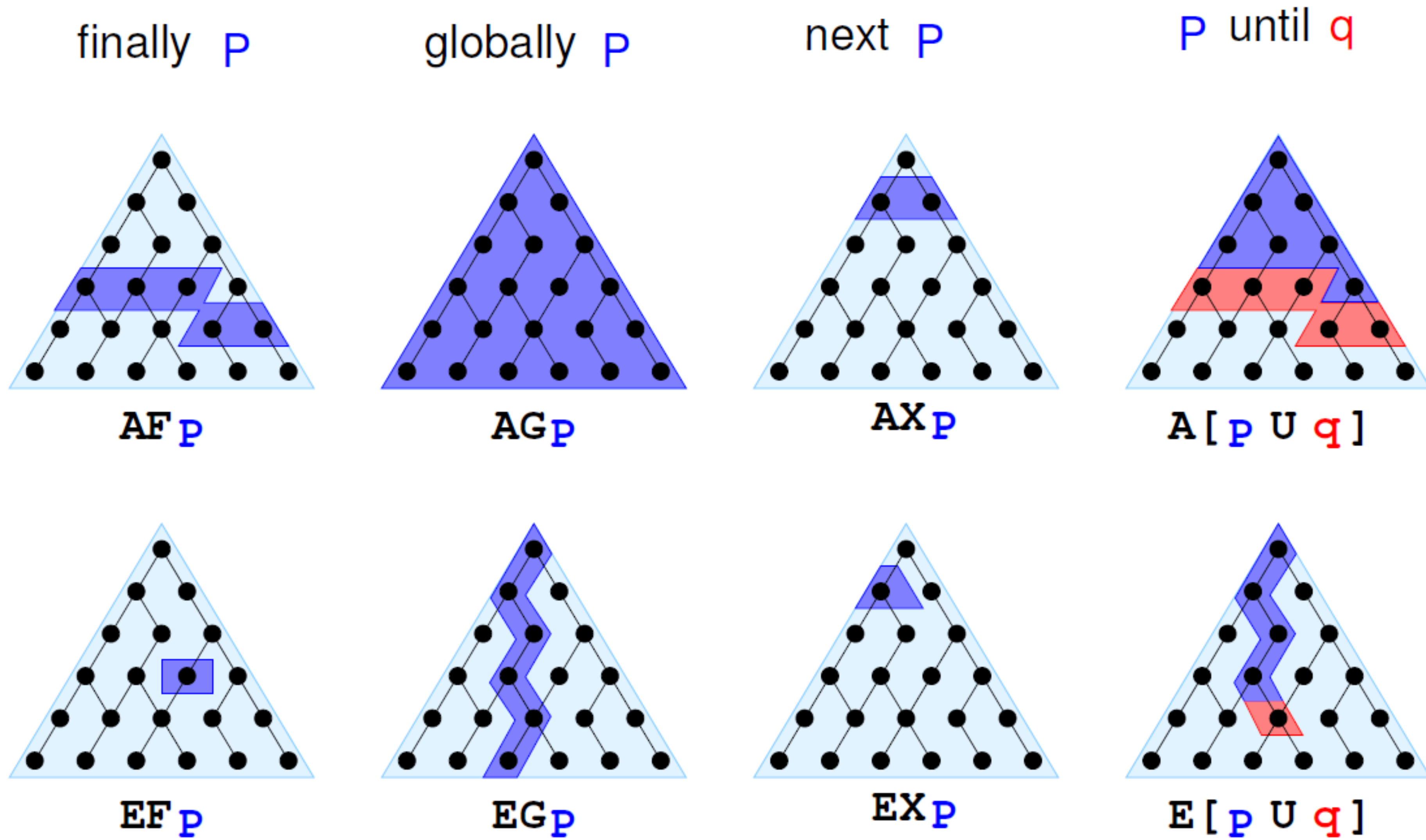
~~(◊ amour) ⇒ (� amour)~~

(◊ amour) ⇒ (◊ � amour)

La logique CTL : Computation Tree Logic

- CTL = la logique initiale d'E. Clarke et E. Emerson
- Modalités acceptées : toute sous-formule de chemin G, F, X ou U est immédiatement précédée de A ou E.
- Depuis un état donné :
  - AGϕ : tous les chemins satisfont toujours ϕ
  - AFϕ : tous les chemins satisfont un jour ϕ
  - AXϕ : tous les premiers suffixes des chemins satisfont ϕ
  - A(ϕUϕ') : tous les chemins satisfont ϕ jusqu'à satisfaire ϕ'

  - EGϕ : il existe un chemin satisfaisant toujours ϕ
  - EXϕ : il existe un chemin dont le premier suffixe satisfait ϕ
  - EFϕ : il existe un chemin satisfaisant un jour ϕ
  - E(ϕUϕ') : il existe un chemin satisfaisant ϕ jusqu'à satisfaire ϕ'



source Alessandro Artale, http://www.inf.unibz.it/~artale/

Propriétés exprimables en LTL et CTL

- Sur le protocole de Dekker

  pas de conflit de section critique (vrai)

  LTL : $\square \neg (C1 \wedge C2)$ → CTL : $AG \neg (C1 \wedge C2)$

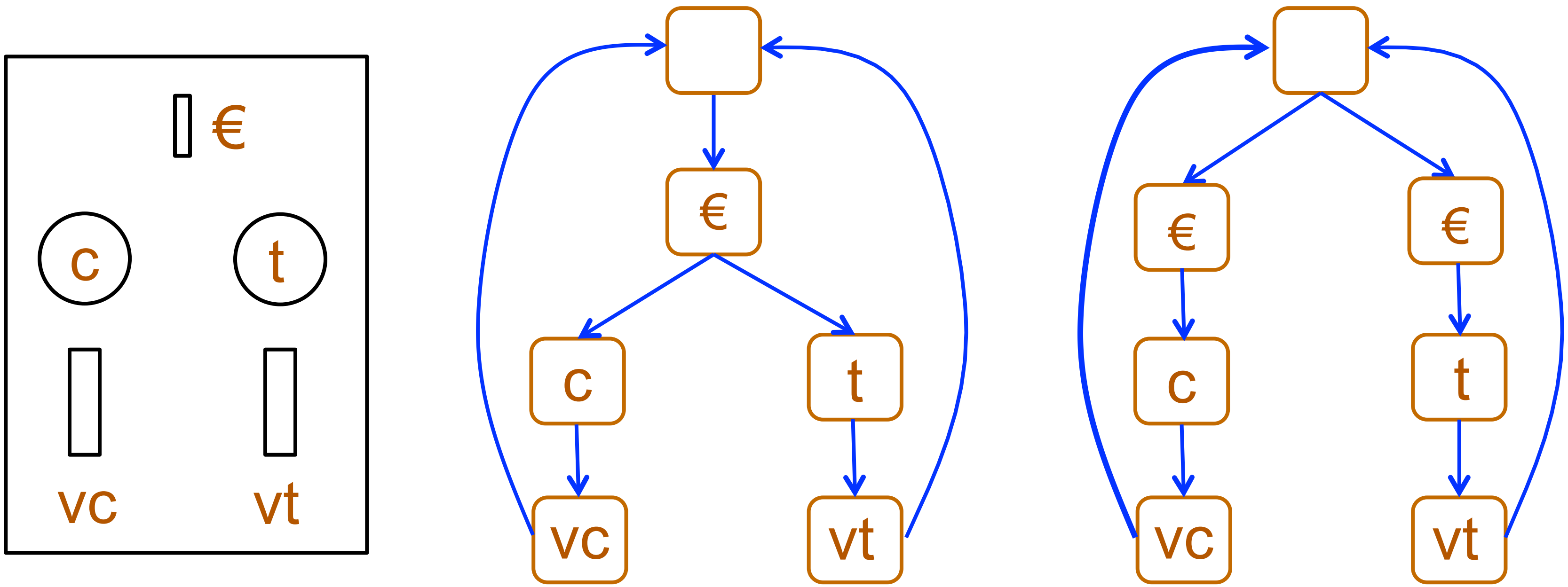
  J1 entre infiniment souvent en section critique (faux)

  LTL : $\square \diamond C1$ → CTL : $AF\ C1$

  J1 finit par entrer en section critique s'il essaie (vrai)

  LTL : $\square (\diamond T1 \Rightarrow \diamond C1)$ → $AF\ (T1 \Rightarrow EF\ C1)$

Les deux machines à café de R. Milner



Indistingables en LTL car les traces linéaires sont identiques :
(. → € → c → vc →)* et (. → € → t → vt →)*

Distingables en CTL, par AG(€ ⇒ E(c U €)) :
après avoir mis un €, je peux toujours avoir un café sans remettre un autre €

Exprimables soit en LTL soit en CTL

- Exprimables en LTL mais pas en CTL (strong fairness)

  dans chaque exécution, J1 entre infiniment souvent en section critique s'il essaie infiniment souvent : $\Box(\Diamond T1 \Rightarrow \Diamond C1)$

- Exprimables en CTL mais pas en LTL, qui n'a pas EF

  Quand J1 est en section non-critique, il existe toujours un chemin où il entre en section critique $AG(T1 \Rightarrow EF\ C1)$:

Le choix de la logique dépend de la propriété à exprimer
Certains moteurs gèrent une seule logique, d'autres les deux

Tout reste exprimable en CTL*, mais de toutes façons, les formules compliquées sont vite incompréhensibles...

Agenda

1. Les logiques temporelles
2. Les systèmes de transitions et la bisimulation
3. La vérification par observateurs
4. Les algorithmes de vérification explicite
5. Le Sudoku en calcul booléen

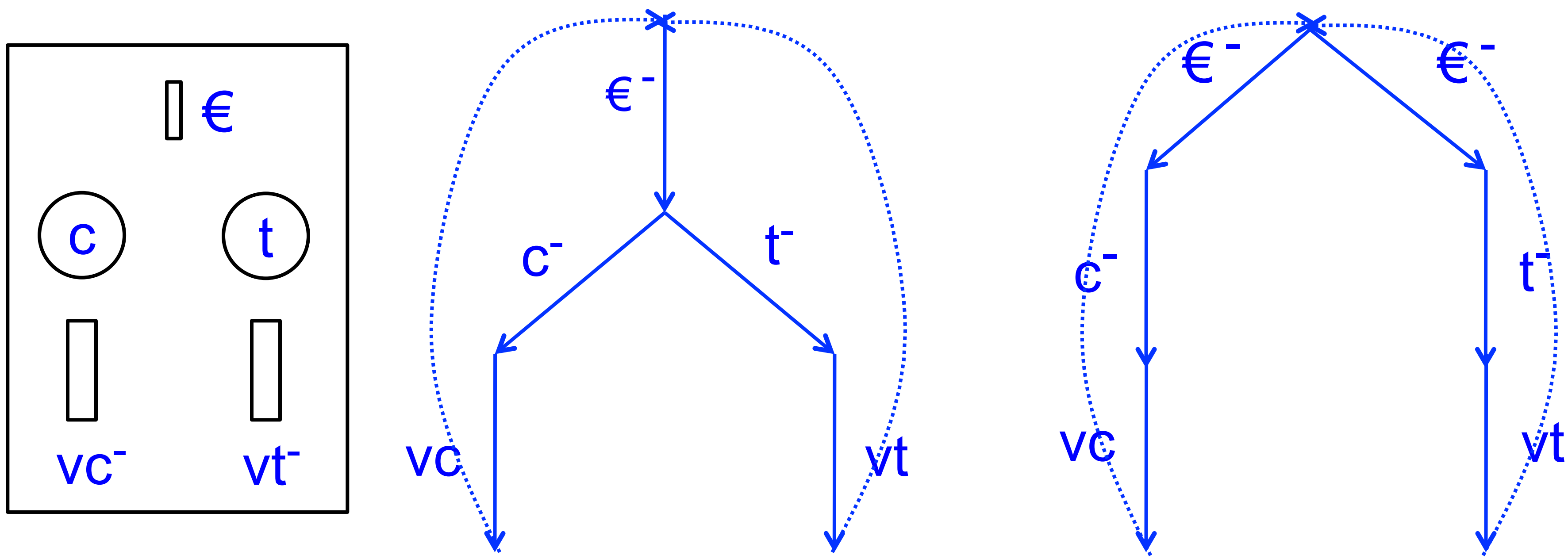
Les systèmes de transitions

- Les structures de Kripke conviennent pour des programmes fermés, mais moins pour des systèmes à entrées-sorties et des processus communicants
- Alternative : les systèmes de transition, où les prédicats sont portés par les transitions et non par les états (quelquefois par les deux)

  S : états ou processus
  T : transitions
  A : actions
  $L : T \rightarrow 2^A$:étiquetage des transitions par des actions

Beaucoup de liberté dans la définition:
composants finis ou infinis, structure algébrique de A, etc.

Les deux machines à café



$$p = €^{-}.(c^{-}.vc + t^{-}.vt) \qquad q = €^{-}.c^{-}.vc + €^{-}.t^{-}.vt$$

CCS = Calculus of Communicating Systems
R. Milner, 1994

Actions :

atomes a, b, c, ..., plus action invisible τ

inverses $\bar{a}$, avec $\bar{\bar{a}} = a$

notation : $\alpha \rightarrow$ a, $\bar{a}$ et $\mu \rightarrow$ a, $\bar{a}$, τ

Etats : termes p, q, ..., définis pas la grammaire suivante

0	: état inerte
μ.p	: action puis continuation
p+q	: choix entre p et q
p|q	: mise en parallèle de p et q
p\a	: déclaration locale de a et $\bar{a}$ dans p
x, rec x.p(x)	: id. de processus, appel récursif

Sémantique SOS de CCS

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

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

$$\frac{p \xrightarrow{\mu} p'}{p|q \xrightarrow{\mu} p'|q} \qquad \frac{q \xrightarrow{\mu} q'}{p|q \xrightarrow{\mu} p|q'} \qquad \boxed{\frac{p \xrightarrow{a} p' \quad q \xrightarrow{\bar{a}} q'}{p|q \xrightarrow{\tau} p'|q'}}$$

$$\boxed{\frac{p \xrightarrow{\mu} p' \quad \mu \neq a, \bar{a}}{p \backslash a \xrightarrow{\mu} p' \backslash a}} \qquad \frac{p \xrightarrow{\mu} p'}{\mathrm{rec}\, x = p \xrightarrow{\mu} p' [x \leftarrow \mathrm{rec}\, x = p]}$$

Idée centrale : communication = $a.\bar{a} \rightarrow \tau$, globale ou locale

La bisimulation forte

- Comment dire que deux systèmes non-déterministes sont équivalents, i.e., indistingables par communication ?
- Une bisimulation est une relation entre états vérifiant

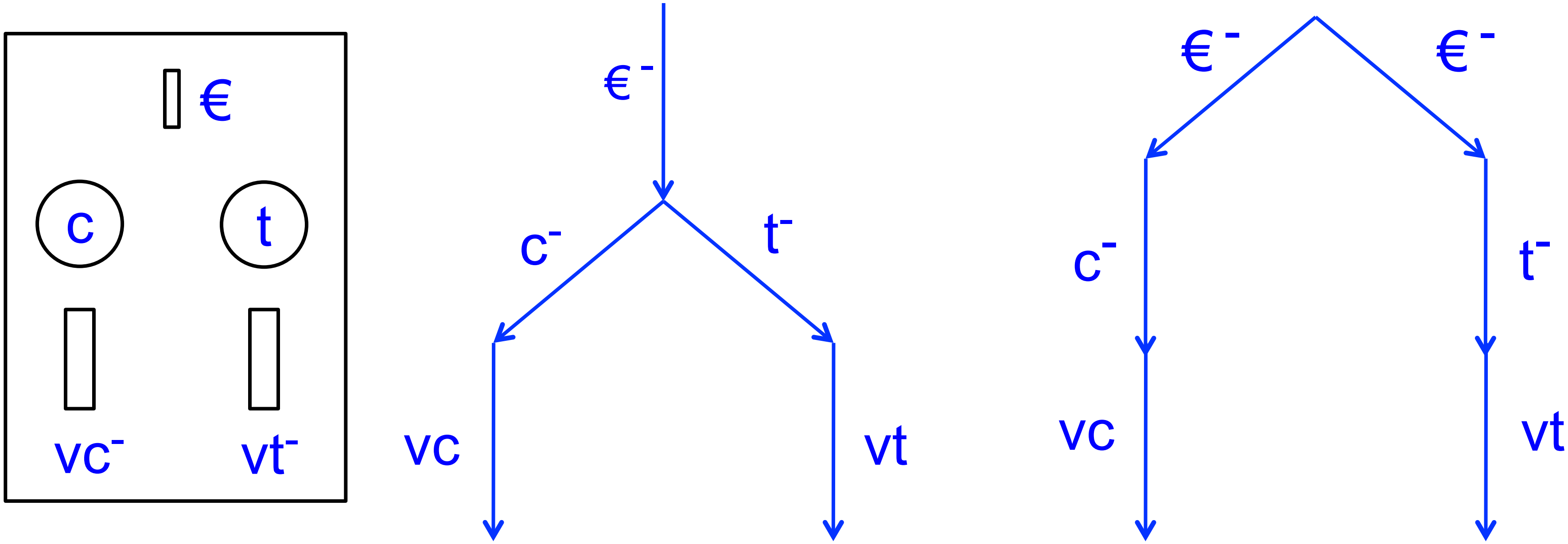
> si p et q sont bisimilaires,
> si p peut faire α, alors q peut le faire aussi
> en restant bisimilaire, et réciproquement

$$\text{si } p\,R\,q \text{ et } p \xrightarrow{\alpha} p', \text{ alors } \exists q'.\ q \xrightarrow{\alpha} q' \text{ et } p'\ R\ q'$$

$$\text{si } p\,R\,q \text{ et } q \xrightarrow{\alpha} q', \text{ alors } \exists p'.\ p \xrightarrow{\alpha} p' \text{ et } p'\ R\ q'$$

- La bisimulation forte est la plus grande bisimulation (beaucoup de caractérisations possibles).
- c'est une équivalence et une congruence pour toutes les opérations

Les deux machines à café



$p = €^{-}.(c^{-}.vc + t^{-}.vt)$ $\qquad$ $q = €^{-}.c^{-}.vc + €^{-}.t^{-}.vt$

p et q ne sont pas bisimilaires :

$p \xrightarrow{€^{-}} (c^{-}.vc + t^{-}.vt) \xrightarrow{t} vc$ $\qquad$ $q \xrightarrow{€^{-}} t^{-}.vt \not\xrightarrow{c}$

La bisimulation faible, ou équivalence observationnelle

- Idée : même définition, mais en tenant compte des chaînes d'actions internes τ, mais pas de leur longueur

$$p \overset{a}{\Longrightarrow\!\!\bullet}\ p' \quad \text{ssi} \quad p \xrightarrow{\tau} p_1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} p_n \xrightarrow{a} p'_1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} p'$$

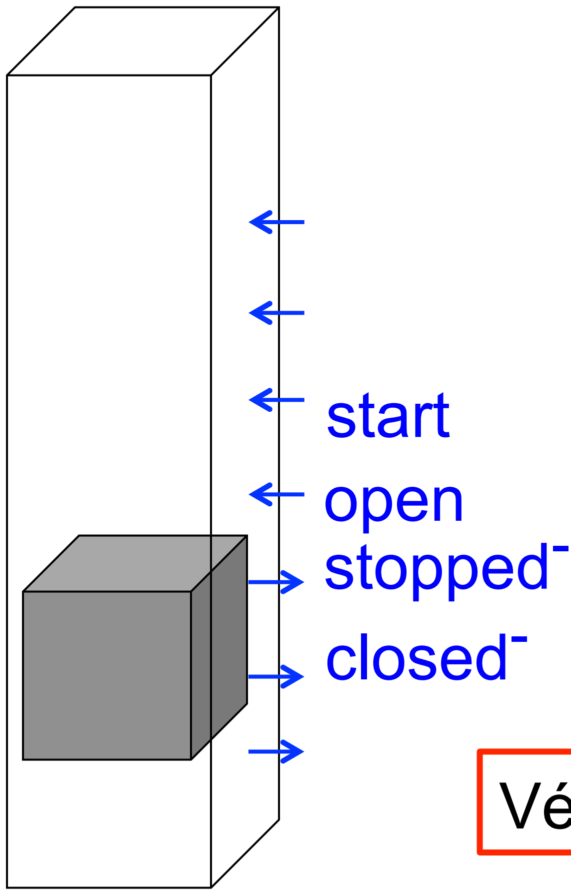
- p et q sont observationnellement équivalents s'ils sont fortement bisimilaires pour $\Longrightarrow\!\!\bullet$

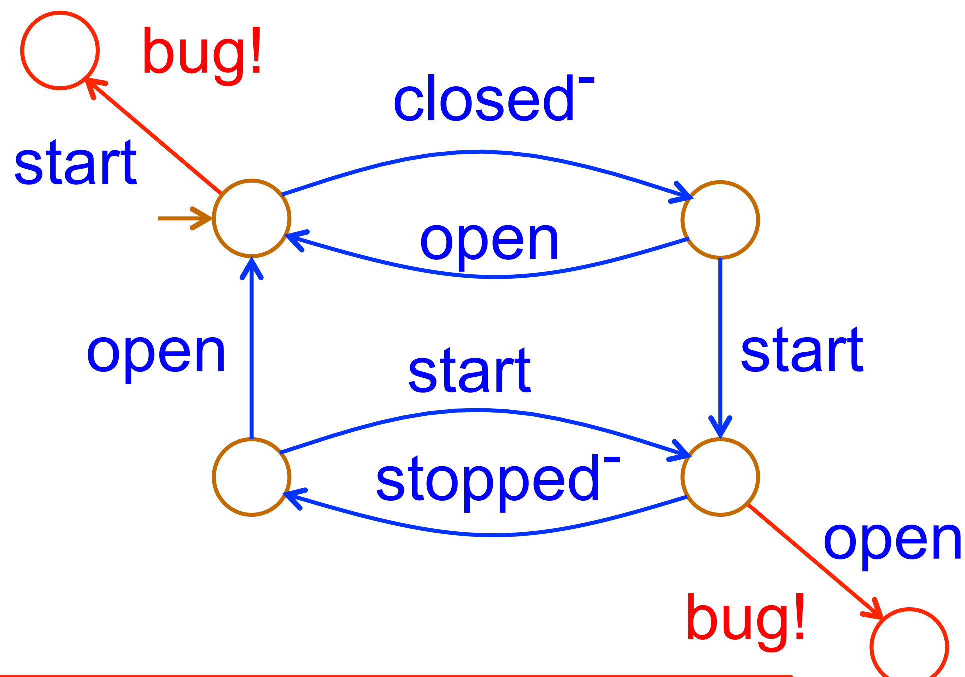
- Attention : l'équivalence observationnelle n'est pas une congruence !

Réduction par équivalence observationnelle

L'ascenseur ne peut pas voyager la porte ouverte

→ réduire le terme ascenseur \ {s_1, s_2,...,s_n} où les s_i cachés sont tous les signaux sauf open, start, closed⁻ et stopped⁻



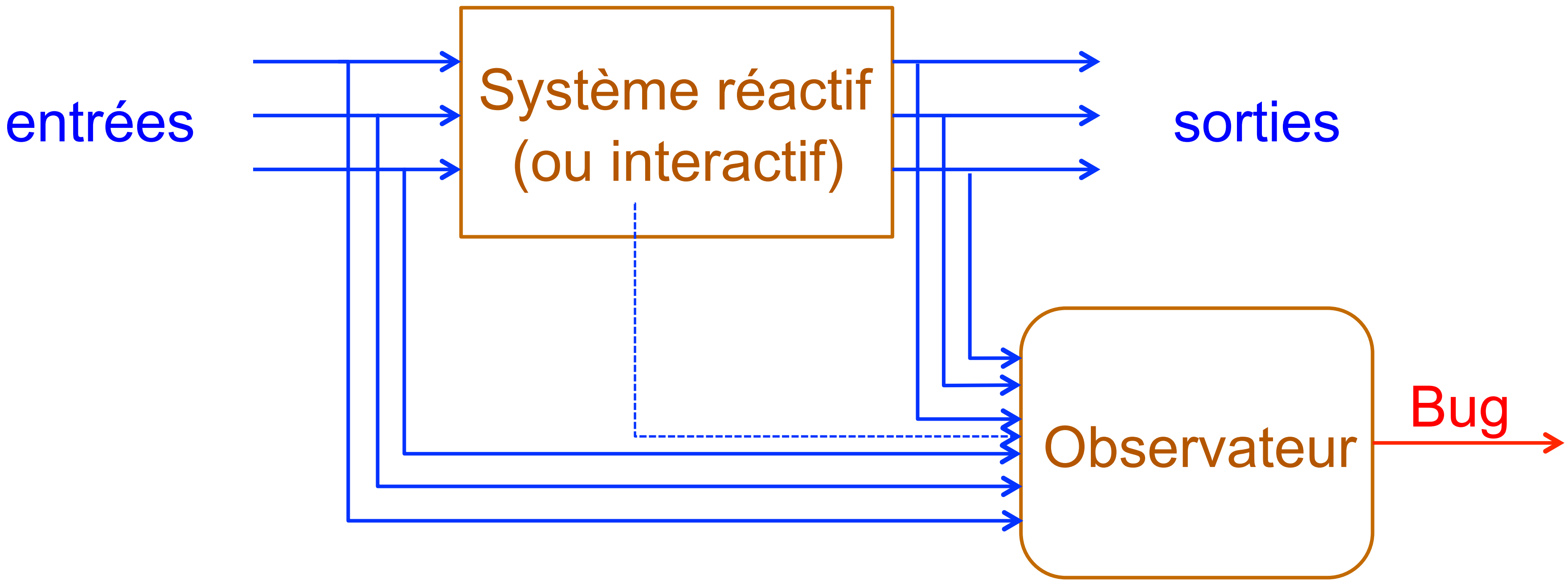


Vérifier = inspecter la réduction

Agenda

1. Les logiques temporelles
2. Les systèmes de transitions et la bisimulation
3. La vérification par observateurs
4. Les algorithmes de vérification explicite
5. Le Sudoku en calcul booléen

Vérification par observateur synchrone



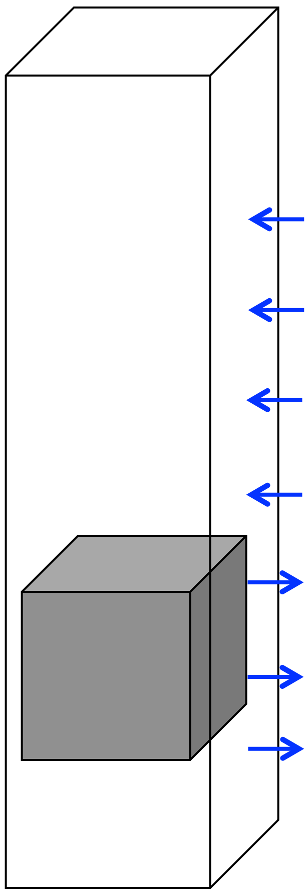
A prouver: Bug ne peut jamais être émis

L'observateur étant lui-même un système réactif, il peut être programmé dans le même langage (Esterel, Lustre, etc.)

Souvent plus simple que la logique temporelle.

Observateur Esterel

L'ascenseur ne peut pas voyager la porte ouverte

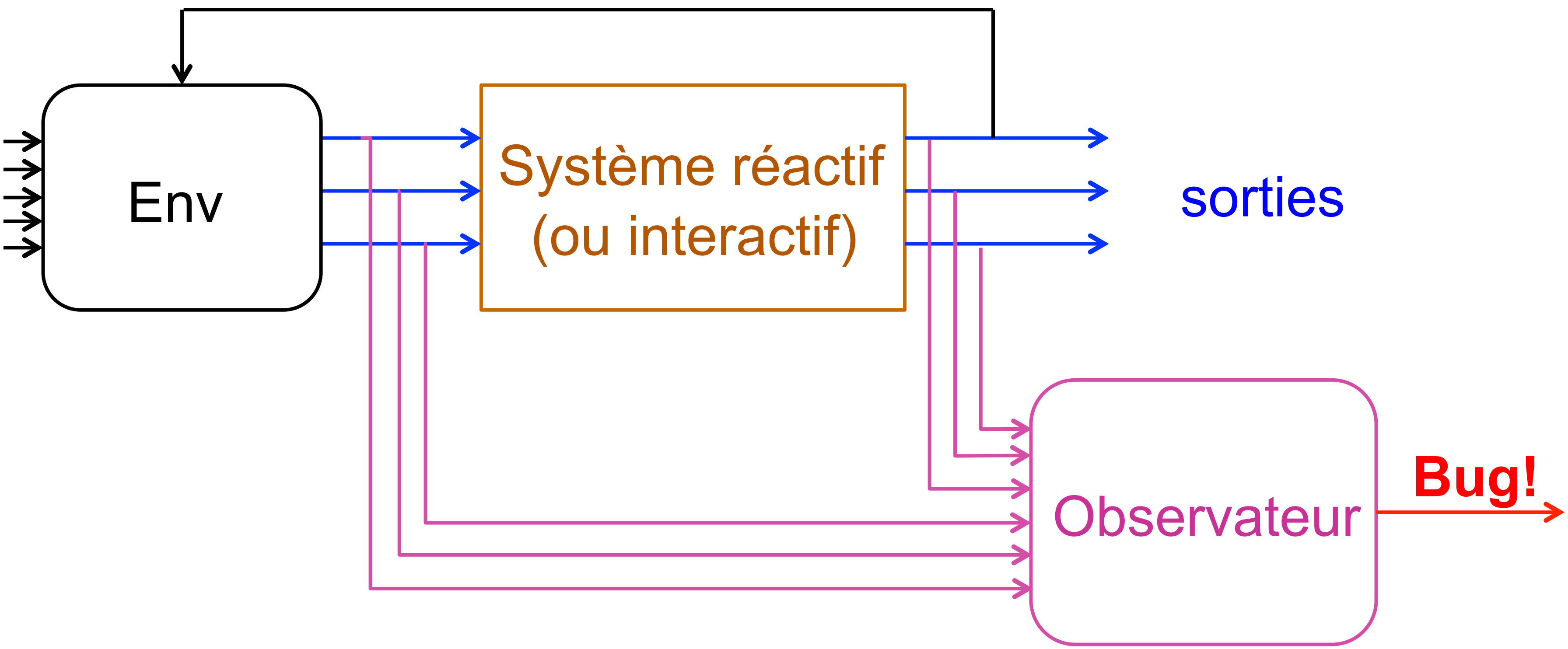



```
signal danger in
   loop
      abort sustain danger when closed ;
      await open
   end loop
||
   loop
      await start ;
      abort
         await immediate danger do emit Bug end
      when stopped
   end loop
end signal
```

A prouver : Bug ne peut jamais être émis

Modéliser l'environnement

- L'environnement dans lequel évolue le programme n'est pas arbitraire, et il est souvent indispensable de le modéliser
  - sinon les propriétés peuvent devenir fausses
  - et pour minimiser la taille de l'espace d'états



Modéliser l'environnement

- Exemple pour l'ascenseur
  - il faut qu'une porte soit ouverte ou fermée
  - pour aller du 2^{e} au 4^{e}, il faut passer par le 3^{e}
  - il n'y a pas de bouton pour aller vers le haut au dernier étage

- Attention :
  - en logique temporelle, il faut prouver Env ⇒ Props
  - il faut s'assurer que le modèle de Env est non-vide, sinon Env ⇒ Props devient trivial !

  ⇒ problème de satisfiabilité des formules de logique temporelle

Agenda

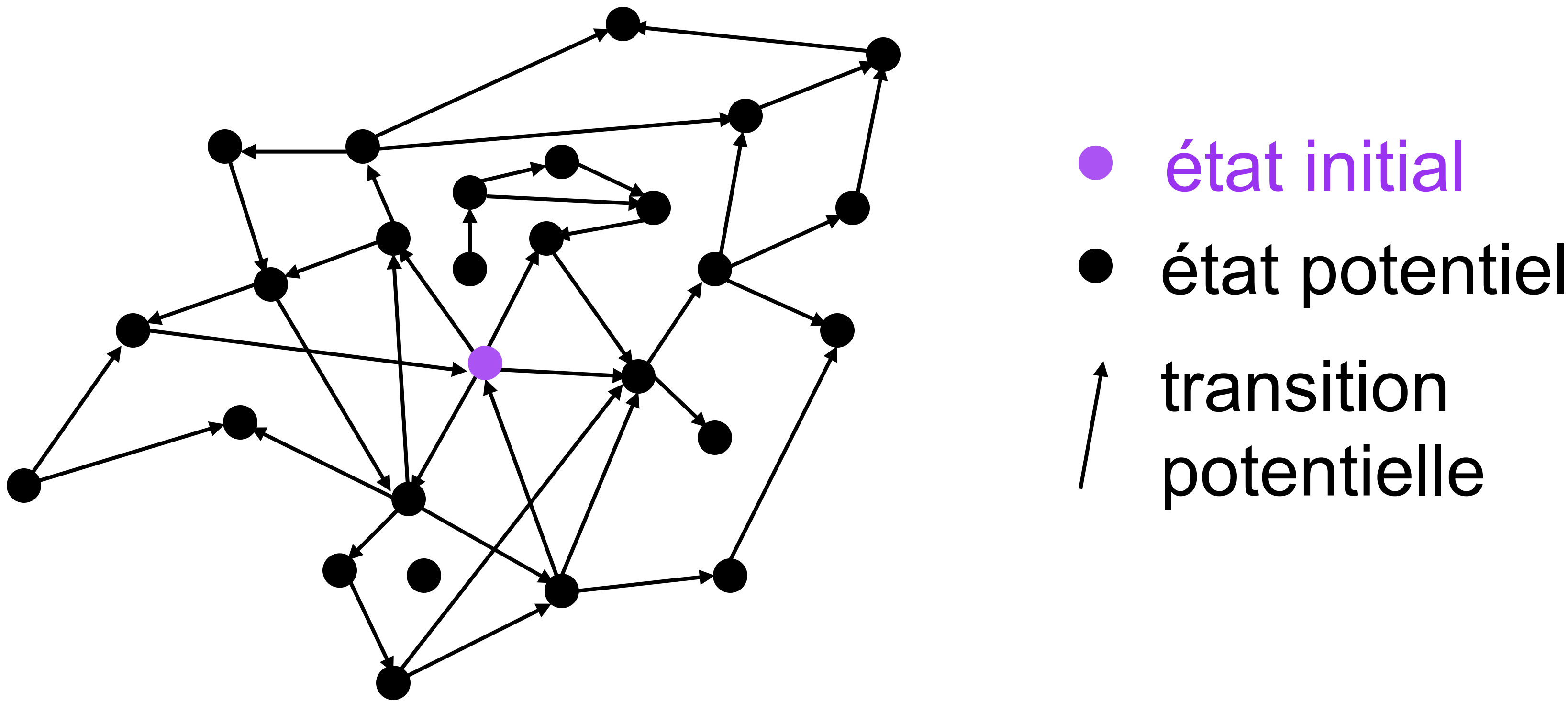
1. Les logiques temporelles
2. Les systèmes de transitions et la bisimulation
3. La vérification par observateurs
4. Les algorithmes de vérification explicite
5. Le Sudoku en calcul booléen

L'algorithmique du Model-Checking

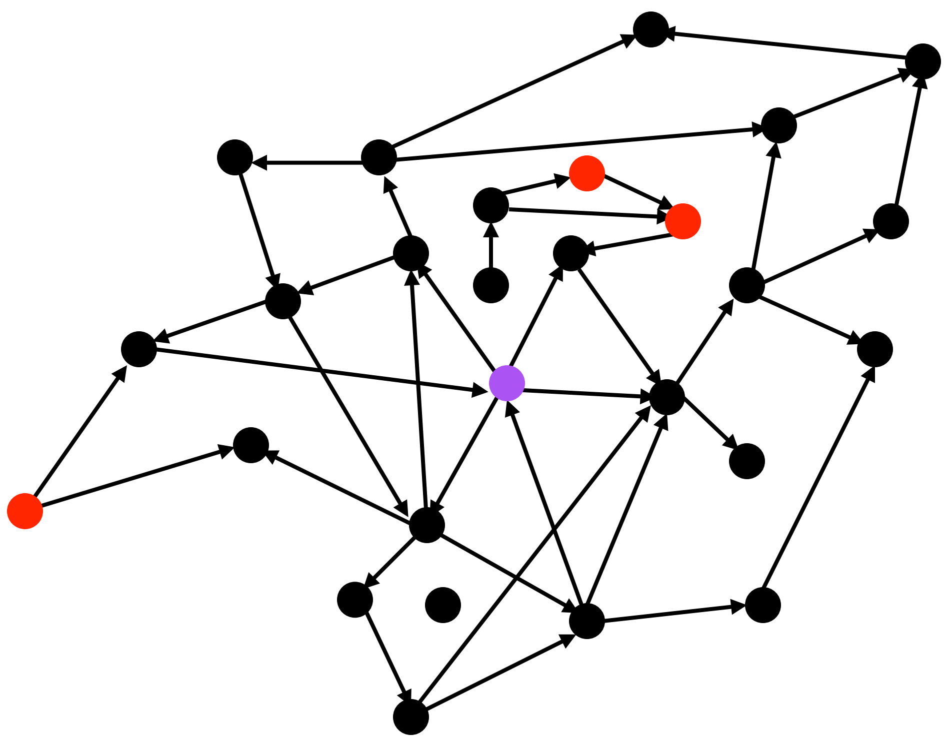
- Objectifs :
  - construire la structure de Kripke ou le système de transitions de l'application (programmée comme on le souhaite)
  - vérifier les propriétés de sécurité (safety) ou vivacité (liveneness) sur ce système. Cela demande en général un prétraitement des formules, par exemple, leur transformation en automates observateurs.
  - produire des diagnostics et contre-exemples sur les propriétés fausses
  - lutter par tous les moyens contre l'explosion en taille, potentiellement exponentielle

- Deux grandes classes de méthodes
  1. méthodes explicites : énumérer explicitement (mais intelligemment) les états et transitions
  2. méthodes implicites : travailler avec des formules logico-numériques décrivant états et transitions sans les énumérer

Propriétés de sécurité : exploration d'états

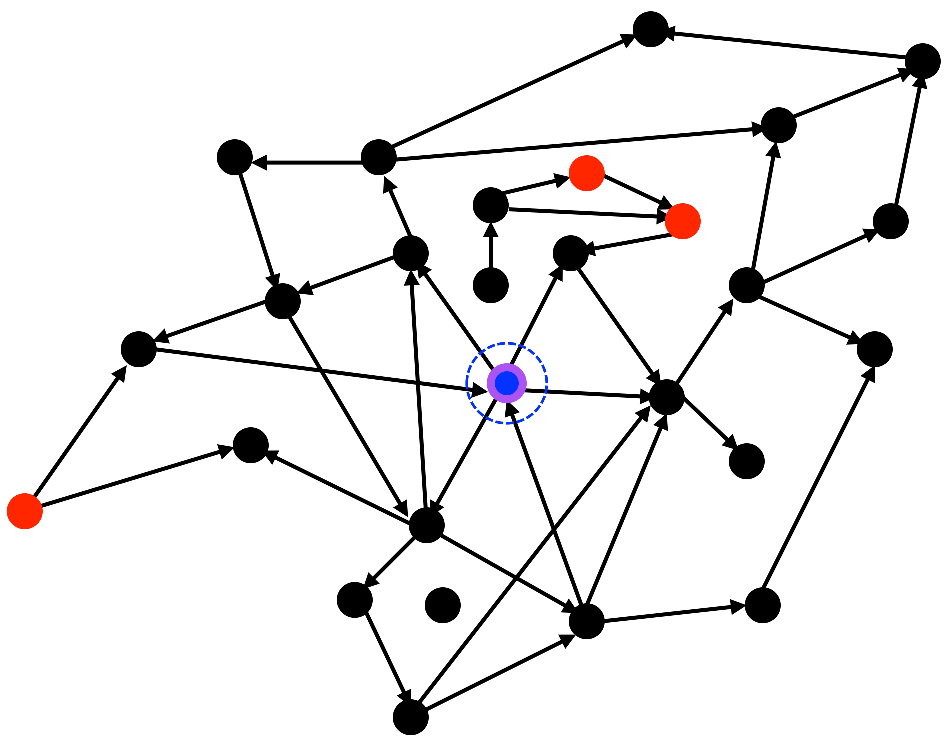


Propriété P = ensemble d'états



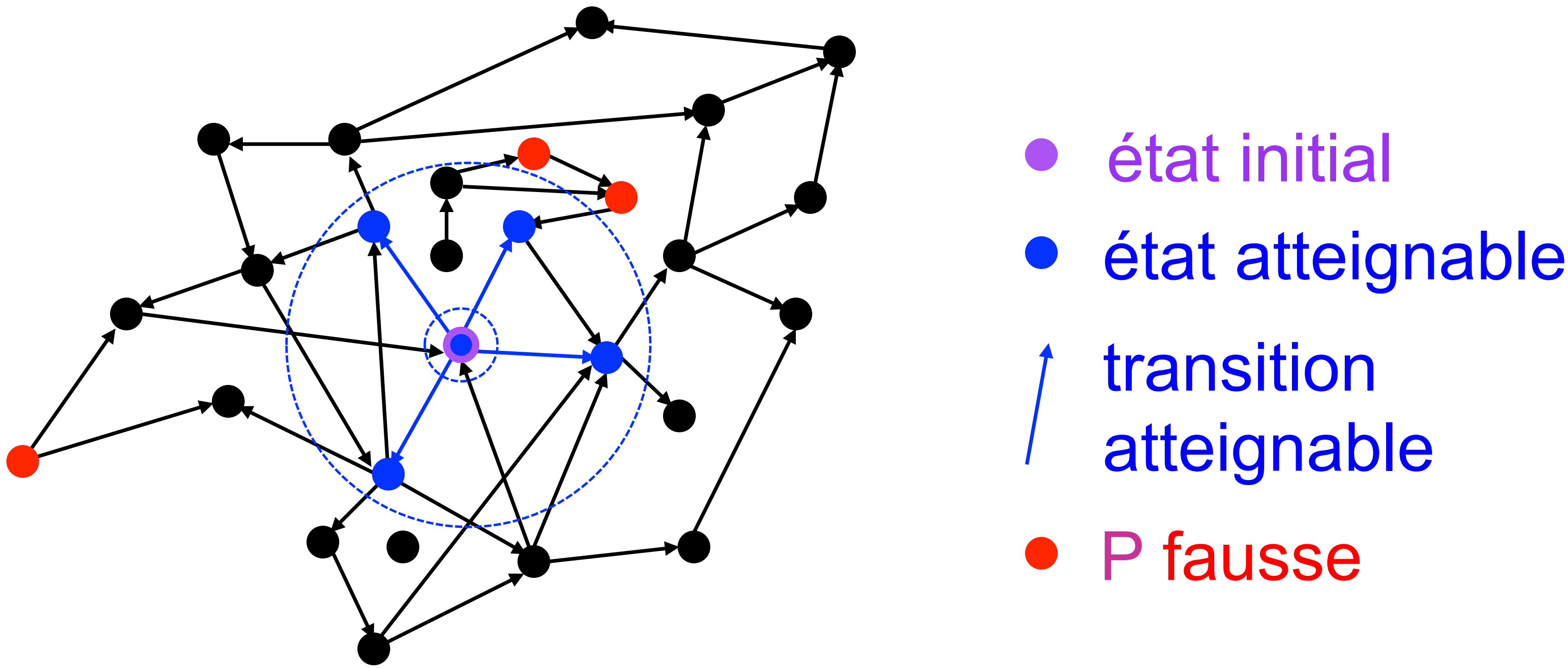
- état initial
- état potentiel
- transition potentielle
- P fausse

Analyse en avant : marquer les états atteignables

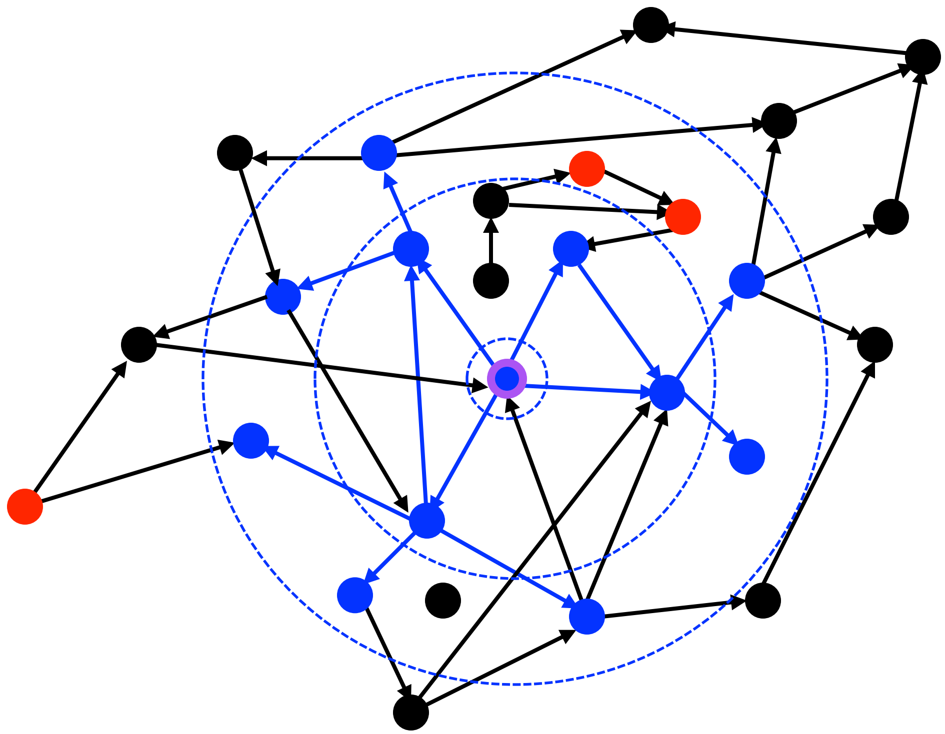


- état initial
- état potentiel
- transition potentielle
- P fausse

Analyse en avant : marquer les états atteignables (1)

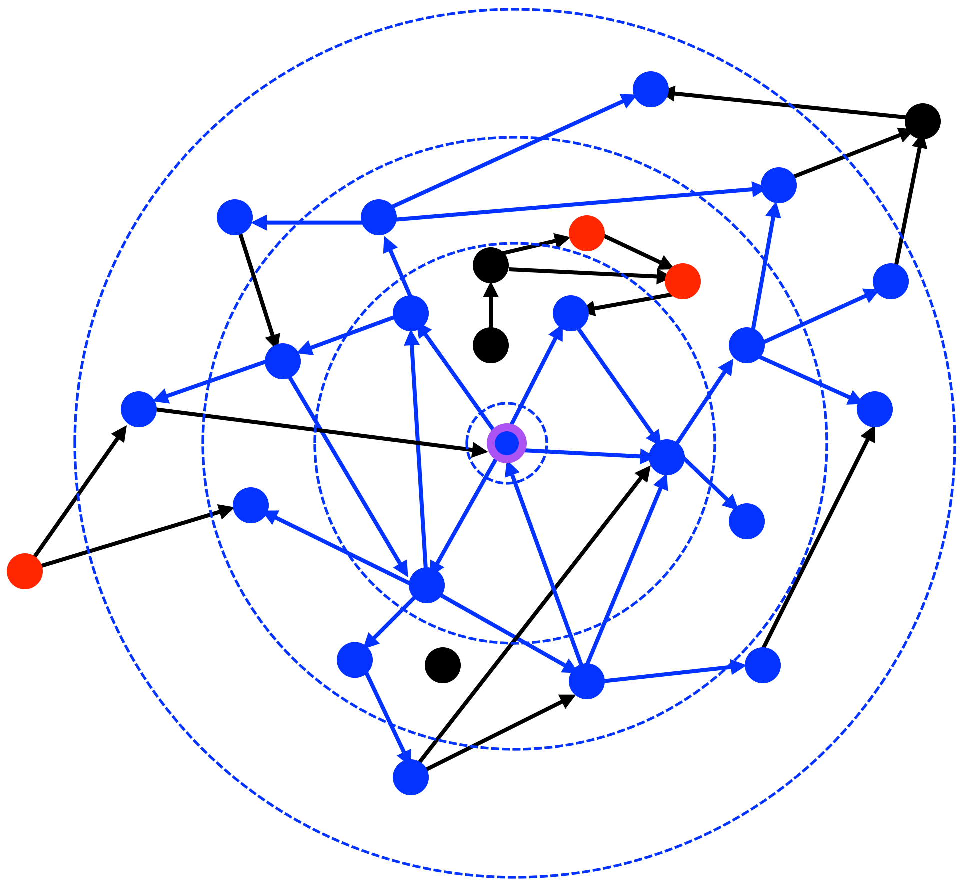


Analyse en avant : marquer les états atteignables (2)



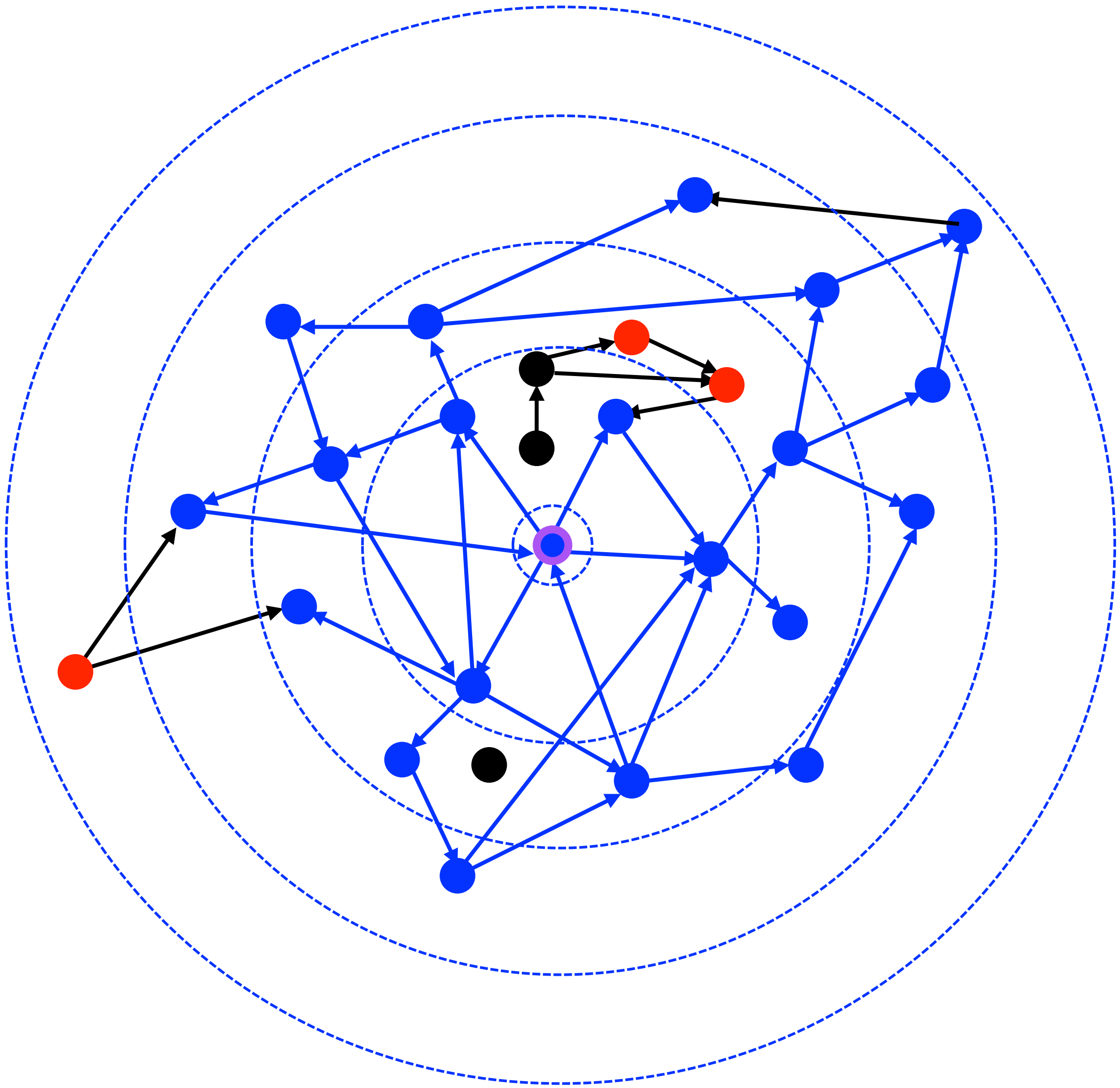
- état initial
- état atteignable
- transition atteignable
- P fausse

Analyse en avant :
marquer les états atteignables (3)



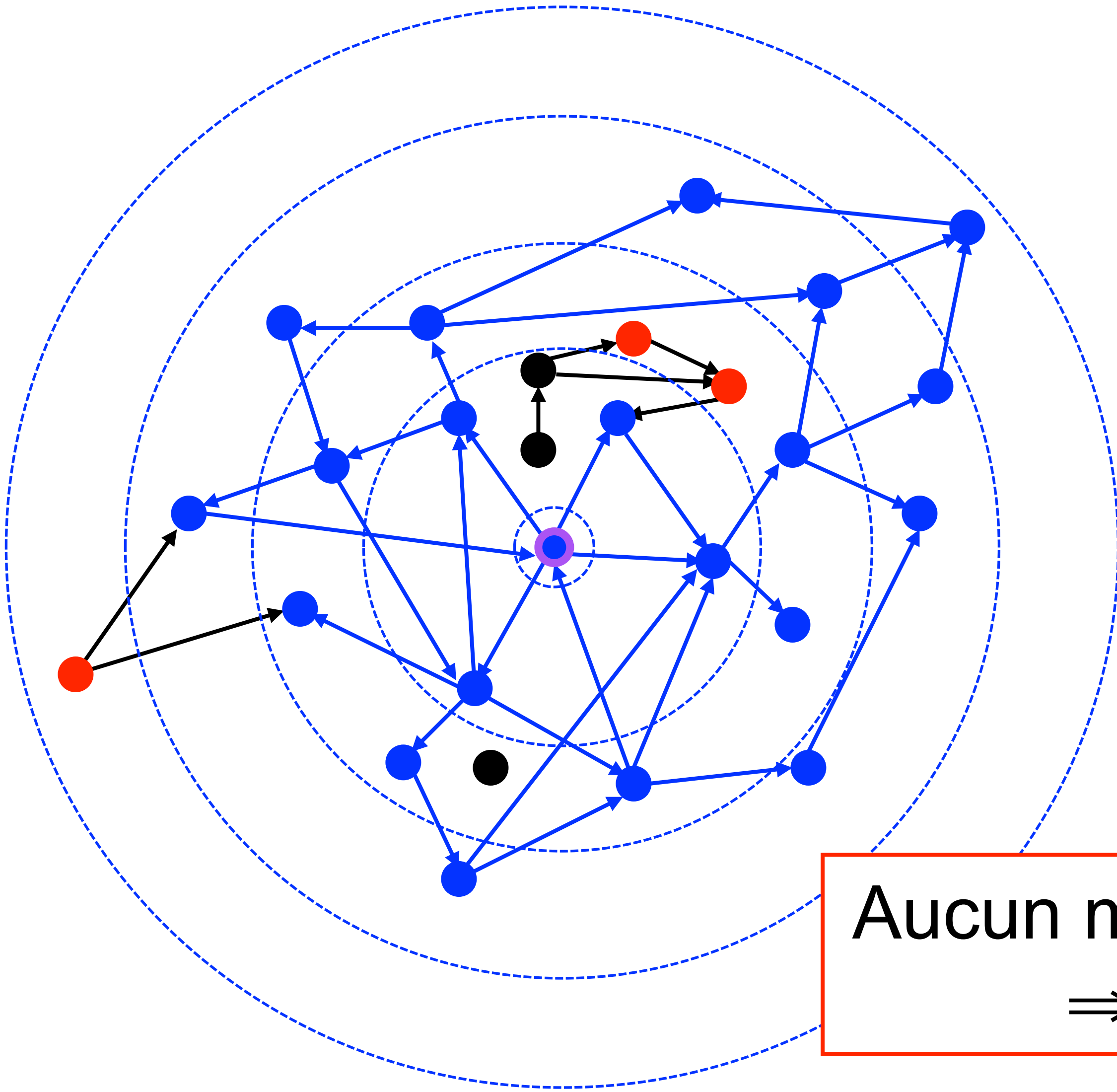
- état initial
- état atteignable
- transition atteignable
- P fausse

Analyse en avant : marquer les états atteignables (4)



- état initial
- état atteignable
- transition atteignable
- P fausse

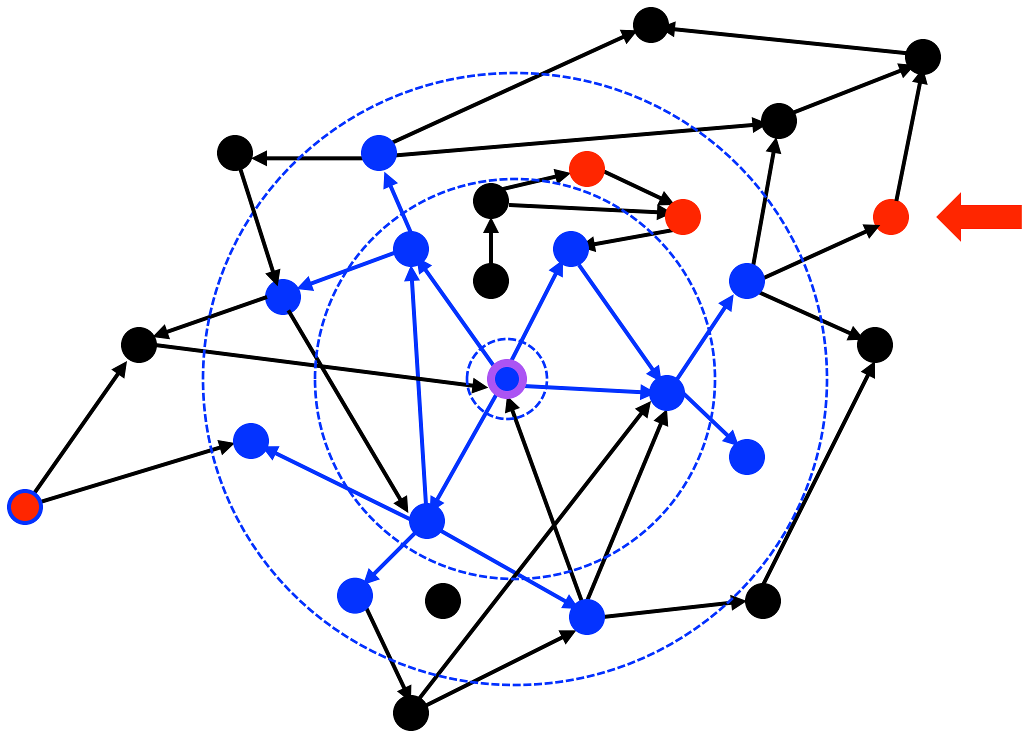
Analyse en avant : marquer les états atteignables (5)



- état initial
- état atteignable
- transition atteignable
- P fausse

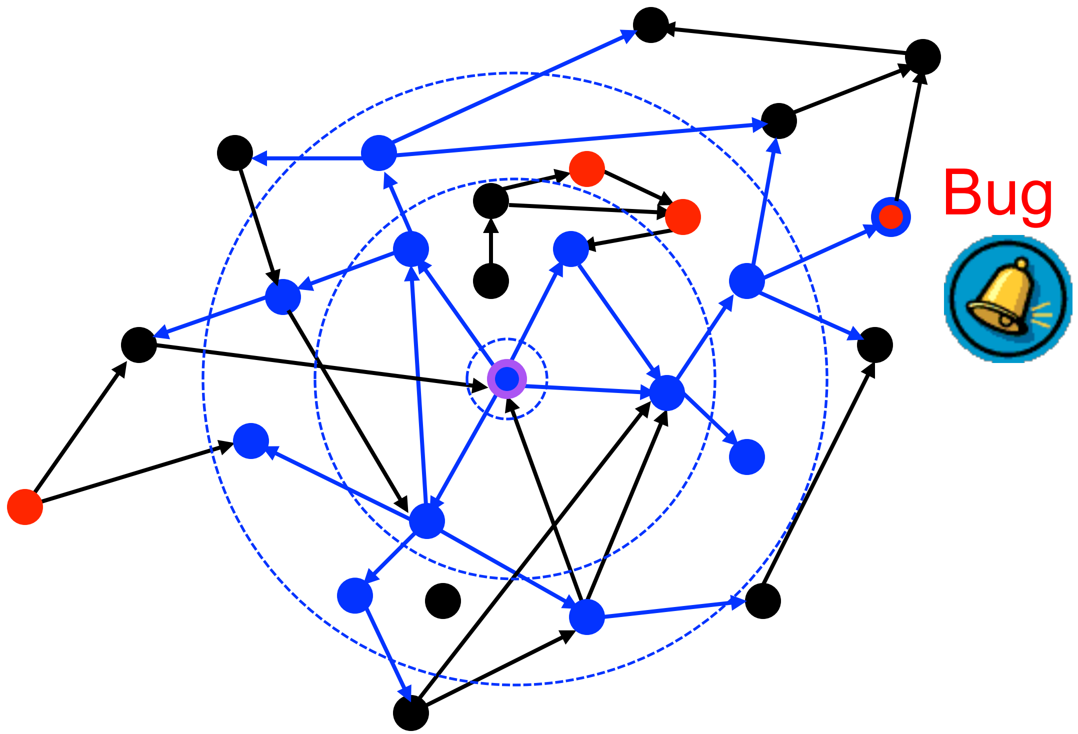
Aucun mauvais état atteint
⇒ P est vraie

Analyse en avant : Production de contre-exemple



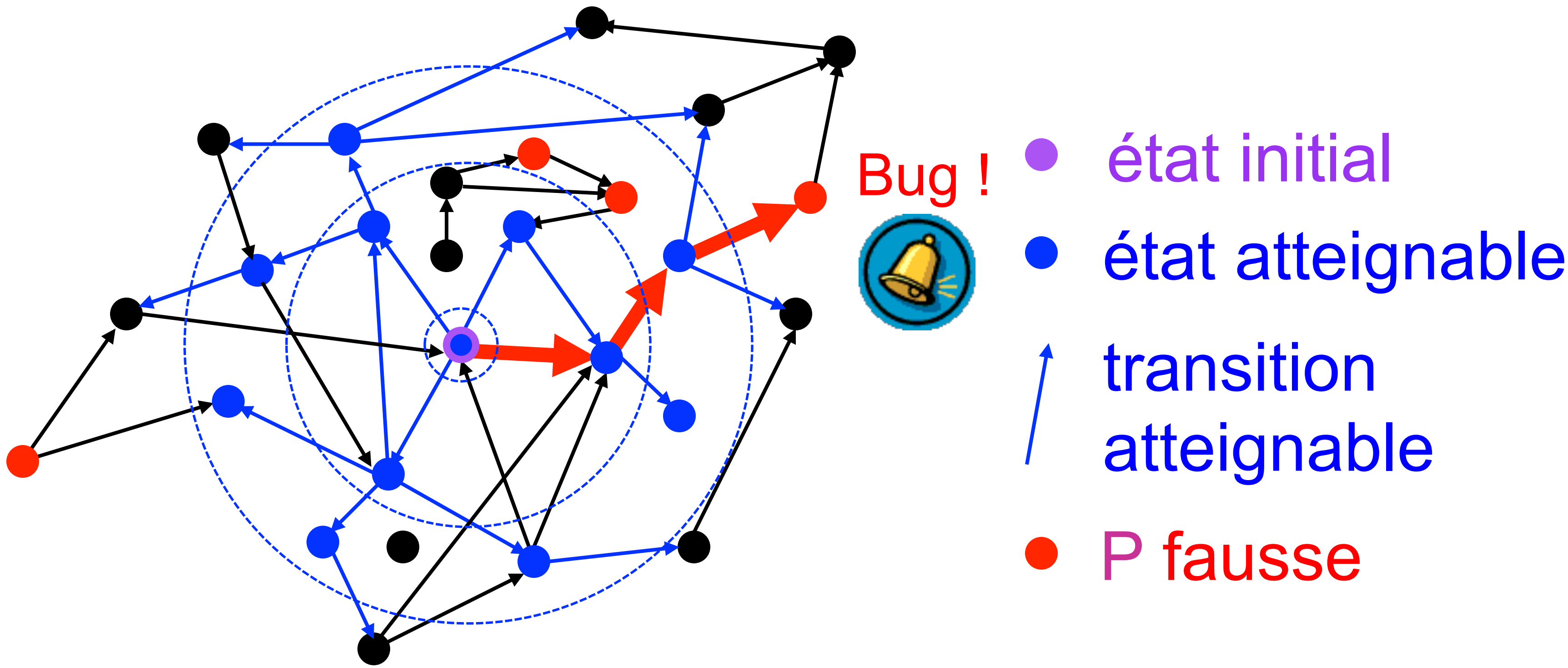
- état initial
- état atteignable
- transition atteignable
- P fausse

Analyse en avant : Production de contre-exemple



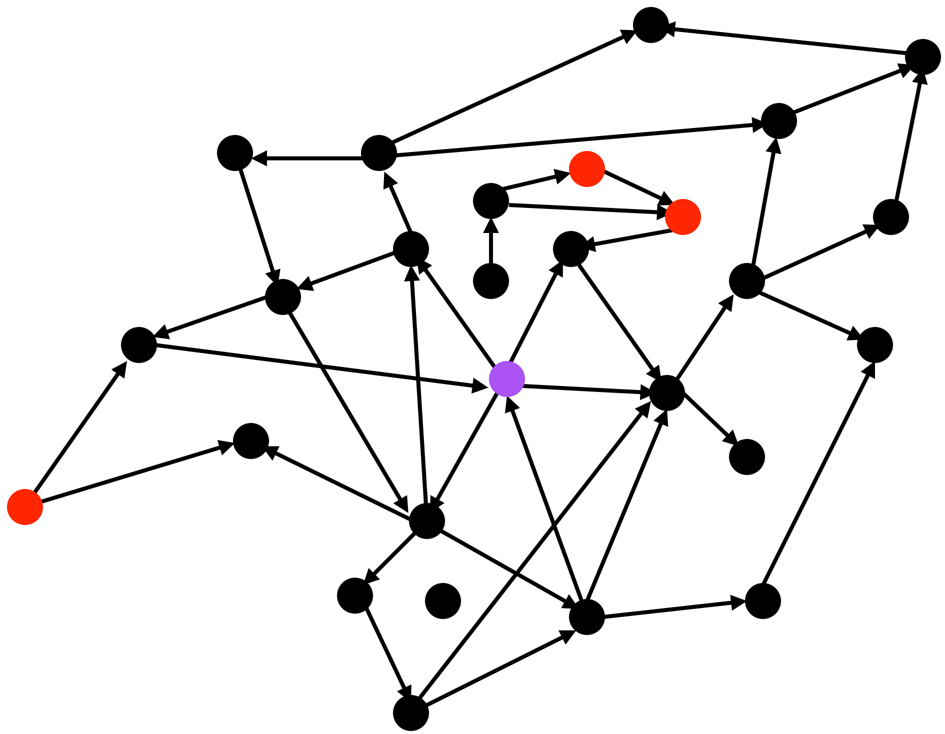
- état initial
- état atteignable
- transition atteignable
- P fausse

Analyse en avant : Production de contre-exemple



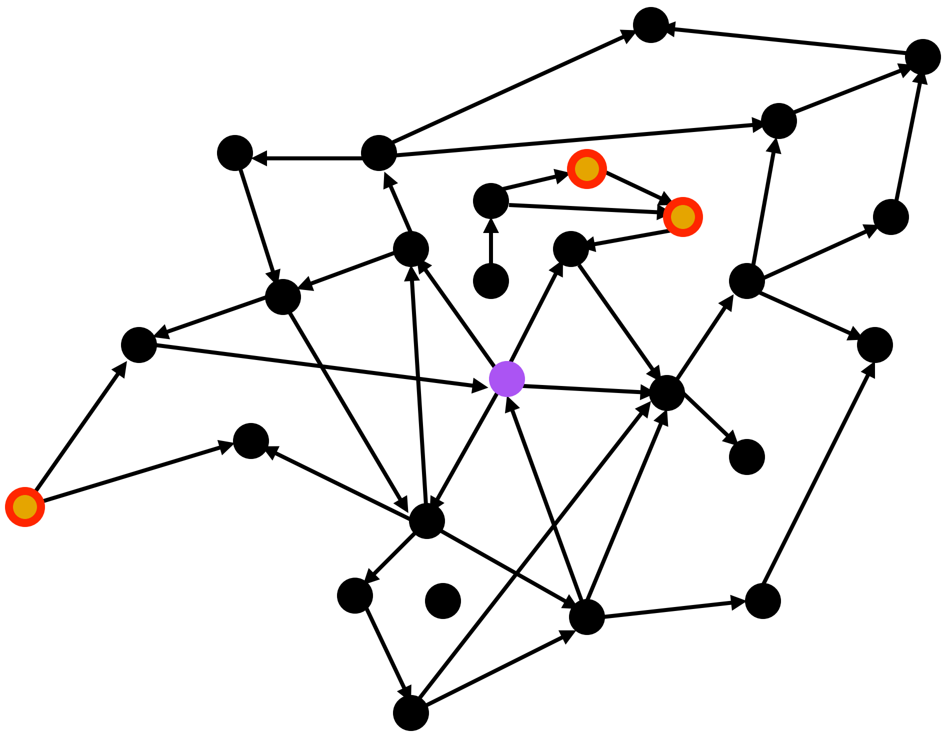
Avantage : contre-exemple de longueur minimale

Analyse en arrière



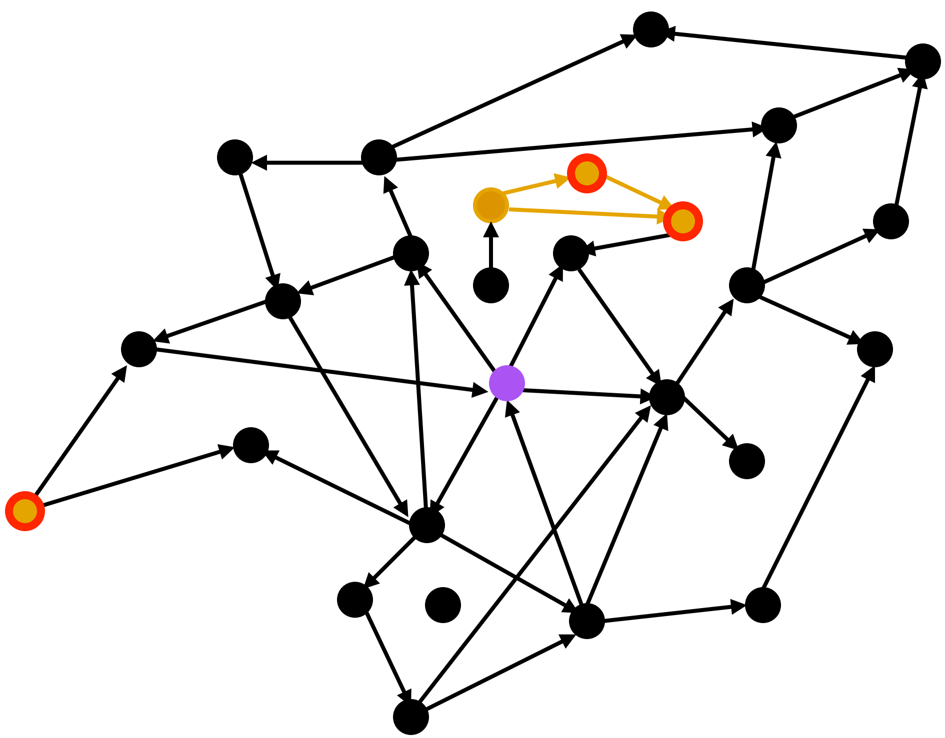
- état initial
- état potentiel
- transition potentielle
- P fausse

Analyse en arrière
Marquer les états faux



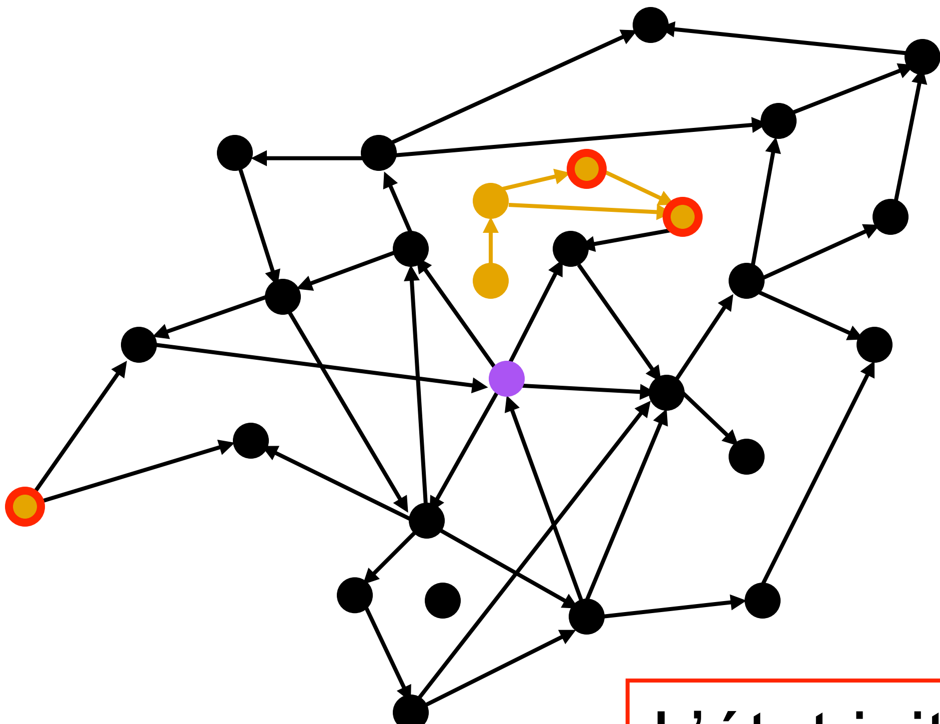
- état initial
- état potentiel
- transition potentielle
- marqué

Analyse en arrière
Marquer les prédécesseurs (1)



- état initial
- état potentiel
- transition potentielle
- marqué

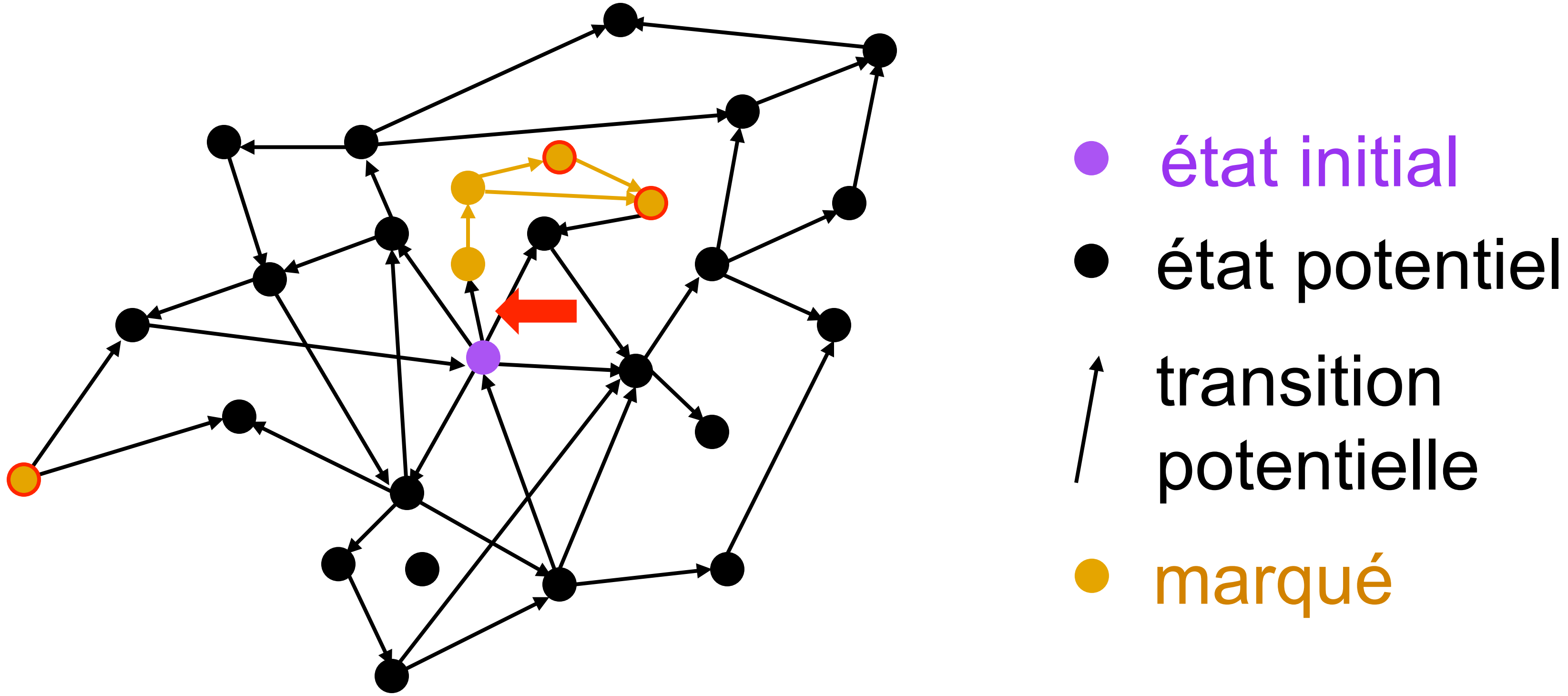
Analyse en arrière
Marquer les prédécesseurs (2)



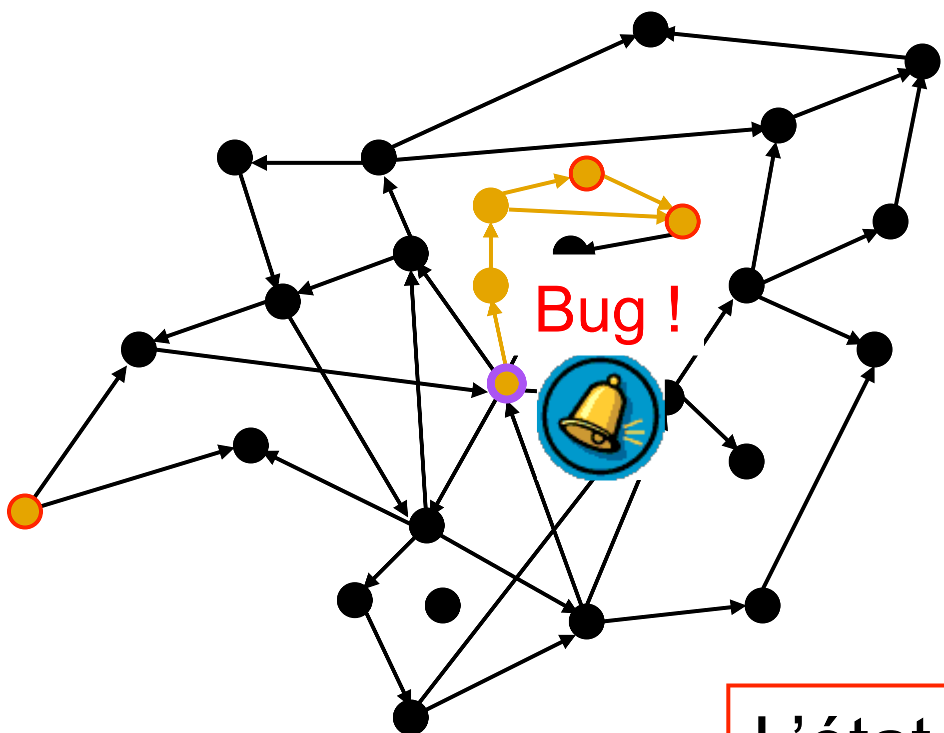
- état initial
- état potentiel
- transition potentielle
- marqué

L'état initial n'est pas marqué
$\Rightarrow$ P est vraie

Analyse en arrière
Production de contre-exemple



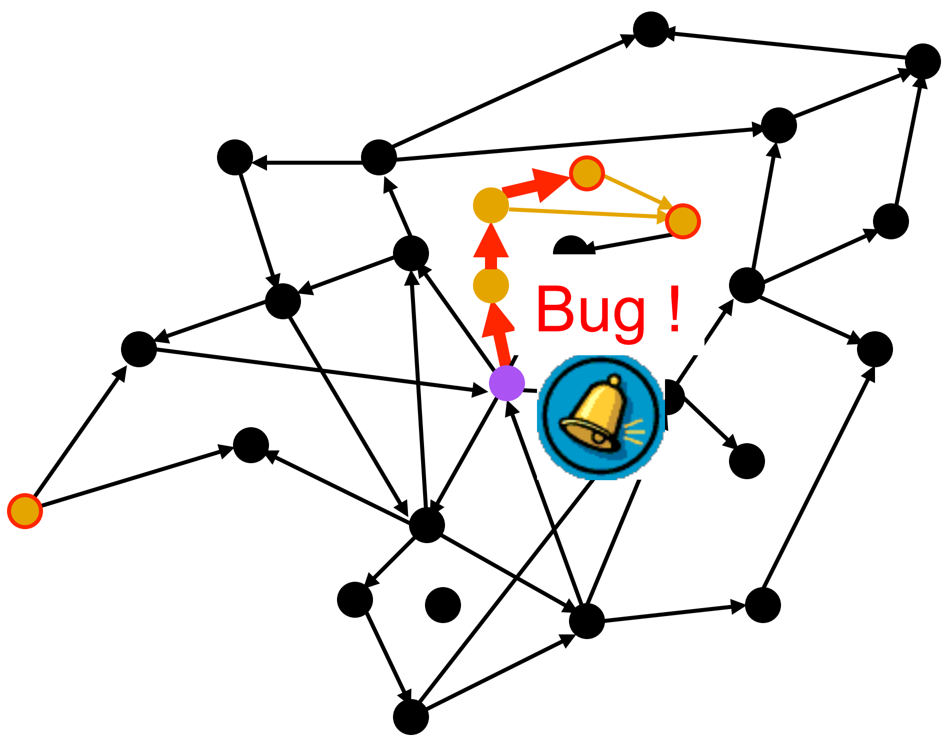
Analyse en arrière
Production de contre-exemple



- état initial
- état potentiel
- transition potentielle
- marqué

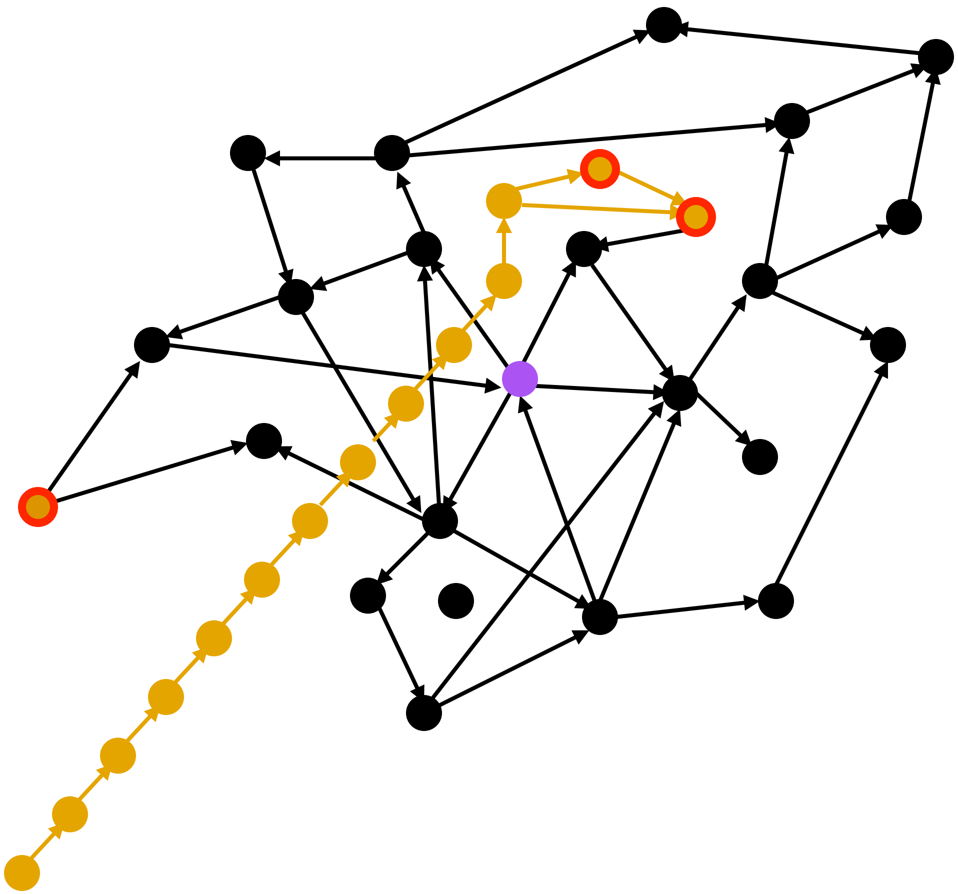
L'état initial est marqué
⇒ P est fausse

Analyse en arrière
Production de contre-exemple



- état initial
- état potentiel
- transition potentielle
- marqué

Pas forcément plus rapide que l'analyse en avant...



- état initial
- état potentiel
- transition potentielle
- marqué

Algorithmes explicites

- Principes de base :
  - logiques temporelles : transformer les formules à prouver en observateurs à explorer de façon synchrone avec le modèle (non trivial !)
  - construire explicitement les états et transitions
  - lutter contre l'explosion de leur nombre
  - produire des contre-exemples pour les propriétés fausses

- En pratique :
  - exploration exhaustive ou aléatoire
  - abstraction par hachage des états, avec collisions possibles (peut rater des bugs)
  - réductions par ordres partiels (commutativité d'opérations)
  - réduction par symétries (si les acteurs sont les mêmes)
  - techniques d'abstraction de l'interprétation abstraite
  - implémentations massivement parallèles

Algorithmes explicites

- Systèmes principaux :
  - Historiques : CESAR de J-P. Queille et J. Sifakis, EMC de E. Clarke et A. Emerson
  - SPIN de G. Holzmann, avec son langage Promela, proche de C, et une grande communauté d'utilsateurs (protocoles, algos distribués)
  - COSPAN de R. Kurshan
  - CADP de H. Garavel, R. Mateescu *et. al.*, originellement sur LOTOS, calcul de processus pour les télécommunications, puis adapté à beaucoup de langages
  - (Outils pour les réseaux de Petri, etc.)

Agenda

1. Les logiques temporelles
2. Les systèmes de transitions et la bisimulation
3. La vérification par observateurs
4. Les algorithmes de vérification explicite
5. Le Sudoku en calcul booléen

Résolution booléenne d'un Sudoku

http://www.cs.qub.ac.uk/~I.Spence/SuDoku/SuDoku.html

4				7	9	6		
		9						
			6				8	3
								7
5			8			1		
7	8			1	2		4	
6	5							
				8		7		4
		4		2		3		

4	3	8	2	7	9	6	5	1
1	6	9	3	5	8	4	7	2
2	7	5	6	4	1	9	8	3
9	2	1	5	6	4	8	3	7
5	4	6	8	3	7	1	2	9
7	8	3	9	1	2	5	4	6
6	5	7	4	9	3	2	1	8
3	9	2	1	8	5	7	6	4
8	1	4	7	2	6	3	9	5

Définition booléenne d'une grille

729 variables xyz : vraie si la case (x,y) contient la valeur z

4				7	9	6		
		9						
			6				8	3
								7
5			8			1		
7	8			1	2		4	
6	5							
				8		7		4
		4		2		3		

$114 \wedge 157 \wedge 169 \wedge 176$
$\wedge\ 239$
$\wedge\ 346 \wedge 388 \wedge 393$
$\wedge\ 497$
$\wedge\ 515 \wedge 548 \wedge 571$
$\wedge\ 617 \wedge 628 \wedge 651 \wedge 662 \wedge 684$
$\wedge\ 716 \wedge 725$
$\wedge\ 858 \wedge 877 \wedge 894$
$\wedge\ 934 \wedge 952 \wedge 973$

25 clauses booléennes

Contraintes booléennes du Sudoku

- la case 1-1 contient un entier de 1 à 9 : 1 clause

111 v 112 v 113 v 114 v 115 v 116 v 117 v 118 v 119

- un seul entier dans la case 1-1 : 36 clauses

∧ (¬111 v ¬112) ∧ (¬111 v ¬113) ... ∧ (¬111 v ¬119)
∧ (¬112 v ¬113) ∧ (¬112 v ¬114) ... ∧ (¬112 v ¬119)
∧ ...
∧ (¬118 v ¬119)

- à répéter pour les 81 cases : 81×(1+36) = 2997 clauses

Contraintes booléennes du Sudoku

- la ligne 1 contient l'entier 1 : 1 clause

  ∧ (111 ∨ 121 ∨ 131 ∨ 141 ∨ 151 ∨ 161 ∨ 171 ∨ 181 ∨ 191)

- l'entier 1 n'est qu'une fois dans la ligne 1 : 36 clauses

  ∧ (¬111 ∨ ¬121) ∧ (¬111 ∨ ¬131) ... ∧ (¬111 ∨ ¬191)
  ∧ (¬121 ∨ ¬131) ∧ (¬121 ∨ ¬141) ... ∧ (¬121 ∨ ¬191)
  ∧ ...
  ∧ (¬181 ∨ ¬191)

- idem pour chaque entier, ligne, colonne et carré 3×3 :

  27×9×(1+36) = 8 991 clauses

- total contraintes : 2997+8991 = 11 988 clauses

  plus grille donnée 25 clauses

  total général **12 013 clauses**

Bibliographie

- Logiques temporelles

  - J-P. Queille, J. Sifakis. *Specification and verification of concurrent systems in CESAR.* International Symposium of Programming, Springer *LNCS 137, 1982, pp 337 – 351*
  - E.M. Clarke, E.A. Emerson, and A.P. Sistla. *Automatic verification of finite-state concurrent systems using temporal logic specifications*. ACM Trans. Program. Lang. Syst. 2, pp. 244–263 (1986).
  - Z. Manna, A. Pnueli. *The temporal logic of reactive and concurrent systems.* Springer, New York (1992).
  - L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers* . Addison-Wesley (2002).

- Calculs de processus et bisimulation

  - R. Milner. *Communication and Concurrency.* Prentice Hall, 1989

- Observateurs

  - N. Halbwachs, F. Lagnier, and Pascal Raymond. *Synchronous Observers and the Verification of Reactive Systems*. Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93, Twente. Springer Verlag (1993).

Bibliographie

- Méthodes explicites
  - G. Holtzmann. *The Spin Model Checker — Primer and Reference Manual*, Addison-Wesley, 2003
  - H. Garavel, F. Lang, R. Mateescu, W. Serwe. *CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes* International Journal on Software Tools for Technology Transfer (STTT), 15(2):89-107, April 2013.
- Méthodes implicites
  - J. Burch, E.. Clarke, K. McMillan, D. Dill, L. Hwang: *Symbolic Model Checking: 10^20 States and Beyond*. Proc. LICS Conference 1990.
  - K. McMillan. *Interpolation and SAT-based Model Checking*. Proc Computer-Aided Verification Converence (CAV03), 2003.
- Survey généraux
  - E. Clarke. The Birth of Model-Checking. 25 Years of Model Checking, Springer-Verlag Berlin, Heidelberg, 2008.
  - R. Jahla, R. Majumdar. *Model-Cheking Software Systems.* ACM Computing Surveys, Volume 41 Issue 4, October 2009.