Concepts et Model Checking

Stefan Schwoon

ENSIIE, Année 2022/2023

Organisation

Cours: lundi de 14h à 15h45

TD/TP: lundi de 16h à 17h45

Enseignant: Stefan Schwoon (`schwoon@lsv.fr`)
`http://www.lsv.fr/~schwoon`

Durée: 6 semaines (du 23 janvier au 6 mars)

Évaluation: Examen (le 6 mars)

Connaissances préalables:

logique, théorie d'automates . . .

Littérature:

Clarke, Grumberg, Peled: Model Checking, MIT Press, 1999

Baier, Katoen: Principles of Model Checking, 2008

Emerson: Temporal and Modal Logic, chapître 16 du *Handbook of Theoretical Computer Science*, vol. B, Elsevier, 1991

Vardi: An Automata-Theoretic Approach to Linear Temporal Logic, LNCS 1043, 1996

Holzmann: The SPIN Model Checker, Addison-Wesley, 2003

Partie 1: Introduction

Que veut dire “Model-Checking” ?

Que veut dire “Model-Checking” ?

Que veut dire “Model-Checking” ?

Que veut dire “Model-Checking” ?

Notion de logique :

Tester si un object (p.ex. affectation des variables) est modèle d’une formule.

Ici : appliquer de la logique pour vérifier la correction d’un système

Logique temporelle : extension de la logique du premier ordre

Exemple: Quicksort - correct ou non ?

```
void quicksort (int left, int right) {
   int lo,hi,piv;
   if (left >= right) return;
   piv = a[right]; lo = left; hi = right;
   while (lo <= hi) {
     if (a[hi] > piv) {
       hi = hi - 1;
     } else {
       swap a[lo],a[hi];
       lo = lo + 1;
     }
   }
   quicksort(left,hi);
   quicksort(lo,right);
}
```

Exemple : Quicksort

On considère que l'algorithme est correct si :

Il trie correctement.

Il termine pour tout argument légal.

Dans ce cas, l'algorithme ne termine pas toujours,
notamment quand `a[right]` est l'élément maximal.

Remarque : Ce bug aurait peut être trouvé par testing rigureux.

Bug du processeur Pentium (1994)

Le Pentium produisait de faux résultats pour certains opérations mathématiques:

$$4195835 - (4195835/3145727) \times 3145727 = 256$$

La raison en étaient quelques valeurs fausses dans un tableau précalculé pour effectuer la division.

Coût approximatif pour Intel : 500 millions de dollars

Exemple: Variables partagées

Démonstration : counter.c

On crée deux threads qui augmentent une variable partagée n fois.

Résultat attendu : $2n$

Exemple: Variables partagées

Démonstration : counter.c

On crée deux threads qui augmentent une variable partagée n fois.

Résultat attendu : $2n$

Pourtant, le résultat réel est souvent moins que $2n$.

$\rightarrow$ Condition de compétition (*race condition*),
facile à manquer lors d'une inspection manuelle du code.

$\rightarrow$ Solution : Assurer *exclusion mutuelle* sur l'accès à n.

Exemple: Exclusion mutuelle (Peterson)

Variables partagées : `flag[0], flag[1], victime`, initialement 0

Code du processus i=0,1 (autour d'une zone critique) :

```
while (1) {
  ...
  autre = 1-i;
  flag[i] = 1;
  victime = i;
  while (victime == i && flag[autre]);
  // zone critique
  flag[i] = 0;
  ...
}
```

Exemple: Exclusion mutuelle (Peterson)

Variables partagées : `flag[0], flag[1], victime`, initialement 0

Code du processus i=0,1 (autour d'une zone critique) :

```
while (1) {
  ...
  autre = 1-i;
  flag[i] = 1;
  victime = i;
  while (victime == i && flag[autre]);
  // zone critique
  flag[i] = 0;
  ...
}
```

L'algorithme assure bien l'exclusion mutuelle. Pourtant, sa correction est déjoué par les optimisations faites sur les processeur modernes (réordonanncements des read et write).

Approches pour assurer la correction des systèmes

Éviter les erreurs:

- langages de programmation appropriés
- méthodes du génie logiciel

Détecter les erreurs:

- Simulation, testing

Prouver leur absence:

- Vérification déductive (Hoare, preuve automatique)
- Vérification automatique (model checking)

Simulation et testing

Trouver des erreurs dans la phase de conception (simulation) ou dans le produit final (testing).

Méthodes: Blackbox/whitebox testing, critères de couverture, etc

Avantages: peut trouver des erreurs rapidement et économiquement

Inconvénients: incomplet

$\rightarrow$ Aucun critère de couverture ne garantit l'absence des erreurs.

$\rightarrow$ Pas du tout adapté aux effets non-déterministes tel que la concurrence.

Testing et vérification

Simulation et testing peuvent identifier des erreurs mais pas prouver leur absence. Ces méthodes considèrent un sous-ensemble des exécutions potentielles.

$\rightarrow$ pas suffisant pour des aspects de sécurité

La vérification considère toutes les exécutions d'un système

$\rightarrow$ on peut prouver l'absence d'erreurs
(mais c'est plus couteux/difficile à mettre en œuvre)

Vérification déductive

Preuve par sémantique formelle du programme (Dijkstra, Hoare et al.)

Exemple: Logique de Hoare:

$$\{P\}\ S\ \{Q\}$$

“Si P est vrai avant l’exécution de S, alors Q est vrai après.”

Règles de preuve, p.ex.:

$$\{P\}\ \mathbf{skip}\ \{P\} \qquad \{P[x/e]\}\ x := e\ \{P\} \qquad \frac{\{P\}\ S_1\ \{Q\} \wedge \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1; S_2\ \{R\}}$$

Exemple: régle de preuve pour les boucles

$$\{P\}\ \mathbf{while}\ \beta\ \mathbf{do}\ C\ \{Q\}$$

Il faut trouver une invariante I avec les propriétés suivantes :

$$P \Rightarrow I \qquad \{I \wedge \beta\}\ C\ \{I\} \qquad I \wedge \neg\beta \Rightarrow Q$$

Terminaison: trouver une fonction $f(x, y, \ldots)$ sur les variables telle que

$$\{\beta \wedge f(x, y, \ldots) = k\}\ C\ \{f(x, y, \ldots) < k\} \qquad f(x, y, \ldots) \leq 0 \Rightarrow \neg\beta$$

Le programme C est considéré correct si $\{true\}\ C\ \{P\}$, où P est la propriété finale désirée.

Vérification déductive

Avantages:

Complète; limitée seulement par l'ingéniosité humaine.

Inconvénients:

voir ci-dessus

Preuves manuelles lourdes (peut-être aidé par la démonstration automatique).

Le schéma ci-dessus marche pour les systèmes séquentielles (pas de concurrence).

Plutôt conçu pour les programmes genre entrée/sortie, mais pas pour les systèmes réactives.

Systèmes réactives

Exemples: système d'exploitation, serveurs, distributeur de billets, ...

pas de "fonction" calculée, terminaison non-désirable

On s'intéresse à certains propriétés de leur exécutions telles que :

Absence de blocage

Exclusion mutuelle dans une "zone critique"

Progrès: un processus qui souhaite entrer dans une zone critique y parviendra finalement.

$\Rightarrow$ formalisation par logique temporelle

Rappels: Calcul/Logique propositionnel(le)

Formules avec prédicats :

$A \quad \widehat{=} \quad$ “Anne est architecte”

$B \quad \widehat{=} \quad$ “Bruno is boulanger”

and connecteurs, p.ex. $\wedge$ (“et”), $\vee$ (“ou”), $\neg$ (“non”), $\rightarrow$ (“implique”).

Exemples

Formules de logique propositionnelle :

$A \wedge B$ (“Anne est architecte et Bruno est boulanger”)

$\neg B$ (“Bruno n’est pas un boulanger”)

Une telle formule, est-elle vraie ?

Ça depend.

Certaines formules sont toujours vraies ($A \vee \neg A$) ou toujours fausses ($B \wedge \neg B$).

Mais en général, les formules sont évaluées par rapport à une affectation des prédicats.

Logique propsitionnelle

Une affectation $\mathcal{B}$ est une fonction qui donne (1 ou 0) pour tout prédicat.

La semantique d'une formule (définie inductivement) est l'ensemble des affactions qui rendent la formule vraie, notée $[\![F]\!]$. P.ex.,

Si $F = A$ alors $[\![F]\!] = \{ \mathcal{B} \mid \mathcal{B}(A) = 1 \}$;

Si $F = F_1 \wedge F_2$ alors $[\![F]\!] = [\![F_1]\!] \cap [\![F_2]\!]$; ...

D'autres notations: $\mathcal{B} \models F$ ssi $\mathcal{B} \in [\![F]\!]$.
On dit : "$\mathcal{B}$ satisfait F" ou "$\mathcal{B}$ est un modèle de F".

Model-checking pour logique propositionnelle

Problème:

Étant donné une affectation $\mathcal{B}$ et une formule F du calcul propositionnel, tester si $\mathcal{B}$ est un modele de F.

Solution:

Remplacer les prédicats par leurs valeurs dans $\mathcal{B}$, utiliser le tableau de vérité pour voir si ça donne 1 ou 0.

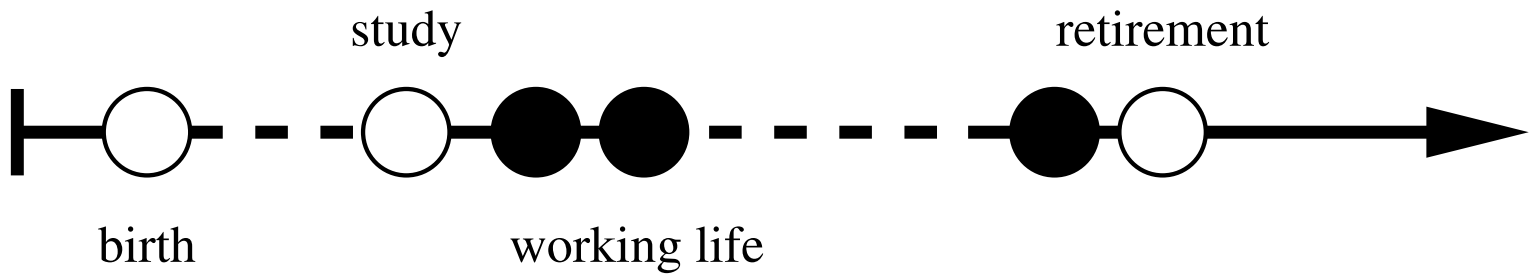
Exemples: Let $\mathcal{B}_1(A) = 1$ et $\mathcal{B}_1(B) = 0$. Alors $\mathcal{B}_1 \not\models A \wedge B$ et $\mathcal{B}_1 \models \neg B$.

Let $\mathcal{B}_2(A) = 1$ et $\mathcal{B}_2(B) = 1$. Alors $\mathcal{B}_2 \models A \wedge B$ et $\mathcal{B}_2 \not\models \neg B$.

Logique temporelle

On permet aux prédicats de changer au fil de temps.

Exemple: Valeurs de *A* dans la vie d'Anne:



Propriétés qu'on veut exprimer :

Anne sera finalement architecte (à un point dans le futur).

Anne sera architecte jusqu'à ce qu'elle prend sa retraite.

$\Longrightarrow$ Extension de la logique propositionnelle avec des modalités temporelles ("finalement", "jusqu'à").

Aperçu

Logique temporelles linéaire (exemple: LTL)

formules avec modalités temporelles

évaluée sur les *séquences* (infinies) d'affectations

Le problème de model-checking pour LTL: Étant donné une formule de LTL et une séquence de valuations, tester si la séquence est un modèle de la formule.

Logique temporelles branchantes (CTL, CTL*)

On considère des *arbres* (infinis) d'affectations

Interprétation: nondéterminisme; plusieurs développement potentiels

Le rapport avec la vérification de programmes

Espace d'états d'un programme:

- compteur d'instruction
- valeurs de variables
- contenu de la pile, du tas, ...

Prédicats, p.ex.

- "x possède une valeur positive."
- "Le compteur d'instructions est dans la ligne ℓ."

Étant donné un ensemble de prédicats, un état du programme donne une affectation de ces prédicats.

Programmes et logique temporelle

Logique temporelle linéaire:

Toute exécution donne une séquence d'affectations.

Interprétation du programme: l'ensemble de ces séquences

Question : La formule, est-elle satisfaite par toutes les séquences ?

Logique branchante:

Le programme peut brancher à certains endroits, ses exécutions forment un arbre d'affectations.

Interprétation du programme: arbre avec l'état initial comme racine

Question : Cet arbre, satisfait-il la formule ?

Donc: problème de vérification $\widehat{=}$ problème de model-checking

Model checking

Généralement parlant, le terme model checking est donné aux méthodes qui :

verifient automatiquement si un système satisfait une spécification donnée;

soit prouvent la correction du système par rapport à la spécification;

soit trouvent un contre-exemple, une exécution qui ne respecte pas la spécification (au moins dans le cadre de LTL).

Les “pros et cons” du model checking

Avantages:

automatique(!)

bien adapté aux systèmes réactifs, concurrents, distribués

on peut tester des propriétés complexes, pas just l’accessibilité

Inconvénients:

En général, les programmes sont aussi expressifs qu’une machine de Turing
$\rightarrow$ indécidabilité

Approche: on étudie des sous-classes où le problème reste décidable,
p.ex. les automates finis

Espace d’états souvent très, très large $\rightarrow$ (algorithmiquement) couteux

approche: algorithmes et structures de données efficaces

Limites du model checking

On ne peut pas espérer de vérifier n'importe quelle propriété de n'importe quel programme !

Il faut éventuellement considérer un modèle simplifié d'un programme focalisé sur ses aspects "importants".

La construction d'un tel modèle, la spécification et la vérification elle-même sont coûteux et nécessitent un effort.

$\Rightarrow$ utile dans les premières phases de conception

$\Rightarrow$ indispensable lorsque les erreurs sont très coûteuses ou même fatales (processeur, protocôlesi de communication, avions, . . .)

Succès du model checking

Depuis les années 1970: recherche sur les fondations théoriques

Depuis les années 1990: applications dans l'industrie

D'abord vérification de matériel, puis logiciel :

- vérification du protocôle de cohérence de cache dans le IEEE Futurebus+ (1992)

  L'outil SMV était en mésure de trouver plusieurs bugs quatre ans après la conception initial du bus.

- vérification de l'unité arithmétique du Pentium4 (2001)
- Static Driver Verifier (Microsoft, 2000–2004) (pilotes dans Windows)

groupes de recherche dans les grandes entreprises: IBM, Intel, Microsoft, ...

Prix Turing 2007 pour les pionniers (Clarke, Emerson, Sifakis)

Objectifs du cours

Fondements du model checking, théorie, applications

Modélisation: systèmes de transition, structures de Kripke; outils: Spin, SMV

Spécification: LTL, CTL

Vérification: techniques fondamentales et extensions (BDDs, abstraction)

Partie 2: Structures de Kripke

Modèles

On étudie un modèle très générique, les systèmes de transitions (ST):

$$\mathcal{T} = (S, \rightarrow, r)$$

S	$\widehat{=}$	espace d'états; les états que le système peut atteindre (ensemble fini ou infini)
$\rightarrow \subseteq S \times S$	$\widehat{=}$	relation de transitions; décrit les actions possibles
$r \in S$	$\widehat{=}$	état initial ("racine")

Exemple 1: Producteur/Consummateur

Pseudocode avec variables et concurrence:

var *turn* {0,1} **init** 0;
cobegin {*P* ‖ *K*} **coend**

P =

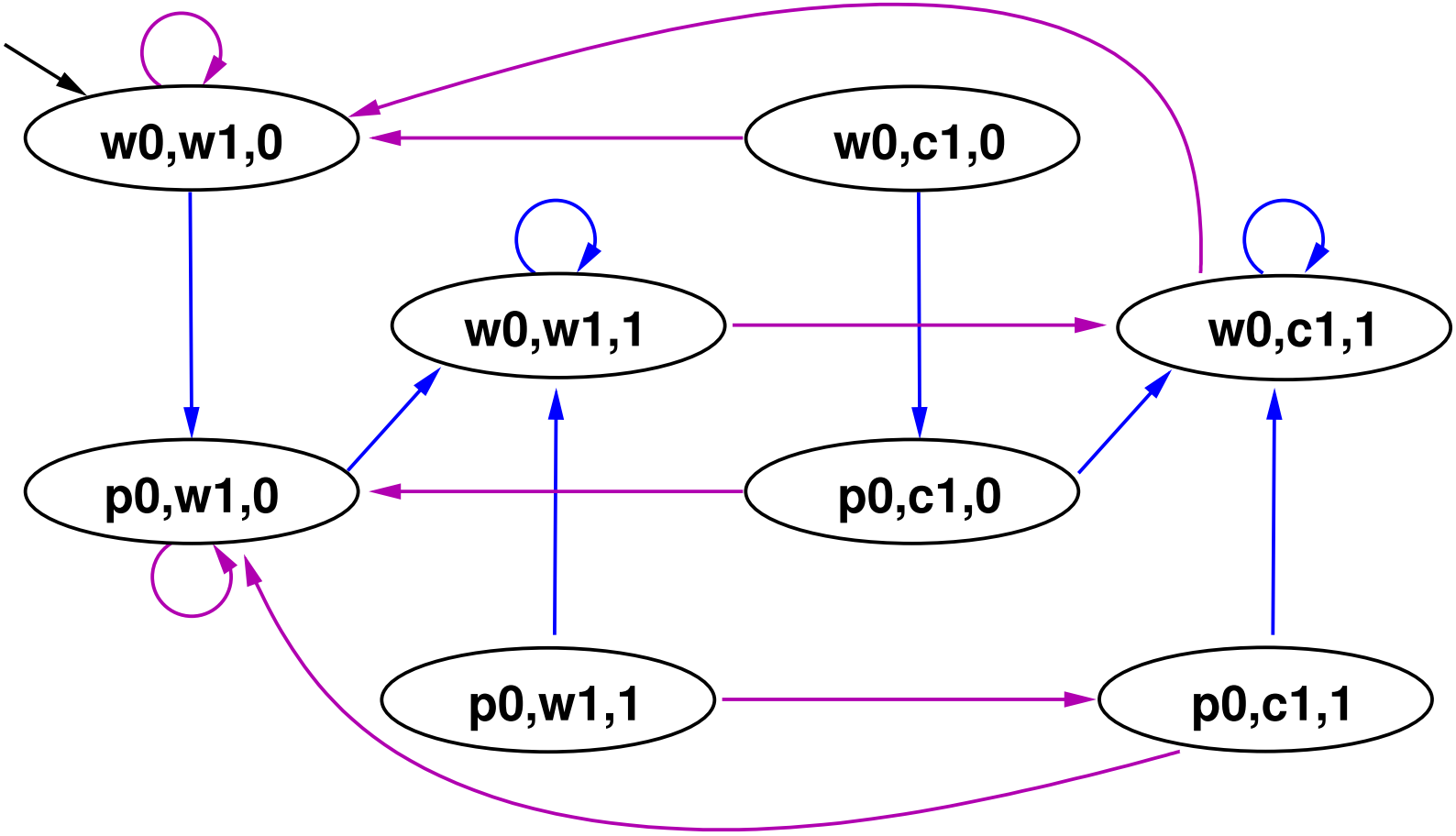
start;
while true do
  w_0: **wait** $(turn = 0)$;
  p_0: /* produce */
    $turn := 1$;
od;
end

K =

start;
while true do
  w_1: **wait** $(turn = 1)$;
  c_1: /* consume */
    $turn := 0$;
od;
end

Exemple 1: ST correspondant

$S = \{w_0, p_0\} \times \{w_1, c_1\} \times \{0, 1\}$; racine $(w_0, w_1, 0)$



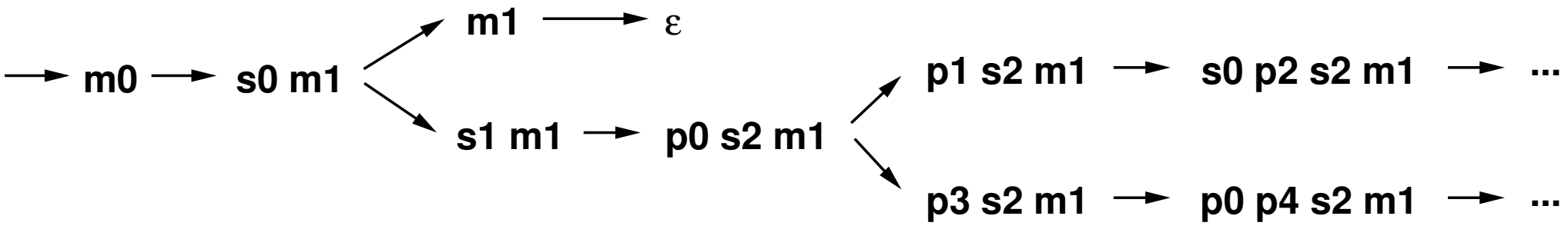
Exemple 2: Programme recursif

procedure p;
p_0: **if** ? **then**
p_1: **call** s;
p_2: **if** ? **then call** p; **end if**;
else
p_3: **call** p;
end if
p_4: **return**

procedure s;
s_0: **if** ? **then return**; **end if**;
s_1: **call** p;
s_2: **return**;

procedure $main$;
m_0: **call** s;
m_1: **return**;

$S = \{p_0, \ldots, p_4, s_0, \ldots, s_2, m_0, m_1\}^*$, racine m_0



Notations pour ST

On écrit $s \to t$ si $(s, t) \in \to$.

Si $s \to t$ alors s s'appelle prédecesseur direct de t et t successeur direct de s.

S^* dénote les séquences (mots) *finis*, S^ω les mots *infinis* sur S.

$w = s_0 \dots s_n$ est un chemin de longueur n si $s_i \to s_{i+1}$ pour tout $0 \leq i < n$.

$\rho = s_0 s_1 \dots$ est un chemin infini si $s_i \to s_{i+1}$ pour tout $i \geq 0$.

Notation pour ST II

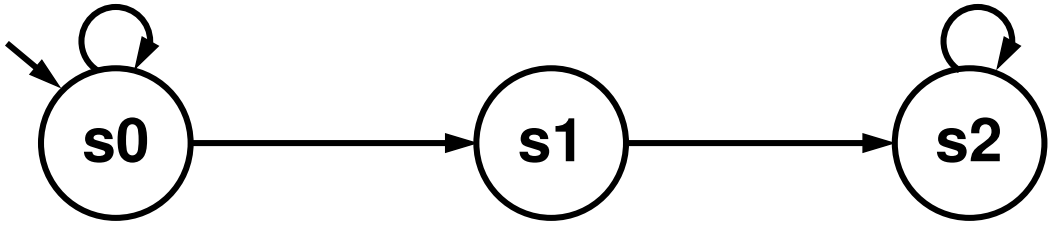
$\rho(i)$ dénote le i-ème élément de ρ et ρ^i le suffixe partant de $\rho(i)$.

$s \rightarrow^* t$ s'il existe un chemin de s à t.

$s \rightarrow^+ t$ s'il existe un tel chemin de longueur au moins 1.

Si $s \rightarrow^* t$ alors s est un predecesseur de t et t un successeur de s.

Exemple



$S = \{s_0, s_1, s_2\}$; racine s_0

$s_0 \to s_0$ $s_0 \to s_1$ $s_1 \to s_2$ $s_2 \to s_2$

$s_0 s_1 s_2$ est un chemin de longueur 2, $s_0 \to^* s_2$ et $s_0 \to^+ s_2$

$s_1 \to^* s_1$ mais $s_1 \not\to^+ s_1$

$\rho = s_0 s_0 s_1 s_2 s_2 s_2 \ldots$ est un chemin infini.

$\rho(2) = s_1$ $\rho^1 = s_0 s_1 s_2 s_2 s_2 \ldots$

ST finis et infinis

Plusieurs raisons font qu'un ST peut être infini:

- Données: entiers, réels, lists, arbres, pointeurs, ...
- Contrôle: récursion, création de threads dynamique ...
- Communication: canaux FIFO ...
- Paramètres: nombre de participants dans un protocôle ...
- Temps réel: continu ou discret

Certains (pas tout!) de ces caractéristics donnent lieu à des problèmes de vérification indécidables. Ici, on se concentrera sur les systèmes à états finis.

Espaces d'états fini

Systèmes finis: p.ex. circuits, programmes booléens, . . .

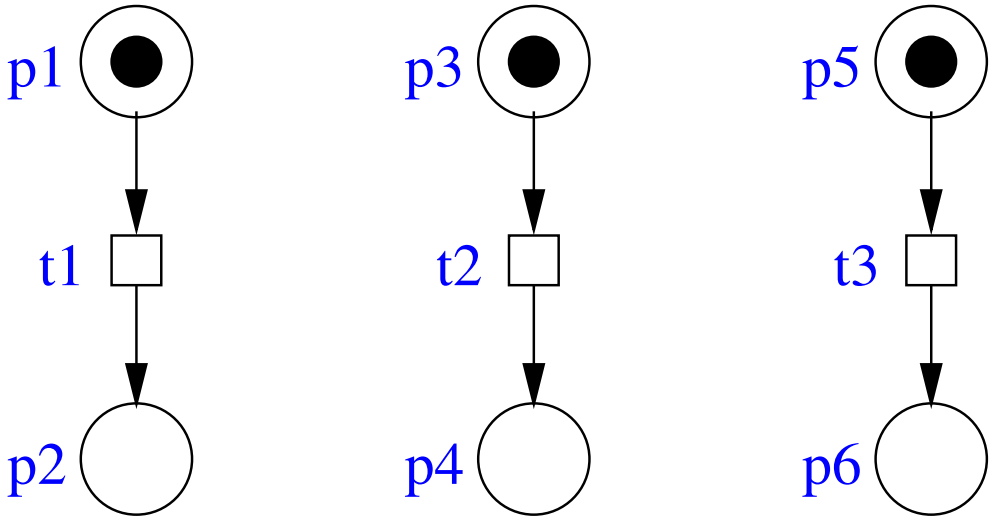
Systèmes obtenus par abstraction fini d'un système infini.

Problème d'explosion d'états : un systèms peut être fini mais TRÈS large.

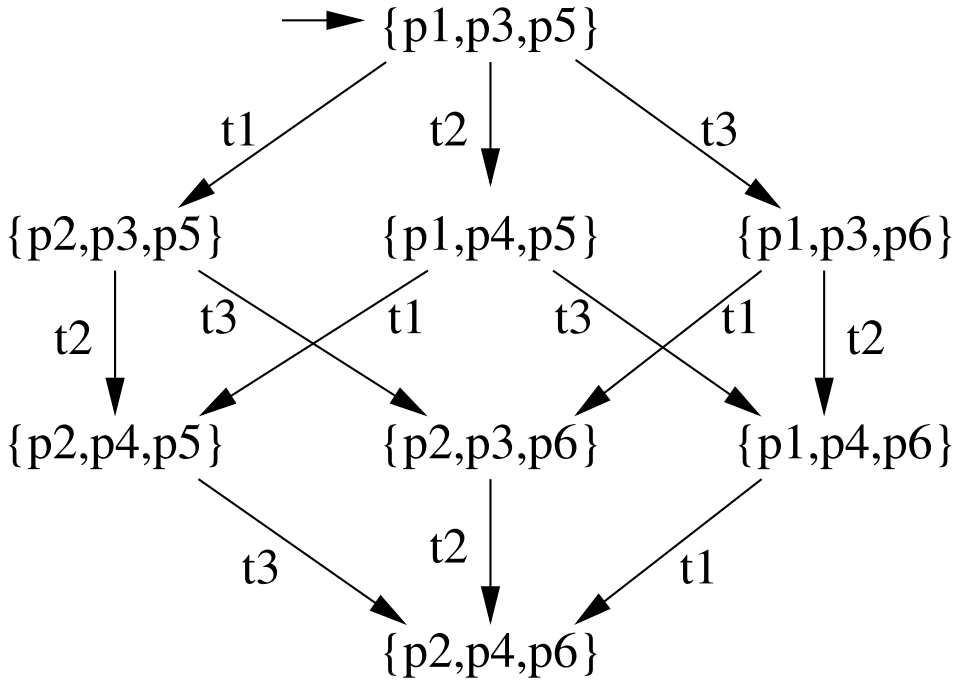
Raisons pour explosion d'états (I)

Une raison en est la concurrence.

Exemple: Réseau de Petri:



On a $8 = 2^3$ états et $6 = 3!$ chemins.



Avec n composants on aurait 2^n états et $n!$ chemins.

Raisons pour explosion d'états (2)

Les données.

nombre d'états : 2^n, où n est le nombre de bits

Il existe plusieurs méthodes pour contrer ces effects, on en étudiera une ou deux.

Structures de Kripke (SK)

Idée: Extraire des affectations de chaque état:

$$\mathcal{K} = (S, \rightarrow, r, AP, \nu)$$

$(S, \rightarrow, r)$	$\widehat{=}$	le ST sous-jacent
AP	$\widehat{=}$	ensemble de prédicats
$\nu : S \rightarrow 2^{AP}$	$\widehat{=}$	Interprétation des prédicats

Remarques:

- 2^{AP} dénote les *parties* de AP.
- On représente une affectation par le sous-ensemble des prédicats vrais.

Exemple d'une SK

ST $(S, \rightarrow, r)$ comme dans l'Exemple 1.

Supposons qu'on s'intéresse aux actions de production et consommation:

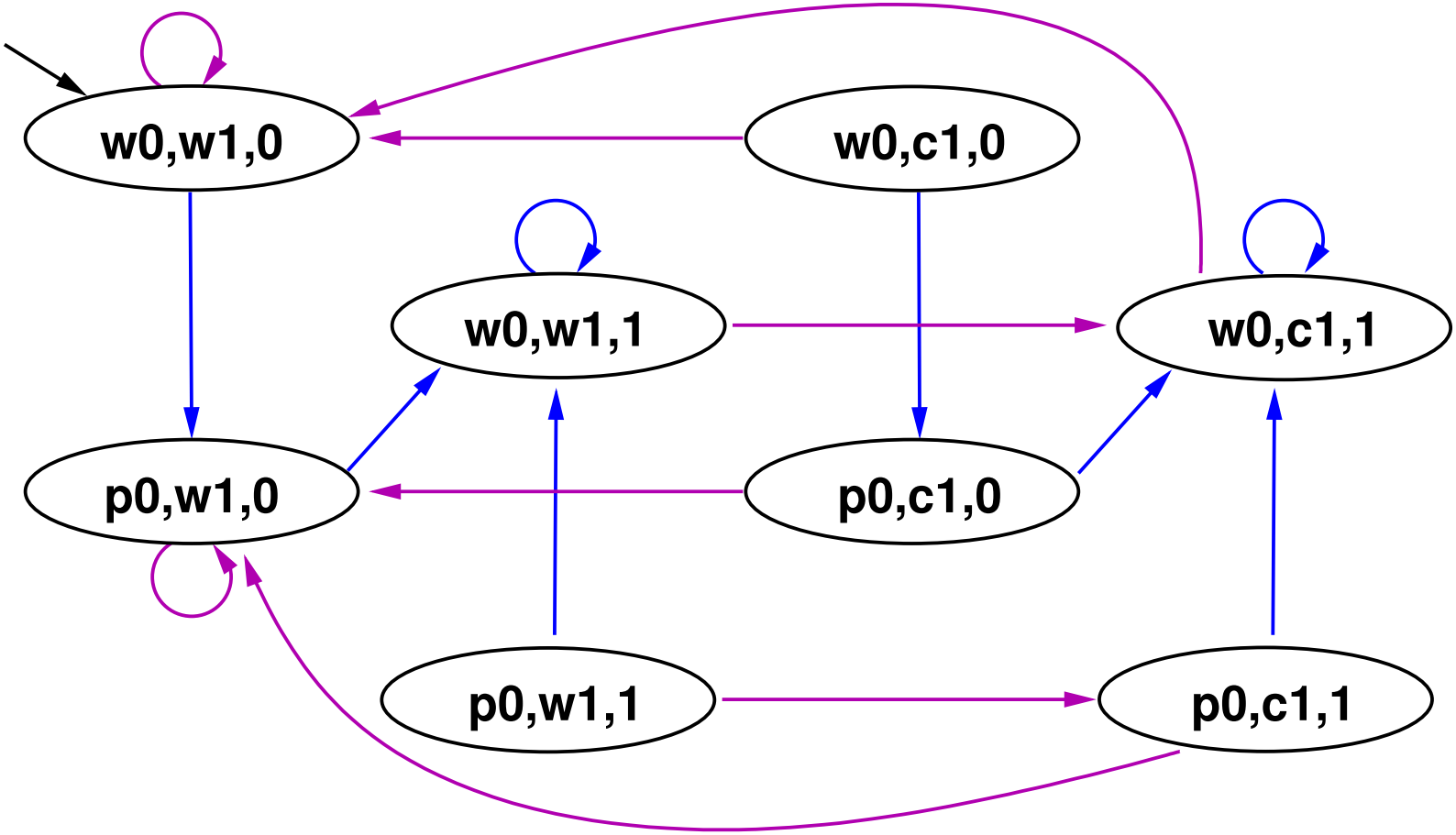
Soit $AP = \{prod, cons\}$;

$\nu^{-1}(prod) = \{p_0\} \times \{w_1, c_1\} \times \{0, 1\}$;

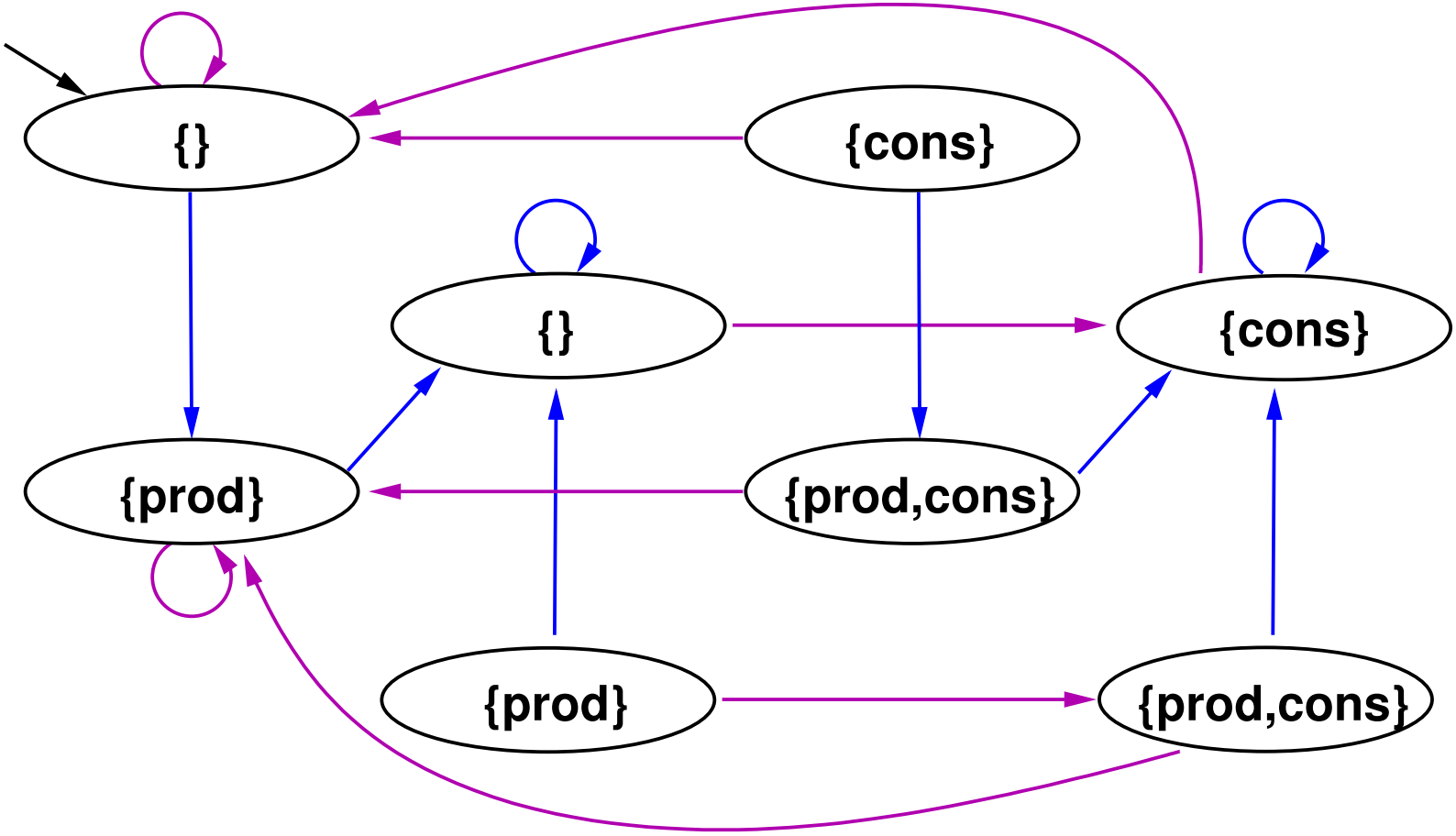
$\nu^{-1}(cons) = \{w_0, p_0\} \times \{c_1\} \times \{0, 1\}$.

Rappel: Exemple 1

Dans l'Exemple 1, . . .



. . . les affectations sont ainsi :

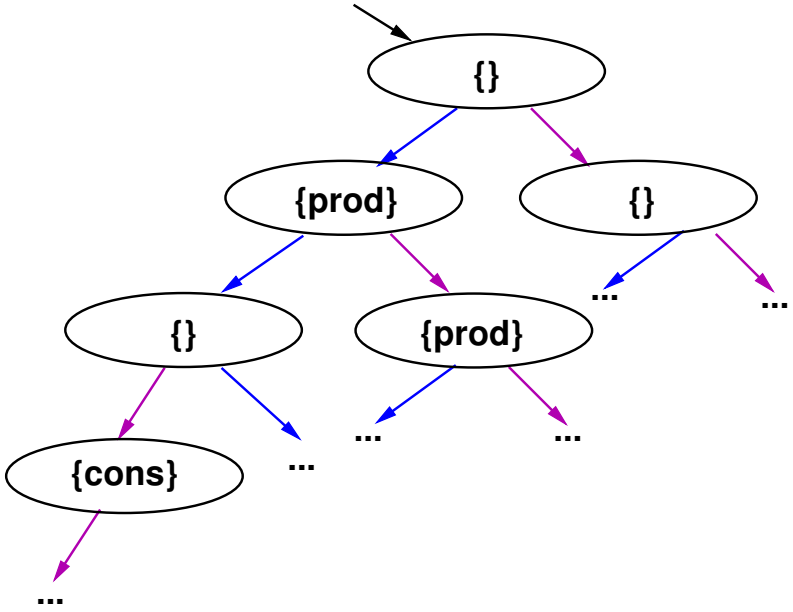


Séquences et arbres

Dans la logique linéaire, on considère les séquences :

p.ex. $\emptyset\ \emptyset\ \{prod\}\ \emptyset\ \{cons\}\ \ldots$ ou $\emptyset\ \{prod\}\ \{prod\}\ \{prod\}\ \ldots$

Dans la logique branchante on considère l'arbre des exécutions :



Exemples de propriétés

“*prod* et *cons* ne sont jamais vrais en même temps.”

(exemple d’une invariante)

“Après une production il peut y avoir une consommation.”

(exemple d’une propriété de vivacité)

Partie 3: Logique temporelle linéaire

Préliminaires

Idée: le temps progress de façon discrète et linéaire, chaque moment possède un seul successeur dans le futur

origines dans la philosophie et la logique

Exemple le plus prominent : LTL

- utilisé pour la vérification depuis les années 1970

Syntaxe de LTL

Soit AP un ensemble de prédicats. Les formules de LTL sur AP sont définies comme suit :

Si $p \in AP$ alors p est une formule.

Si ϕ_1, ϕ_2 sont des formules, alors aussi les suivants:

$$\neg\phi_1, \qquad \phi_1 \vee \phi_2, \qquad \mathbf{X}\,\phi_1, \qquad \phi_1\,\mathbf{U}\,\phi_2$$

Intuition: $\mathbf{X} \mathrel{\widehat{=}}$ “next” (prochain), $\mathbf{U} \mathrel{\widehat{=}}$ “until” (jusqu’à).

Remarques

C'est une syntaxe minimale qu'on utilisera pour des preuves.

Pour plus d'expressivité, on définit quelques raccourcis:

Comparaison de logique propositionnelle (LP) and LTL:

	LP	LTL
Syntaxe	prédicats, opérateurs logiques	+ modalités temporelles
Évaluée sur...	affectations	séquences d'affectations
Semantique	ensemble d'affectations	ensemble de séquences

Semantique de LTL

Soit ϕ une formule de LTL formula et σ une séquence d'affectations.
On écrit $\sigma \models \phi$ pour "σ satisfait ϕ."

$$
\begin{array}{lll}
\sigma \models & p & \text{if } p \in AP \text{ and } p \in \sigma(0) \\
\sigma \models & \neg\phi & \text{if } \sigma \not\models \phi \\
\sigma \models & \phi_1 \vee \phi_2 & \text{if } \sigma \models \phi_1 \text{ or } \sigma \models \phi_2 \\
\sigma \models & \mathbf{X}\,\phi & \text{if } \sigma^1 \models \phi \\
\sigma \models & \phi_1 \,\mathbf{U}\, \phi_2 & \text{if } \exists i : \left(\sigma^i \models \phi_2 \ \wedge \ \forall k < i : \sigma^k \models \phi_1\right)
\end{array}
$$

Semantique de ϕ: $\quad [\![\phi]\!] = \{\, \sigma \mid \sigma \models \phi \,\}$

Exemples

Soit $AP = \{p, q, r\}$. Trouver si la séquence

$$\sigma = \{p\}\ \{q\}\ \{p\}^{\omega}$$

satisfait les formules suivantes :

$$p$$

$$q$$

$$\mathbf{X}\, q$$

$$\mathbf{X}\, \neg p$$

$$p\, \mathbf{U}\, q$$

$$q\, \mathbf{U}\, p$$

$$(p \vee q)\, \mathbf{U}\, r$$

Raccourcis

On utilisera les définitions suivantes :

$$\begin{aligned}
\phi_1 \wedge \phi_2 &\equiv \neg(\neg\phi_1 \vee \neg\phi_2) & \qquad \mathbf{F}\,\phi &\equiv \mathbf{true}\,\mathbf{U}\,\phi \\
\phi_1 \rightarrow \phi_2 &\equiv \neg\phi_1 \vee \phi_2 & \qquad \mathbf{G}\,\phi &\equiv \neg\,\mathbf{F}\,\neg\phi \\
\mathbf{true} &\equiv a \vee \neg a & \qquad \phi_1\,\mathbf{W}\,\phi_2 &\equiv (\phi_1\,\mathbf{U}\,\phi_2) \vee \mathbf{G}\,\phi_1 \\
\mathbf{false} &\equiv \neg\mathbf{true} & \qquad \phi_1\,\mathbf{R}\,\phi_2 &\equiv \neg(\neg\phi_1\,\mathbf{U}\,\neg\phi_2)
\end{aligned}$$

Signification: $\mathbf{F} \mathrel{\widehat{=}}$ “finalement”, $\mathbf{G} \mathrel{\widehat{=}}$ “globalement” (toujours),
$\mathbf{W} \mathrel{\widehat{=}}$ “weak until”, $\mathbf{R} \mathrel{\widehat{=}}$ “release”.

Des exemples

Invariant: $\mathbf{G}\,\neg(cs_1 \wedge cs_2)$

cs_1 et cs_2 ne sont jamais vrais en même temps.

Sûreté: $(\neg x)\ \mathbf{W}\ y$

x n'apparaît pas avant y

Remarque: Si y n'apparaît jamais, alors x n'apparaît non plus.

Vivacité: $(\neg x)\ \mathbf{U}\ y$

x n'apparaît pas avant y et y apparaît sûrement.

Des exemples

$\mathbf{G}\,\mathbf{F}\,p$

p apparaît infiniment souvent.

$\mathbf{F}\,\mathbf{G}\,p$

À partir d'un moment, *p* tient toujours.

$\mathbf{G}(try_1 \rightarrow \mathbf{F}\,cs_1)$

Pour exclusion mutuelle: Si processus 1 essaie d'entrer dans la zone critique, il y parviendra.

Tautologie, équivalence

Tautologie: formule ϕ avec $[\![\phi]\!] = (2^{AP})^\omega$

Insatisfaisable: formule ϕ avec $[\![\phi]\!] = \emptyset$

Équivalence: formules ϕ_1, ϕ_2 avec iff $[\![\phi_1]\!] = [\![\phi_2]\!]$.
Notation: $\phi_1 \equiv \phi_2$

Équivalences: relations entre modalités

$$
\begin{aligned}
\mathbf{X}(\phi_1 \vee \phi_2) &\equiv \mathbf{X}\,\phi_1 \vee \mathbf{X}\,\phi_2 \\
\mathbf{X}(\phi_1 \wedge \phi_2) &\equiv \mathbf{X}\,\phi_1 \wedge \mathbf{X}\,\phi_2 \\
\mathbf{X}\,\neg\phi &\equiv \neg\,\mathbf{X}\,\phi \\
\\
\mathbf{F}(\phi_1 \vee \phi_2) &\equiv \mathbf{F}\,\phi_1 \vee \mathbf{F}\,\phi_2 \\
\neg\,\mathbf{F}\,\phi &\equiv \mathbf{G}\,\neg\phi \\
\\
\mathbf{G}(\phi_1 \wedge \phi_2) &\equiv \mathbf{G}\,\phi_1 \wedge \mathbf{G}\,\phi_2 \\
\neg\,\mathbf{G}\,\phi &\equiv \mathbf{F}\,\neg\phi \\
\\
(\phi_1 \wedge \phi_2)\,\mathbf{U}\,\psi &\equiv (\phi_1\,\mathbf{U}\,\psi) \wedge (\phi_2\,\mathbf{U}\,\psi) \\
\phi\,\mathbf{U}\,(\psi_1 \vee \psi_2) &\equiv (\phi\,\mathbf{U}\,\psi_1) \vee (\phi_1\,\mathbf{U}\,\psi_2)
\end{aligned}
$$

Équivalences: idempotence et recursion

$$\mathbf{F}\,\phi \equiv \mathbf{F}\,\mathbf{F}\,\phi$$

$$\mathbf{G}\,\phi \equiv \mathbf{G}\,\mathbf{G}\,\phi$$

$$\phi\,\mathbf{U}\,\psi \equiv \phi\,\mathbf{U}\,(\phi\,\mathbf{U}\,\psi)$$

$$\mathbf{F}\,\phi \equiv \phi \vee \mathbf{X}\,\mathbf{F}\,\phi$$

$$\mathbf{G}\,\phi \equiv \phi \wedge \mathbf{X}\,\mathbf{G}\,\phi$$

$$\phi\,\mathbf{U}\,\psi \equiv \psi \vee (\phi \wedge \mathbf{X}(\phi\,\mathbf{U}\,\psi))$$

$$\phi\,\mathbf{W}\,\psi \equiv \psi \vee (\phi \wedge \mathbf{X}(\phi\,\mathbf{W}\,\psi))$$

Interprétation de LTL sur une SK

Soit $\mathcal{K} = (S, \rightarrow, r, AP, \nu)$ une SK.
On s'intéresse aux séquence générées par $\mathcal{K}$.

Soit $\rho \in S^\omega$ un chemin infini dans $\mathcal{K}$.

On affecte à ρ son "image" $\nu(\rho)$ dans $(2^{AP})^\omega$; pour tout $i \geq 0$ soit

$$\nu(\rho)(i) = \nu(\rho(i))$$

alors $\nu(\rho)$ est la séquence d'affectations correspondante.

On note $[\![\mathcal{K}]\!]$ l'ensemble de ces séquences :

$$[\![\mathcal{K}]\!] = \{\, \nu(\rho) \mid \rho \text{ is an infinite path of } \mathcal{K} \,\}$$

Le problème de model-checking pour LTL

Problème: Étant donné une SK $\mathcal{K} = (S, \rightarrow, r, AP, \nu)$ et une formula de LTL ϕ sur AP, tester si $[\![\mathcal{K}]\!] \subseteq [\![\phi]\!]$.

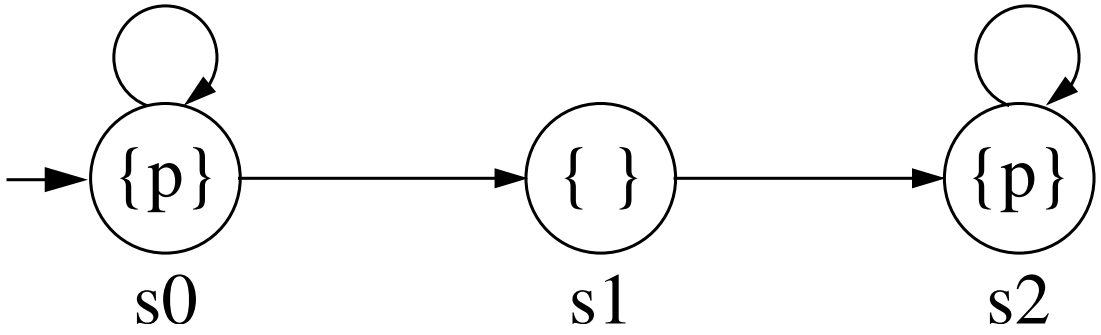
Définition: Si $[\![\mathcal{K}]\!] \subseteq [\![\phi]\!]$ alors on écrit $\mathcal{K} \models \phi$.

Interprétation: Toute exécution de $\mathcal{K}$ satisfait ϕ.

Remarque: Il est possible que have $\mathcal{K} \not\models \phi$ et $\mathcal{K} \not\models \neg\phi$!

Exemple

On considère la SK suivante $\mathcal{K}$ avec $AP = \{p\}$:



Il y a deux espèces de chemins infinis dans $\mathcal{K}$:

(i) Soit le système reste dans s_0 à jamais,

(ii) soit il parvient à s_2 via s_1.

On a:

$\mathcal{K} \models \mathbf{F}\,\mathbf{G}\,p$

$\mathcal{K} \not\models \mathbf{G}\,p$

Partie 4: Automates de Büchi

Contenu

Problème de model-checking: $[\![\mathcal{K}]\!] \subseteq [\![\phi]\!]$ – comment tester algorithmiquement?

On utilise la théorie des langages formels : $[\![\mathcal{K}]\!]$ et $[\![\phi]\!]$ sont des langages (de mots infinis).

Trouver une classe d'automates pour représenter ces langages.

Définir des opérations utiles sur ces automates pour résoudre le problème.

Automates de Büchi

Un automate de Büchi (AB) est un tuple $\mathcal{B} = (\Sigma, S, s_0, \Delta, F)$, où:

Σ	est un alphabet fini;
S	est un ensemble fini d'états;
$s_0 \in S$	est un état initial;
$\Delta \subseteq S \times \Sigma \times S$	sont des transitions;
$F \subseteq S$	sont des états acceptants.

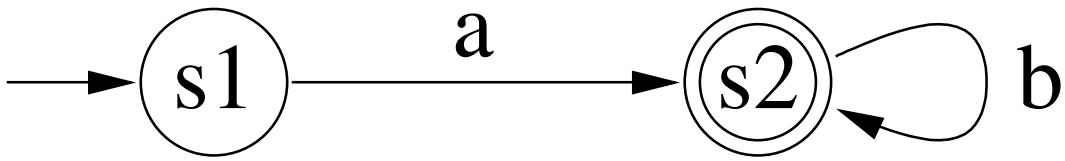
Remarques:

La définition et représentation graphique est identique aux automates finis.

Par contre, les AB travaillent sur les mots *infinis*, ce qui nécessite une condition d'acceptation différente.

Exemple

Répresentation d'un AB :



Les éléments de cet automate sont $(\Sigma, S, s_1, \Delta, F)$, avec:

- $\Sigma = \{a, b\}$ (symboles sur les arêtes)
- $S = \{s_1, s_2\}$ (cercles)
- s_1 (indiqué par une flèche entrante)
- $\Delta = \{(s_1, a, s_2), (s_2, b, s_2)\}$ (arêtes)
- $F = \{s_2\}$ (avec cercle double)

Langage d'un AB

Soit $\mathcal{B} = (\Sigma, S, s_0, \Delta, F)$ un automate de Büchi.

Un calcul de $\mathcal{B}$ sur un mot infini $\sigma \in \Sigma^\omega$ est une séquence infinie d'états $\rho \in S^\omega$ telle que $\rho(0) = s_0$ et $(\rho(i), \sigma(i), \rho(i+1)) \in \Delta$ pour tout $i \geq 0$.

On appelle ρ acceptant si $\rho(i) \in F$ pour un nombre infini de i.

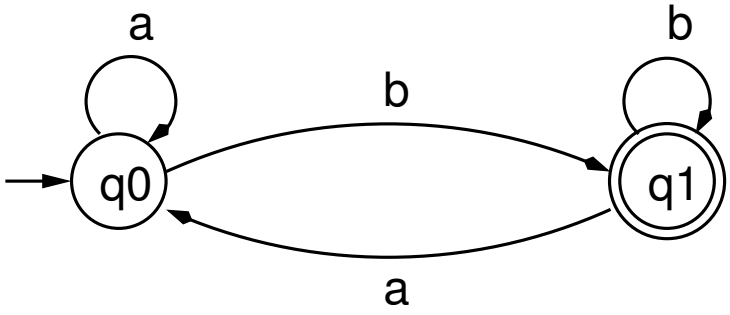
> Du coup, ρ contient infiniment souvent un état acceptant. Par le principe de tiroirs, il y a donc au moins un état acceptant qui est visité infiniment souvent.

$\sigma \in \Sigma^\omega$ est accepté par $\mathcal{B}$ s'il existe un calcul acceptant sur σ dans $\mathcal{B}$.

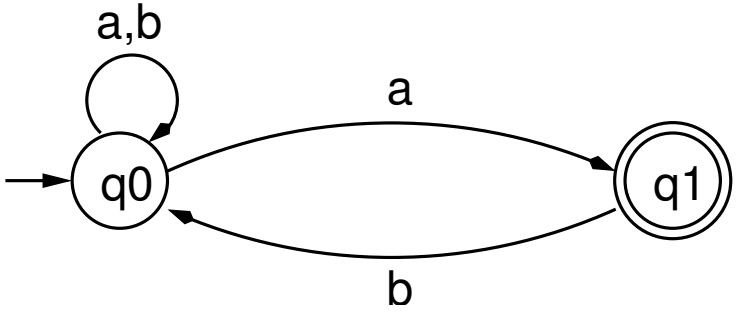
Le langage de $\mathcal{B}$, noté $\mathcal{L}(\mathcal{B})$, est l'ensemble de mots acceptés par $\mathcal{B}$.

Automates de Büchi: exemples

"infiniment souvent b"



"infiniment souvent ab"



Les automates de Büchi et LTL

Soit AP un ensemble de prédicats.

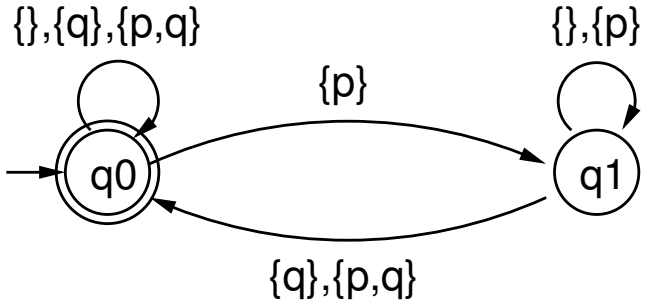
Un AB avec alphabet 2^{AP} accepte une séquence d'affectations.

Lemme: Pour toute formule de LTL il existe un AB $\mathcal{B}$ tel que $\mathcal{L}(\mathcal{B}) = [\![\phi]\!]$.

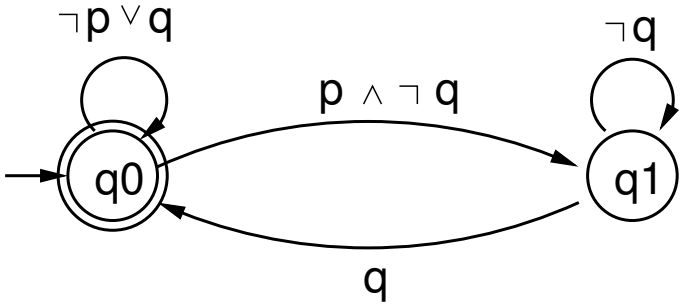
(On prouvera ce lemme plus tard.)

Exemples: $\mathbf{F}\, p$, $\mathbf{G}\, p$, $\mathbf{G}\,\mathbf{F}\, p$, $\mathbf{G}(p \rightarrow \mathbf{F}\, q)$, $\mathbf{F}\,\mathbf{G}\, p$

AB pour $\mathbf{G}(p \rightarrow \mathbf{F}\, q)$, avec $AP = \{p, q\}$.



On se permet d'étiqueter les arêtes avec des formules; dans ce cas, une formule F est un raccourci pour toutes les modèles de $[\![F]\!]$.



Opérations sur les AB

Les langages acceptés par des AB s'appellent ω-reguliers ou ω-reconnaissables.

Tout comme pour les langages reconnaissables sur mots finis, on a des propriétés de fermeture:

Si $\mathcal{L}_1$ et $\mathcal{L}_2$ sont ω-reconnaissables, alors les suivants le sont aussi :

$$\mathcal{L}_1 \cup \mathcal{L}_2, \qquad \mathcal{L}_1 \cap \mathcal{L}_2, \qquad \mathcal{L}_1^c.$$

On étudiera des opérations pour construire des AB pour certains cas.

Sur les transparents suivants on prend $\mathcal{B}_1 = (\Sigma, S, s_0, \Delta_1, F)$ et $\mathcal{B}_2 = (\Sigma, T, t_0, \Delta_2, G)$ (avec $S \cap T = \emptyset$).

Union

"Juxtaposer" $\mathcal{B}_1$ et $\mathcal{B}_2$, et ajouter un nouvel état initial.

Autrement dit, l'AB $\mathcal{B} = (\Sigma,\ S \cup T \cup \{u\},\ u,\ \Delta_1 \cup \Delta_2 \cup \Delta_u,\ F \cup G)$
accepte $\mathcal{L}(\mathcal{B}_1) \cup \mathcal{L}(\mathcal{B}_2)$, où

u est un état "frais" ($u \notin S \cup T$);

$\Delta_u = \{\, (u, \sigma, s) \mid (s_0, \sigma, s) \in \Delta_1 \,\} \cup \{\, (u, \sigma, t) \mid (t_0, \sigma, t) \in \Delta_2 \,\}$.

Intersection I (cas spécial)

On considère d'abord le cas ou tous les états de $\mathcal{B}_2$ acceptent ($G = T$).

Idée: Construire un automate de produit, tester si on visite F infiniment souvent.

Soit $\mathcal{B} = (\Sigma,\ S \times T,\ \langle s_0, t_0\rangle,\ \Delta,\ F \times T)$, avec

$$\Delta = \{\, (\langle s, t\rangle, a, \langle s', t'\rangle) \mid a \in \Sigma,\ (s, a, s') \in \Delta_1,\ (t, a, t') \in \Delta_2 \,\}.$$

Alors: $\mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{B}_1) \cap \mathcal{L}(\mathcal{B}_2)$.

Intersection II (cas général)

Problème: La condition d'acceptance doit tester si *les deux* conditions d'acceptance sont satisfaites, potentiellement à des instants différents.

Idée: construire deux copies de l'automate de produit.

- La première copie attend un état de F.
- La seconde copie attend un état de G.
- Ayant trouvé un tel état, on bascule entre les deux copies.
- La condition d'acceptance assure qu'on bascule infiniment souvent.

Soit $\mathcal{B} = (\Sigma, U, u, \Delta, H)$, où

$$U = S \times T \times \{1,2\}, \quad u = \langle s_0, t_0, 1\rangle, \quad H = F \times T \times \{1\}$$

$$(\langle s,t,1\rangle, a, \langle s',t',1\rangle) \in \Delta \text{ iff } (s,a,s') \in \Delta_1,\ (t,a,t') \in \Delta_2,\ s \notin F$$
$$(\langle s,t,1\rangle, a, \langle s',t',2\rangle) \in \Delta \text{ iff } (s,a,s') \in \Delta_1,\ (t,a,t') \in \Delta_2,\ s \in F$$
$$(\langle s,t,2\rangle, a, \langle s',t',2\rangle) \in \Delta \text{ iff } (s,a,s') \in \Delta_1,\ (t,a,t') \in \Delta_2,\ t \notin G$$
$$(\langle s,t,2\rangle, a, \langle s',t',1\rangle) \in \Delta \text{ iff } (s,a,s') \in \Delta_1,\ (t,a,t') \in \Delta_2,\ t \in G$$

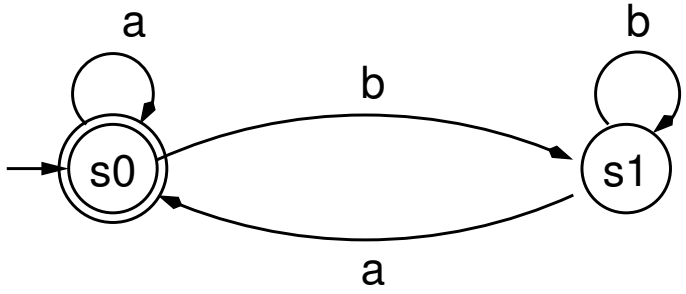
Remarques:

On démarre dans la première copie.

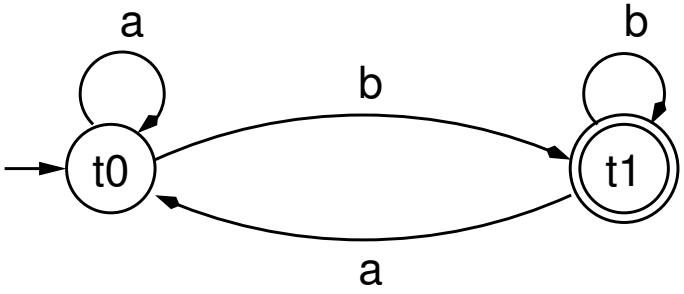
$S \times G \times \{2\}$ marche aussi comme condition d'acceptane.

On peut généraliser la construction à l'intersection entre n automates.

Intersection: exemple

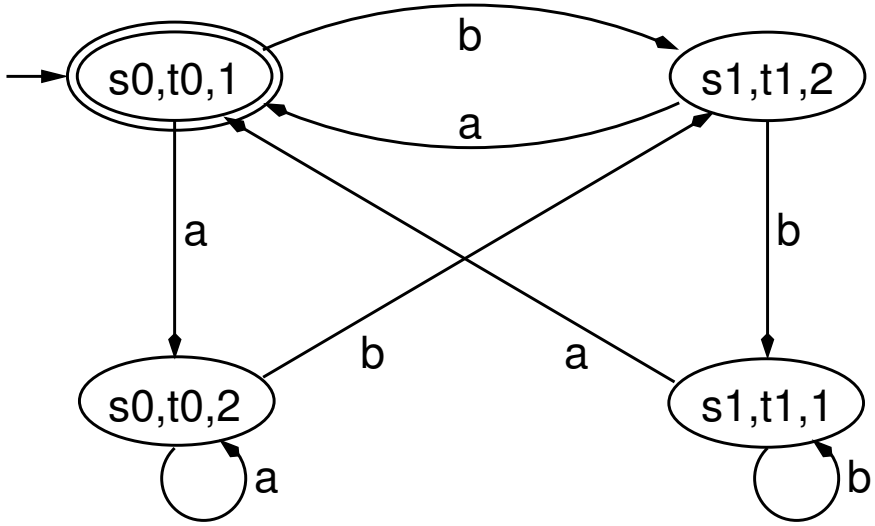


B1



B2

B1 x B2



Complément

Problème: Étant donné $\mathcal{B}_1$, construire $\mathcal{B}$ avec $\mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{B}_1)^c$.

Une telle construction est possible (mais compliqué). On n'en a pas besoin pour ce cours.

Littérature:

Wolfgang Thomas, *Automata on Infinite Objects*,
Chapter 4 in *Handbook of Theoretical Computer Science*

Igor Walukiewicz, lecture notes on *Automata and Logic*, chapter 3,
`www.labri.fr/perso/igw/Papers/igw-eefss01.pdf`

Grädel, Thomas, Wilke (eds.), *Automata, Logics, and Infinite Games*,
LNCS 2500, chapîtres 3 et 4.

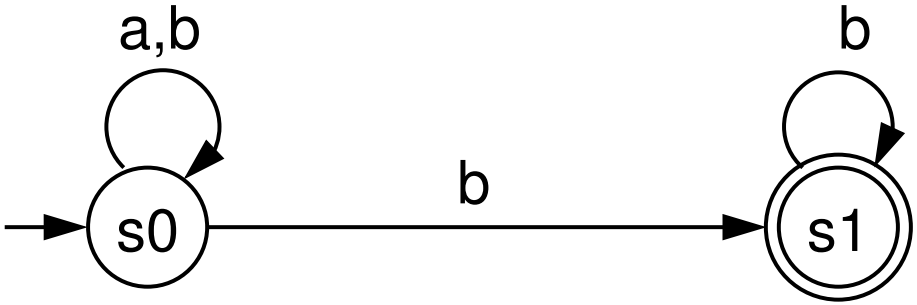
AB déterministes

Dans les cas des automates finis, il est bien connu que les automates finis *déterministes* sont aussi expressifs que les non-déterministes.

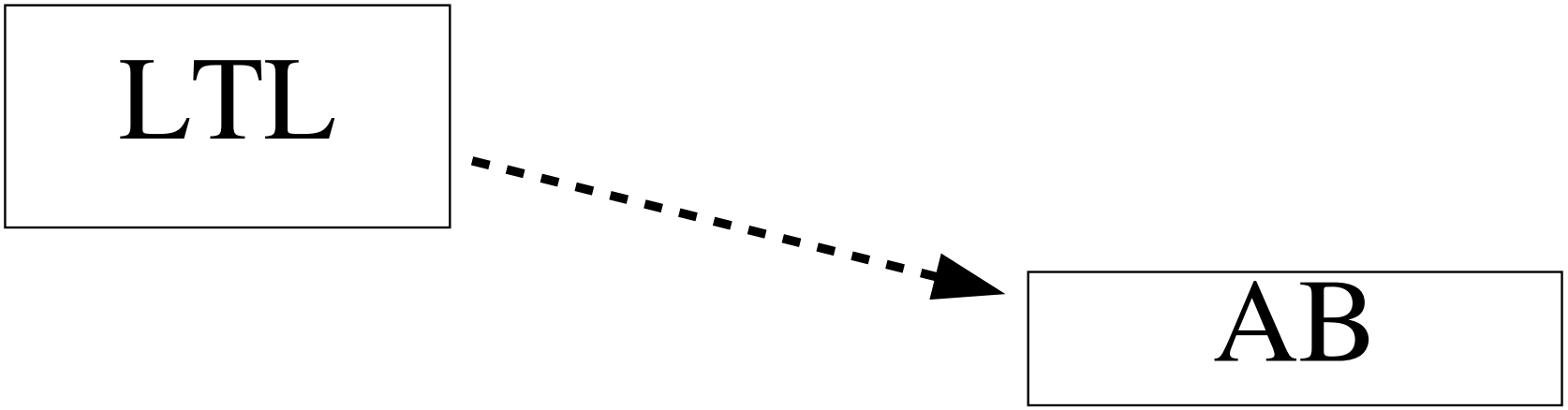
Ceci n'est pas vrai pour les automates de Büchi.

Contre-exemple : il n'y a pas d'AB déterministe équivalent à l'AB ci-dessous:

"Nombre fini de *a*."

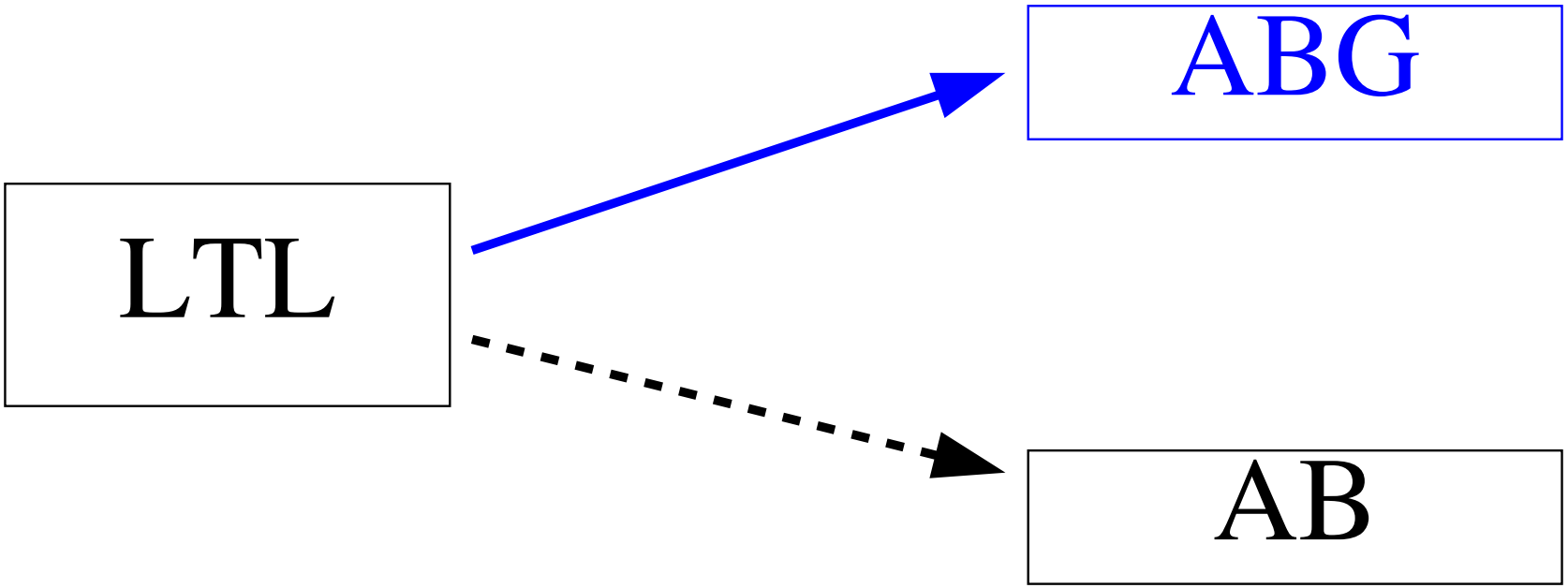


Aperçu



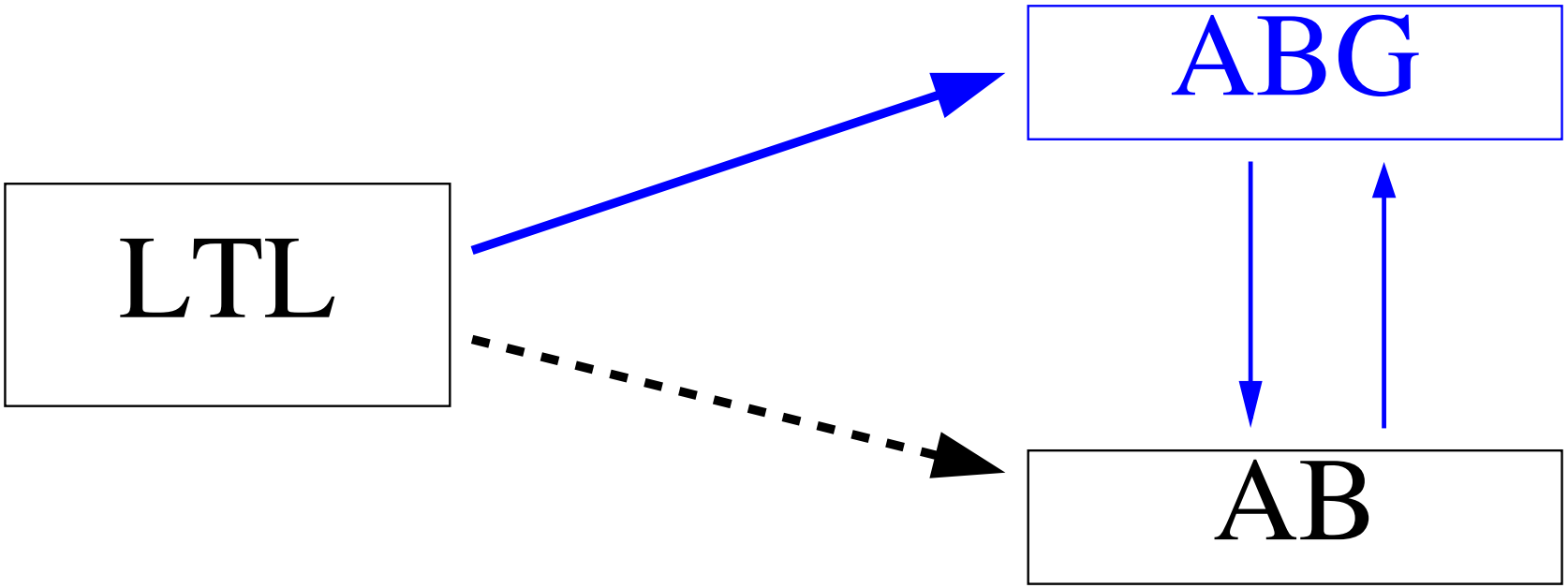
Nous souhaitons traduire les formules de LTL en automate de Büchi.

Aperçu



Détour: On va les traduire en *automates de Büchi généralisés* (ABG).

Aperçu



Les ABG accepteront les même langages que les AB.

Resoudre le problème de model-checking pour LTL

Étant donné une SK $\mathcal{K} = (S, \rightarrow, r, AP, \nu)$ et une formule de LTL ϕ sur AP, on demande si $\mathcal{K} \models \phi$.

Solution:

On interprète $\mathcal{K}$ comme un AB $\mathcal{B}_\mathcal{K}$:

$\mathcal{B}_\mathcal{K} = (2^{AP}, S, r, \Delta, S)$, où $\Delta = \{ (s, \nu(s), t) \mid s \rightarrow t \}$

Évidemment, $[\![\mathcal{K}]\!] = \mathcal{L}(\mathcal{B}_\mathcal{K})$.

On traduit $\neg\phi$ en un AB $\mathcal{B}_{\neg\phi}$.

On a alors:

$$\begin{array}{rrcl} & \mathcal{K} & \models & \phi \\ \iff & [\![\mathcal{K}]\!] & \subseteq & [\![\phi]\!] \\ \iff & [\![\mathcal{K}]\!] & \cap & [\![\neg\phi]\!] = \emptyset \\ \iff & \mathcal{L}(\mathcal{B}_{\mathcal{K}}) & \cap & \mathcal{L}(\mathcal{B}_{\neg\phi}) = \emptyset \end{array}$$

Du coup:

On construit les AB $\mathcal{B}_{\mathcal{K}}$ et $\mathcal{B}_{\neg\phi}$.

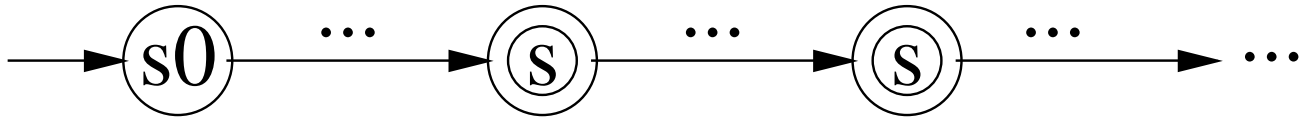
On les intersecte en utilisant le cas spécial car tous les états de $\mathcal{B}_{\mathcal{K}}$ sont acceptants.

Le problème de model-checking se reduit alors au problème du vide pour cette intersection.

Problème du vide

Problème: Étant donné un AB $\mathcal{B}$, tester si $\mathcal{L}(\mathcal{B}) = \emptyset$.

Observation: $\mathcal{L}(\mathcal{B}) \neq \emptyset$ ssi il existe $s \in F$ tel que $s_0 \rightarrow^* s \rightarrow^+ s$.



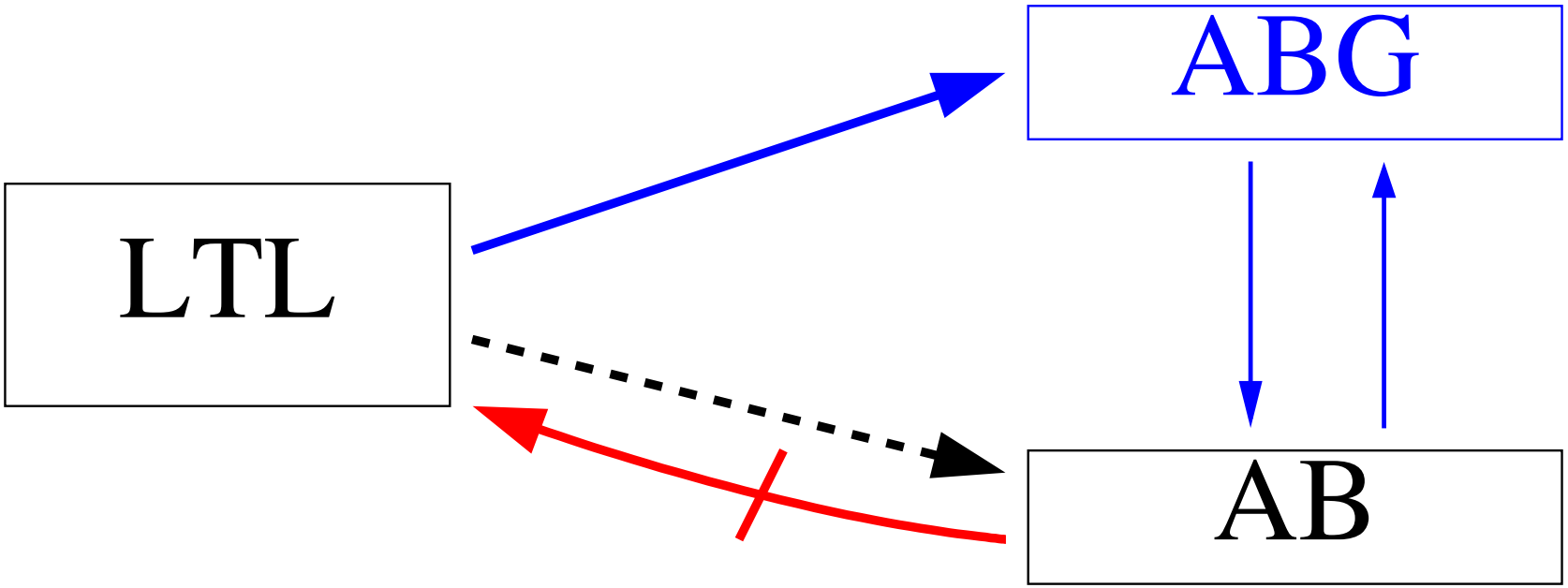
Solution efficace:

Identifier les composantes fortement connexes de $\mathcal{B}$ accessibles depuis s_0.

Tester s'il en existe une qui est *non triviale* (contient au moins une transition) et qui contient un état acceptant.

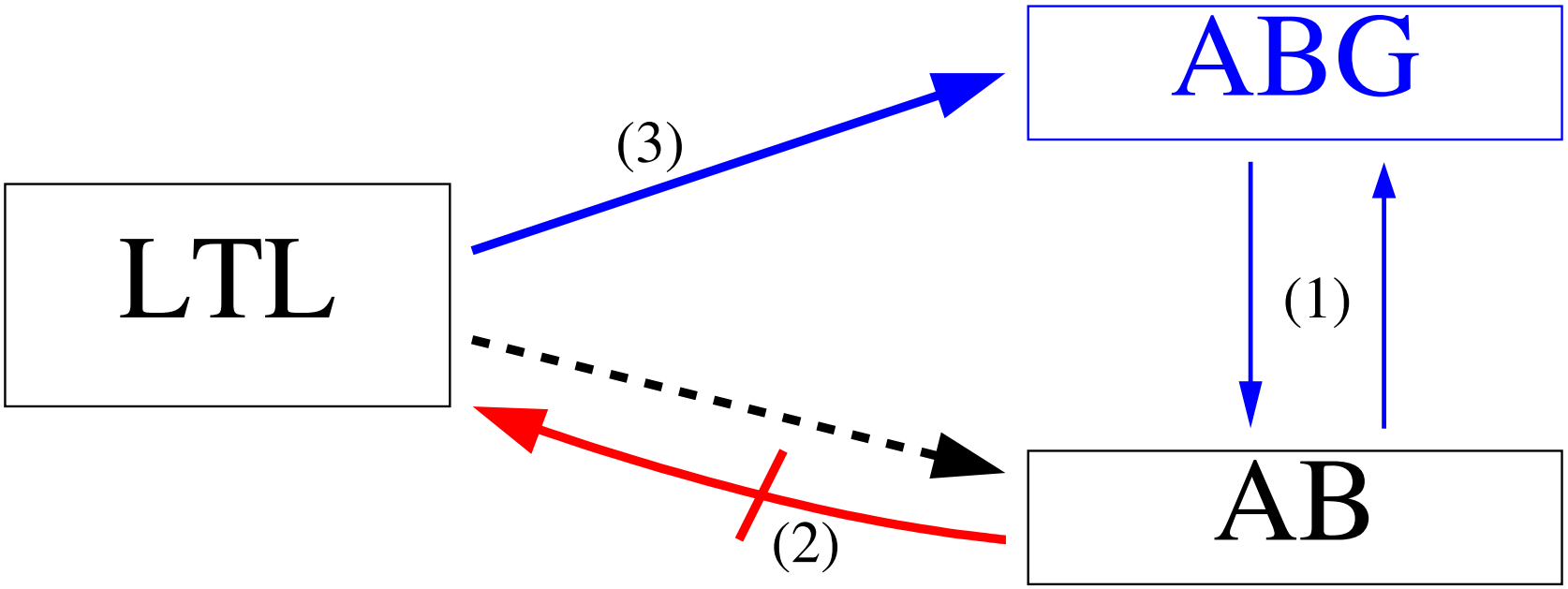
Algorithme de Tarjan: $\mathcal{O}(|\mathcal{B}|)$ temps et espace

Aperçu



La traduction inverse (AB $\rightarrow$ LTL) n'est pas possible en général.

Aperçu



On procède dans l'ordre indiqué.

(1) AB généralisés

Un automate de Büchi généralisé (ABG) est un tuple $\mathcal{G} = (\Sigma, S, s_0, \Delta, \mathcal{F})$.

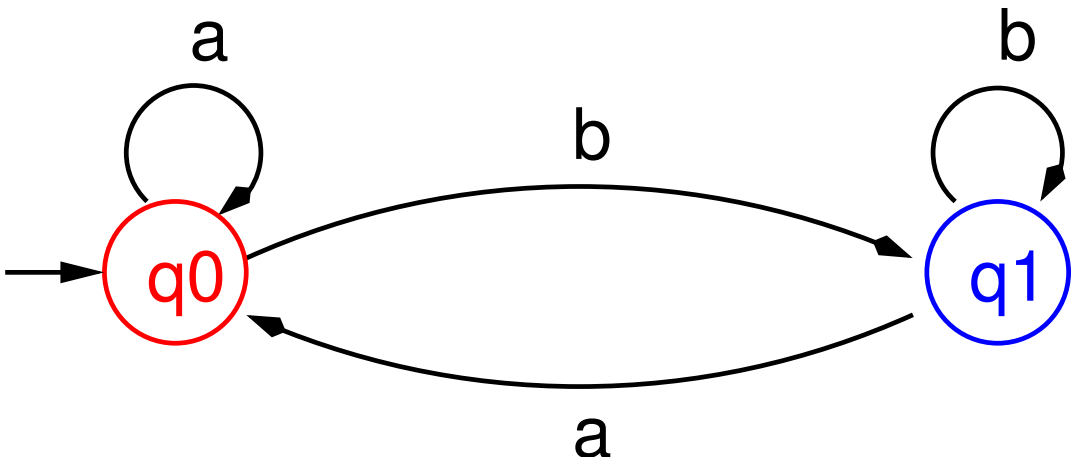
Il y a une seule différence par rapport aux AB normaux:

La condition d'acceptance $\mathcal{F} \subseteq 2^S$ est un *sous-ensemble des parties de S*.

P.ex., soit $\mathcal{F} = \{F_1, \ldots, F_n\}$. Un calcul ρ de $\mathcal{G}$ est acceptant si pour tout F_i ($i = 1, \ldots, n$), ρ contient un nombre infini d'occurrences d'états dans F_i.

ABG: Exemple

Pour le ABG ci-dessous, soit $\mathcal{F} = \{ \{q_0\}, \{q_1\} \}$.



Langage de l'automate: "infiniment souvent *a et* infiniment souvent *b*"

Note: Les ensembles de $\mathcal{F}$ ne sont pas forcement disjoints.

Avantage: Les ABG sont plus succints que les AB.

Traduction AB $\leftrightarrow$ ABG

Les ABG acceptent les mêmes langages que les AB.

AB $\rightarrow$ ABG: (trivial)

Soit $\mathcal{B} = (\Sigma, S, s_0, \Delta, F)$ un AB.

Alors $\mathcal{G} = (\Sigma, S, s_0, \Delta, \{F\})$ est un ABG tel que $\mathcal{L}(\mathcal{G}) = \mathcal{L}(\mathcal{B})$.

ABG $\rightarrow$ AB:

Soit $\mathcal{G} = (\Sigma, S, s_0, \Delta, \{F_1, \ldots, F_n\})$ un ABG.

On construit $\mathcal{B} = (\Sigma, S', s'_0, \Delta', F)$ comme suit:

$S' = S \times \{1, \ldots, n\}$

$s'_0 = (s_0, 1)$

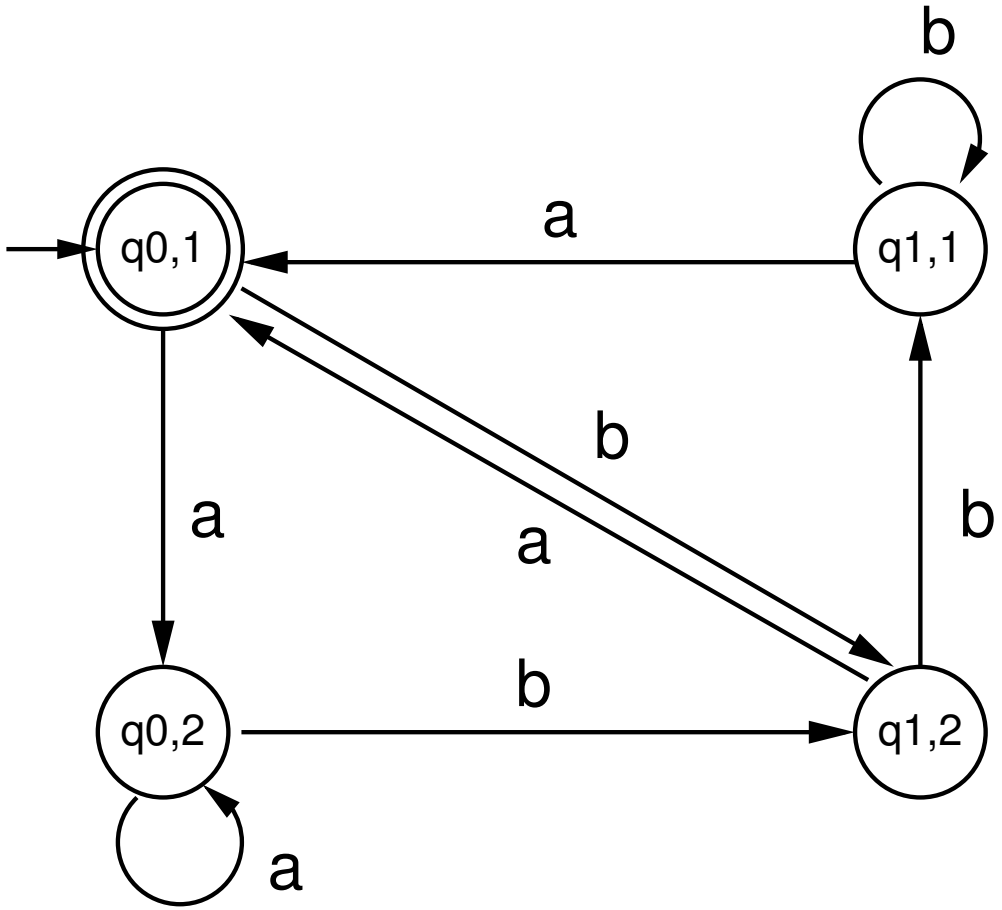
$F = F_1 \times \{1\}$

$((s, i), a, (s', k)) \in \Delta'$ ssi $1 \leq i \leq n,\ (s, a, s') \in \Delta$
et $k = \begin{cases} i & \text{si } s \notin F_i \\ (i \bmod n) + 1 & \text{si } s \in F_i \end{cases}$

Alors $\mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{G})$. (Idée: intersection spécialisée de n automates)

ABG $\rightarrow$ AB: exemple

L'AB correspondant à l'ABG précédent ("infiniment souvent *a et* infiniment souvent *b*") est comme suit:



Remarque: États initiaux multiples

Les définitions données pour les AB(G) contiennent exactement un état initial.

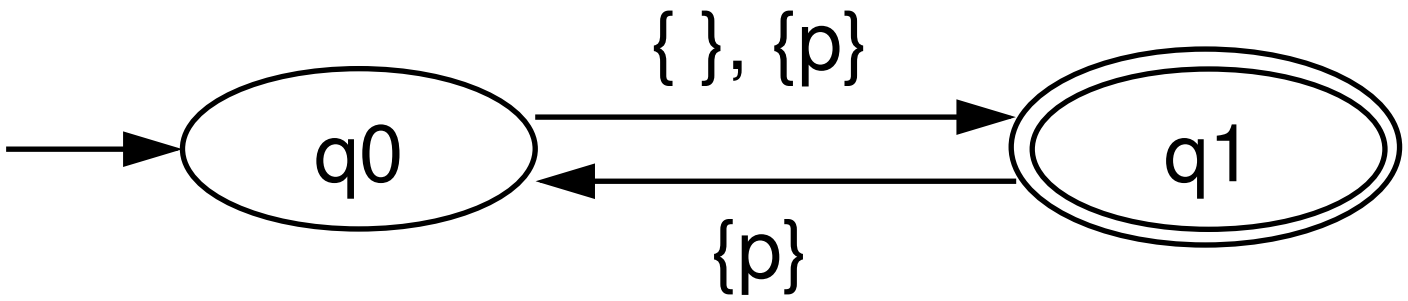
Pour la traduction LTL $\rightarrow$ ABG il conviendra d'avoir de multiples états initiaux (où un mot est accepté s'il est accepté à partir de n'importe quel état initial).

Évidemment, cette extension ne change pas le pouvoir d'expression des AB(G).

(2) Traduction AB $\rightarrow$ LTL

Cette tradition n'est pas possible en général.

Il existe des AB $\mathcal{B}$ tels qu'aucune formule de LTL ϕ ne satisfait $\mathcal{L}(\mathcal{B}) = [\![\phi]\!]$ (Wolper, 1983).



La propriété "*p* tient dans tous les deux coups" n'est pas expressible en LTL.

Preuve (AB $\not\to$ LTL)

On va d'abord montrer un lemme différent:

> Soit ϕ une formule LTL quelconque sur AP et n le nombre d'opérateurs $\mathbf{X}$ dans ϕ. On considère les séquences
>
> $$\sigma_i = \{p\}^i \emptyset \{p\}^\omega$$
>
> pour $i \geq 0$. Pour toute paire $i, k > n$ on a : $\sigma_i \models \phi$ iff $\sigma_k \models \phi$.

Preuve par récurrence sur ϕ:

> Si $\phi = p$, où $p \in AP$, alors $n = 0$ et $i, k \geq 1$.
> Du coup, $\sigma_i \models p$ et $\sigma_k \models p$.
>
> Pour les autres cas, on suppose que la propriété est vraie pour ϕ_1 und ϕ_2, c.à.d. si ϕ_1, ϕ_2 contiennent n_1 et n_2 occurrences de $\mathbf{X}$, respectivement, alors pour tout $i_1, k_1 > n_1$ on a $\sigma_{i_1} \models \phi_j$ ssi $\sigma_{k_1} \models \phi_j$, pour $j = 1, 2$.

Si $\phi = \neg\phi_1$, alors la preuve est immédiate.

Pour $\phi = \phi_1 \vee \phi_2$: pareil

Si $\phi = \mathbf{X}\,\phi_1$, alors $n_1 = n - 1$. Comme $i - 1, k - 1 > n - 1 = n_1$, l'hypothèse de récurrence implique: $\sigma_i^1 = \sigma_{i-1} \models \phi_1$ ssi $\sigma_k^1 = \sigma_{k-1} \models \phi_1$, ce qui conclut la preuve.

Pour $\phi = \phi_1\ \mathbf{U}\ \phi_2$: Soit $m > n$. On a:

$$\phi_1\ \mathbf{U}\ \phi_2 \equiv \phi_2 \vee (\phi_1 \wedge \mathbf{X}(\phi_1\ \mathbf{U}\ \phi_2))$$

En itérant cette transformation on obtient:

$$\sigma_m \models \phi \quad \text{ssi} \quad \sigma_m \models \phi_2 \vee (\sigma_m \models \phi_1 \wedge (\sigma_{m-1} \models \phi_2 \vee (\ldots$$
$$(\sigma_{n+1} \models \phi_1 \wedge \sigma_n \models \phi_1\ \mathbf{U}\ \phi_2))))$$

Selon l'hypothèse de récurrence, on peut remplacer les indices plus grands que n par $n+1$ sans rien changer.

$$\sigma_m \models \phi \quad \text{ssi} \quad \sigma_{n+1} \models \phi_2 \vee (\sigma_{n+1} \models \phi_1 \wedge (\sigma_{n+1} \models \phi_2 \vee (\ldots \\ (\sigma_{n+1} \models \phi_1 \wedge \sigma_n \models \phi_1 \,\mathbf{U}\, \phi_2))))$$

Ceci se simplifie comme suit:

$$\sigma_m \models \phi \quad \text{ssi} \quad \sigma_{n+1} \models \phi_2 \vee (\sigma_{n+1} \models \phi_1 \wedge \sigma_n \models \phi_1 \,\mathbf{U}\, \phi_2)$$

Du coup, la validité de $\sigma_m \models \phi$ est entièrement indépendante de m, ce qui implique la propriété souhaitée pour i et k.

Supposons maintenant qu'il existe une formule LTL ϕ qui exprime le langage de l'AB précédent ("*p* tient dans tous les deux coups"). Soit n le nombre de $\mathbf{X}$ dans ϕ.

On considère les séquences σ_{n+1} et σ_{n+2}. Si n est pair alors $\sigma_{n+1} \not\models \phi$ et $\sigma_{n+2} \models \phi$. Si n est impair, c'est l'inverse.

Pourtant, le lemme précédent nous dit que ceci est impossible: soit σ_{n+1} et σ_{n+2} satisfont ϕ tous les deux, soit ni l'une ni l'autre. Du coup, une telle formule ϕ n'existe pas.

Partie 5: LTL et AB

Résumé

On va resoudre le problème suivant:

> Étant donné une formule ϕ sur AP de LTL, construire un ABG $\mathcal{G}$ (avec plusieurs états initiaux) tel que $\mathcal{L}(\mathcal{G}) = [\![\phi]\!]$.
>
> $\mathcal{G}$ peut alors être converti en un AB normal.

Remarques:

> Construction “monolithique” (on traite la formule toute entière), non inductive
>
> En fait, notre ABG doit vérifier de multiples sous-formules *en même temps*: (p.ex.: $(p\,\mathbf{U}\,q)\,\mathbf{U}\,r$ ou $(\mathbf{G}\,\mathbf{F}\,p) \rightarrow (\mathbf{G}(q \rightarrow \mathbf{F}\,r))$).

D’autres remarques:

La construction qu’on utilisera sera relativement simple à présenter, mais produit des automates non optimaux (toujours de taille exponentielle).

Plutôt une “preuve de concept”, mais pas un algorithme qu’on utilisera en réalité.

Il existe de traductions plus efficaces mais plus compliquées algorithmiquement (pointeurs en fin de ce chapitre).

Structure de la construction

1. On convertit ϕ dans une forme normale.

2. Chaque état sera “responsable” d’un ensemble de sous-formules.

3. Les transitions assurent le traitement des sous-formules “simples” (p ou $\mathbf{X}\, p$).

4. La condition d’acceptance prend en charge les sous-formules avec $\mathbf{U}$.

1. Forme normale négative

Soit AP un ensemble de prédicats. Les formules FNN over AP sont les suivantes:

Si $p \in AP$ alors p et $\neg p$ sont FNN.
(Les négations seront *uniquement* devant les prédicats.)

Si ϕ_1 et ϕ_2 sont FNN, alors les suivants le sont aussi:

$$\phi_1 \vee \phi_2, \quad \phi_1 \wedge \phi_2, \quad \mathbf{X}\,\phi_1, \quad \phi_1 \,\mathbf{U}\, \phi_2, \quad \phi_1 \,\mathbf{R}\, \phi_2.$$

Proposition: Pour toute formule LTL ϕ il existe une formule équivalente FNN:

$$\begin{array}{rcl@{\qquad}rcl} \neg(\phi_1 \,\mathbf{R}\, \phi_2) & \equiv & \neg\phi_1 \,\mathbf{U}\, \neg\phi_2 & \neg(\phi_1 \,\mathbf{U}\, \phi_2) & \equiv & \neg\phi_1 \,\mathbf{R}\, \neg\phi_2 \\ \neg(\phi_1 \wedge \phi_2) & \equiv & \neg\phi_1 \vee \neg\phi_2 & \neg(\phi_1 \vee \phi_2) & \equiv & \neg\phi_1 \wedge \neg\phi_2 \\ \neg\,\mathbf{X}\,\phi & \equiv & \mathbf{X}\,\neg\phi & \neg\neg\phi & \equiv & \phi \end{array}$$

FNN: Exemple

Traduction en FNN :

$$
\begin{aligned}
\mathbf{G}(p \rightarrow \mathbf{F}\, q) &\equiv \neg\, \mathbf{F}\, \neg(p \rightarrow \mathbf{F}\, q) \\
&\equiv \neg(\mathbf{true}\, \mathbf{U}\, \neg(p \rightarrow \mathbf{F}\, q)) \\
&\equiv \neg \mathbf{true}\, \mathbf{R}\, (p \rightarrow \mathbf{F}\, q) \\
&\equiv \mathbf{false}\, \mathbf{R}\, (\neg p \vee \mathbf{F}\, q) \\
&\equiv \mathbf{false}\, \mathbf{R}\, (\neg p \vee (\mathbf{true}\, \mathbf{U}\, q))
\end{aligned}
$$

Remarque: Dans la suite, on suppose que ϕ est en FNN.

2. Sous-formules

Soit ϕ une formule FNN. $Sub(\phi)$ est l'ensemble minimal tel que:

- $\phi \in Sub(\phi)$;
- $\mathbf{true} \in Sub(\phi)$;
- $\phi_1 \in Sub(\phi)$ ssi $\neg\phi_1 \in Sub(\phi)$;
- si $\mathbf{X}\,\phi_1 \in Sub(\phi)$ alors $\phi_1 \in Sub(\phi)$;
- si $\phi_1 \vee \phi_2 \in Sub(\phi)$ alors $\phi_1, \phi_2 \in Sub(\phi)$;
- si $\phi_1 \wedge \phi_2 \in Sub(\phi)$ alors $\phi_1, \phi_2 \in Sub(\phi)$;
- si $\phi_1\ \mathbf{U}\ \phi_2 \in Sub(\phi)$ alors $\phi_1, \phi_2 \in Sub(\phi)$;
- si $\phi_1\ \mathbf{R}\ \phi_2 \in Sub(\phi)$ alors $\phi_1, \phi_2 \in Sub(\phi)$.

Note: On a $|Sub(\phi)| = \mathcal{O}(|\phi|)$ (une sous-formule par élément syntaxique).

Ensembles cohérents

Rappelons l'item 2 de la construction:

> Tout état sera étiqueté par un sous-ensemble de $Sub(\phi)$.

Idée: Depuis un état M on accepte les séquences validant l'intersection des sous-formules de M et ne validant aucune sous-formule de $Sub(\phi) \setminus M$.

États cohérents

Petite optimisation : On exclut certains ensembles M dont le langage serait évidemment vide.

Définition: On appelle $M \subseteq Sub(\phi)$ *cohérent* ssi:

- $\mathbf{true} \in M$
- si $\phi_1 \in Sub(\phi)$ alors $\neg\phi_1 \in M$ ssi $\phi_1 \notin M$;
- si $\phi_1 \wedge \phi_2 \in Sub(\phi)$ alors $\phi_1 \wedge \phi_2 \in M$ ssi $\phi_1 \in M$ et $\phi_2 \in M$;
- si $\phi_1 \vee \phi_2 \in Sub(\phi)$ alors $\phi_1 \vee \phi_2 \in M$ ssi $\phi_1 \in M$ ou $\phi_2 \in M$.

$CS(\phi)$ denote les sous-ensembles cohérents de $Sub(\phi)$.

Traduction (1)

Soit ϕ FNN et $\mathcal{G} = (\Sigma, S, S_0, \Delta, \mathcal{F})$ un ABG tel que:

$\Sigma = 2^{AP}$

(les affectations de AP)

$S = CS(\phi)$

(tout état est un ensemble cohérent)

$S_0 = \{ M \in S \mid \phi \in M \}$

(les états initiaux admettent les séquences satisfaisant ϕ)

Δ et $\mathcal{F}$: voir les transparents suivants

Traduction (2)

Transitions: $(M, \sigma, M') \in \Delta$ ssi $\sigma = M \cap AP$ et:

- si $\mathbf{X}\,\phi_1 \in Sub(\phi)$ alors $\mathbf{X}\,\phi_1 \in M$ ssi $\phi_1 \in M'$;
- si $\phi_1\,\mathbf{U}\,\phi_2 \in Sub(\phi)$ alors $\phi_1\,\mathbf{U}\,\phi_2 \in M$
  ssi $\phi_2 \in M$ ou ($\phi_1 \in M$ et $\phi_1\,\mathbf{U}\,\phi_2 \in M'$);
- si $\phi_1\,\mathbf{R}\,\phi_2 \in Sub(\phi)$ alors $\phi_1\,\mathbf{R}\,\phi_2 \in M$
  ssi $\phi_1 \wedge \phi_2 \in M$ ou ($\phi_2 \in M$ et $\phi_1\,\mathbf{R}\,\phi_2 \in M'$).

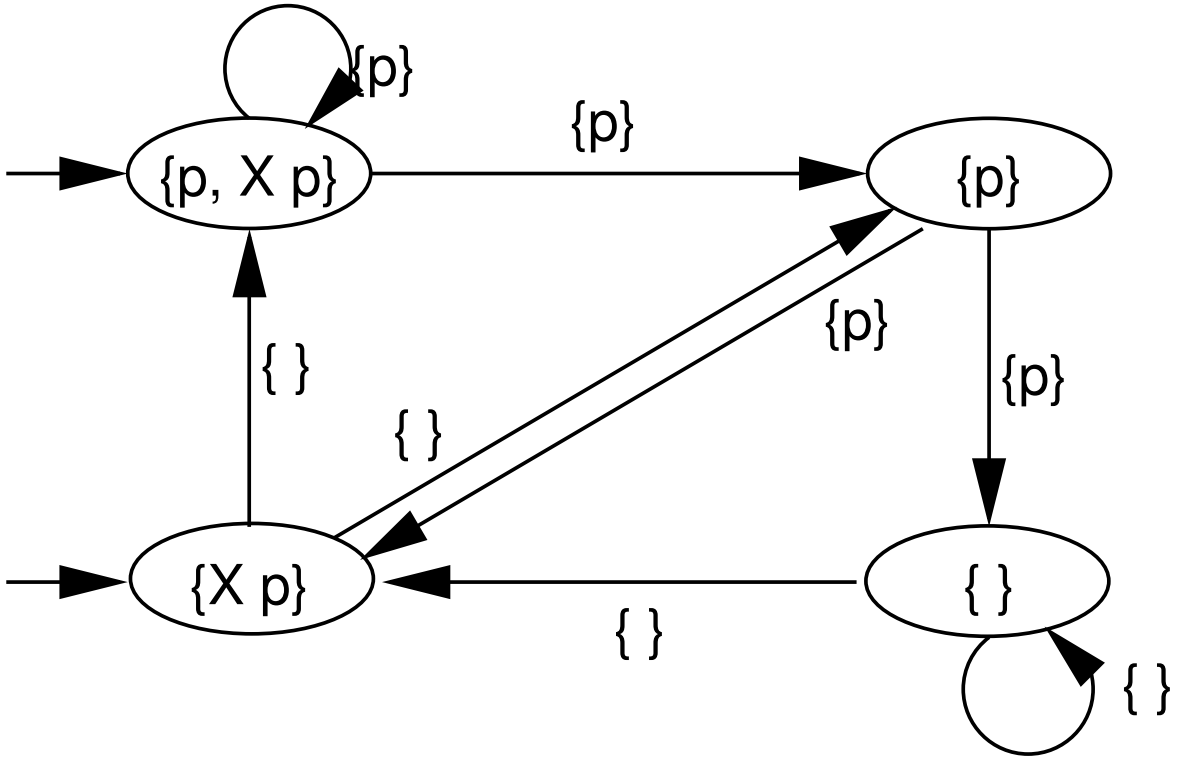
Condition d'acceptance:

$\mathcal{F}$ contient un ensemble F_ψ pour toute sous-formule ψ de la forme $\phi_1\,\mathbf{U}\,\phi_2$:

$$F_\psi = \{\, M \in CS(\phi) \mid \phi_2 \in M \ \text{ ou } \ \neg(\phi_1\,\mathbf{U}\,\phi_2) \in M \,\}.$$

Traduction: Exemple 1

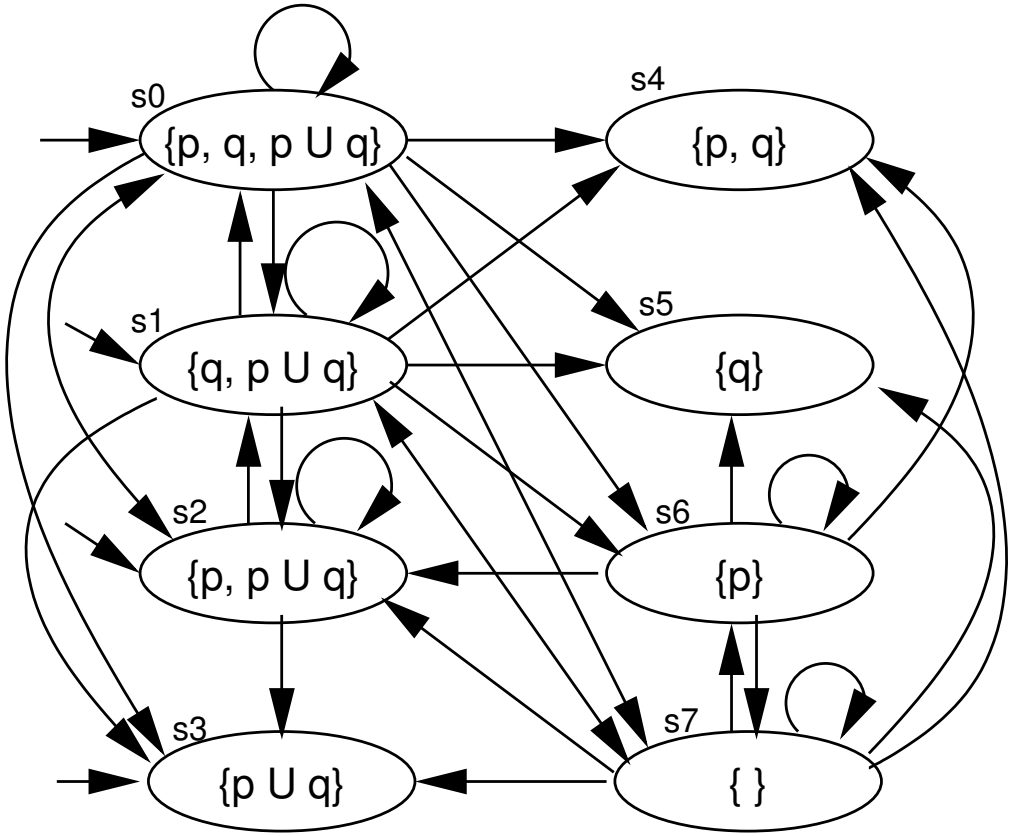
$\phi = \mathbf{X}\, p$



Cet ABG a deux états initiaux et $\mathcal{F} = \emptyset$, alors tout calcul infini est acceptant. (Les formules négatives ont été omises.)

Traduction: Exemple 2

$\phi \equiv p \, \mathbf{U} \, q$



ABG avec $\mathcal{F} = \{\{s_0, s_1, s_4, s_5, s_6, s_7\}\}$, les étiquettes sur les arêtes ont été omises.

Preuve de correction

On souhaite prouver le suivant:

$$\sigma \in \mathcal{L}(\mathcal{G}) \quad \text{ssi} \quad \sigma \in [\![\phi]\!]$$

Pour cela, on va prouver la propriété plus forte suivante:

> Soit α une suite d'ensembles cohérents, c'est à dire des états de $\mathcal{G}$, et soit σ une suite d'affectations de AP.
>
> α est un calcul acceptant de $\mathcal{G}$ sur σ
> ssi $\sigma^i \in [\![\psi]\!]$ pour tout $i \geq 0$ et $\psi \in \alpha(i)$.
>
> La preuve est alors obtenue avec le choix des états initiaux.

Correction (2)

Par construction, on a $\sigma(i) = \alpha(i) \cap AP$ pour tout $i \geq 0$.

Preuve par induction structurelle sur ψ:

pour $\psi = p$ et $\psi = \neg p$ si $p \in AP$:
évident car $\sigma^i \in [\![p]\!]$ ssi $p \in \sigma(i)$ ssi $p \in \alpha(i)$.

pour $\psi_1 \vee \psi_2$ et $\psi_1 \wedge \psi_2$: on rappelle que $\alpha(i)$ est cohérent et on applique la récurrence sur ψ_1 et ψ_2, respectivement.

pour $\mathbf{X}\,\psi_1$: conséquence de la construction de Δ et par récurrence sur ψ_1.

Correction (3)

pour $\psi = \psi_1 \,\mathbf{R}\, \psi_2$:

par construction de Δ, le dépliage de $\mathbf{R}$, et par récurrence sur ϕ_1 et ϕ_2.

pour $\psi = \psi_1 \,\mathbf{U}\, \psi_2$:

analogue à $\mathbf{R}$, mais en plus il faut assurer que $\psi_2 \in \alpha(k)$ pour un $k \geq i$. Si ceci n'est pas le cas, alors on a $\psi_1 \,\mathbf{U}\, \psi_2 \in \alpha(k)$ pour tout $k \geq i$. Pourtant, aucun de ces états est dans F_ψ, du coup α n'est pas acceptant, une contradiction.

Complexité de la traduction

La traduction produit un ABG de taille $\mathcal{O}(2^{|\phi|})$.

Question: Exite-il une traduction meilleure ?

Réponse 1: Non (pas en général). Il existent des formules qui nécessitent un ABG de taille exponentielle.

Exemple: La formule suivante $\{p_0, \ldots, p_{n-1}\}$ simule un compteur à n-bits.

$$\mathbf{G}(p_0 \not\leftrightarrow \mathbf{X}\, p_0) \wedge \bigwedge_{i=1}^{n-1} \mathbf{G}\left(\left(p_i \not\leftrightarrow \mathbf{X}\, p_i\right) \leftrightarrow \left(p_{i-1} \wedge \neg\, \mathbf{X}\, p_{i-1}\right)\right)$$

Cette formule est de taille $\mathcal{O}(n)$, mais évidemment tout ABG doit avoir au moins 2^n états.

Réponse 2: Oui (parfois). Il y a des procédures qui produisent des automates moins grandes *dans la plupart de cas*.

Quelques outils:

Spin (`spin -f 'p U q'`)

LTL2BA (web applet)

Littérature:

Gerth, Peled, Vardi, Wolper: *Simple On-the-fly Automatic Verification of Linear Temporal Logic*, 1996

Oddoux, Gastin: *Fast LTL to Büchi Automata Translation*, 2001

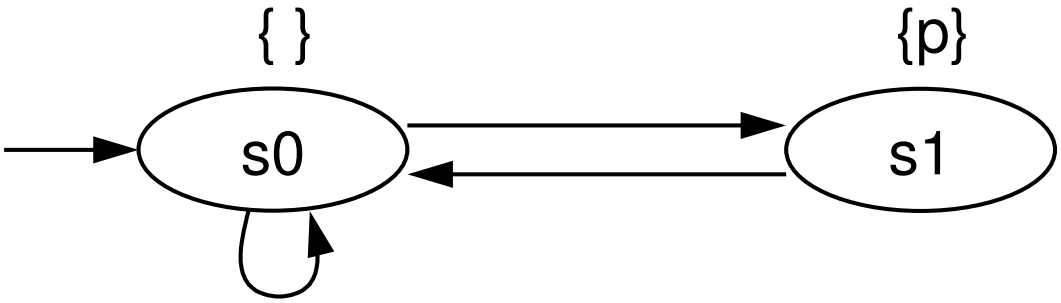
Partie 6: Logique branchante

Motivation

Les logiques linéaires décrivent les propriétés des calculs. Mais il est impossible de parler des évolutions *potentielles*.

Exemple: Dans la structure ci-dessous, à tout moment, *peut* basculer dans un état satisfaisant *p*, mais il n'y est pas obligé.

Or, l'existence de ce choix ne peut être décrite dans LTL.



Dans la suite, on va modéliser le comportement d'un système par un arbre qui exprime tous les futurs possibles.

On étudie alors une logique dite branchante permettant de raisonner sur ces différents comportements. Du coup, on évalue cette logique sur des *arbres* d'affectations.

La logique qu'on étudie s'appelle CTL. Celle-ci est probablement la logique branchante la plus "populaire" car

elle est suffisamment expressive pour pas mal de propriétés;

elle permet des algorithmiques efficaces;

du coup, elle est souvent utilisée dans la vérification automatisée.

Arbre de calcul

Soit $\mathcal{T} = (V, \rightarrow, r, AP, \nu)$ une SK (où V est potentiellement de taille infinie). On appelle $\mathcal{T}$ un arbre d'affectations si $(V, \rightarrow)$ est un arbre dirigé avec r comme racine et $\nu : V \rightarrow 2^{AP}$ associe une affectation à chaque sommet.

$\lceil v \rceil$ denote le sous-arbre dont la racine est $v \in V$.

Soit $\mathcal{K} = (S, \rightarrow, s_0, AP, \nu)$ une SK et $s \in S$.

$\mathcal{T}_{\mathcal{K}}(s)$ denote l'unique arbre d'affectations avec racine r qui possède les propriétés suivantes:

$\nu(r) = \nu(s)$

$s \rightarrow s'$ in $\mathcal{K}$ ssi $\mathcal{T}_{\mathcal{K}}(s)$ possède une transition $r \rightarrow r'$ telle que $\lceil r' \rceil$ est isomorphe à $\mathcal{T}_{\mathcal{K}}(s')$.

On appelle $\mathcal{T}_{\mathcal{K}}(s)$ l'arbre de calcul de s.

CTL: Syntaxe

Définissons d'abord une syntaxe minimaliste.

Soit AP un ensemble de prédicats. Les formules de CTL sur AP sont les suivantes:

si $a \in AP$, alors a est une formule.

si ϕ_1, ϕ_2 sont des formules, alors les suivantes le sont aussi:

$$\neg\phi_1, \qquad \phi_1 \vee \phi_2, \qquad \mathbf{EX}\,\phi_1, \qquad \mathbf{EG}\,\phi_1, \qquad \phi_1\,\mathbf{EU}\,\phi_2$$

CTL: Sémantique

Soit $\mathcal{K}$ une SK, s un état et ϕ une formule de CTL. On définit un ensemble $[\![\phi]\!]_{\mathcal{K}}$ tel que $s \in [\![\phi]\!]_{\mathcal{K}}$ ssi $\mathcal{T}_{\mathcal{K}}(s) \models \phi$.

$$
\begin{aligned}
[\![a]\!]_{\mathcal{K}} &= \{\, s \mid a \in \nu(s) \,\} \qquad \text{pour } a \in AP \\
[\![\neg\phi_1]\!]_{\mathcal{K}} &= S \setminus [\![\phi_1]\!]_{\mathcal{K}} \\
[\![\phi_1 \vee \phi_2]\!]_{\mathcal{K}} &= [\![\phi_1]\!]_{\mathcal{K}} \cup [\![\phi_2]\!]_{\mathcal{K}} \\
[\![\mathbf{EX}\,\phi_1]\!]_{\mathcal{K}} &= \{\, s \mid \text{il existe } t \text{ t.q. } s \to t \text{ et } t \in [\![\phi_1]\!]_{\mathcal{K}} \,\} \\
[\![\mathbf{EG}\,\phi_1]\!]_{\mathcal{K}} &= \{\, s \mid \text{il existe un calcul } \rho \text{ avec } \rho(0) = s \\
&\qquad \text{et } \rho(i) \in [\![\phi_1]\!]_{\mathcal{K}} \text{ pour tout } i \geq 0 \,\} \\
[\![\phi_1\ \mathbf{EU}\ \phi_2]\!]_{\mathcal{K}} &= \{\, s \mid \text{il existe un calcul } \rho \text{ avec } \rho(0) = s \text{ et } k \geq 0 \text{ t.q.} \\
&\qquad \rho(i) \in [\![\phi_1]\!]_{\mathcal{K}} \text{ pour tout } i < k \text{ et } \rho(k) \in [\![\phi_2]\!]_{\mathcal{K}} \,\}
\end{aligned}
$$

On dit que $\mathcal{K}$ satisfait ϕ (noté $\mathcal{K} \models \phi$) ssi $s_0 \in [\![\phi]\!]_{\mathcal{K}}$.

Le problème de model-checking problem local est de vérifier si $\mathcal{K} \models \phi$.

Le problème de model-checking problem global est de calculer $[\![\phi]\!]_{\mathcal{K}}$.

On déclare deux formules équivalentes ($\phi_1 \equiv \phi_2$) ssi pour toute SK $\mathcal{K}$ on a $[\![\phi_1]\!]_{\mathcal{K}} = [\![\phi_2]\!]_{\mathcal{K}}$.

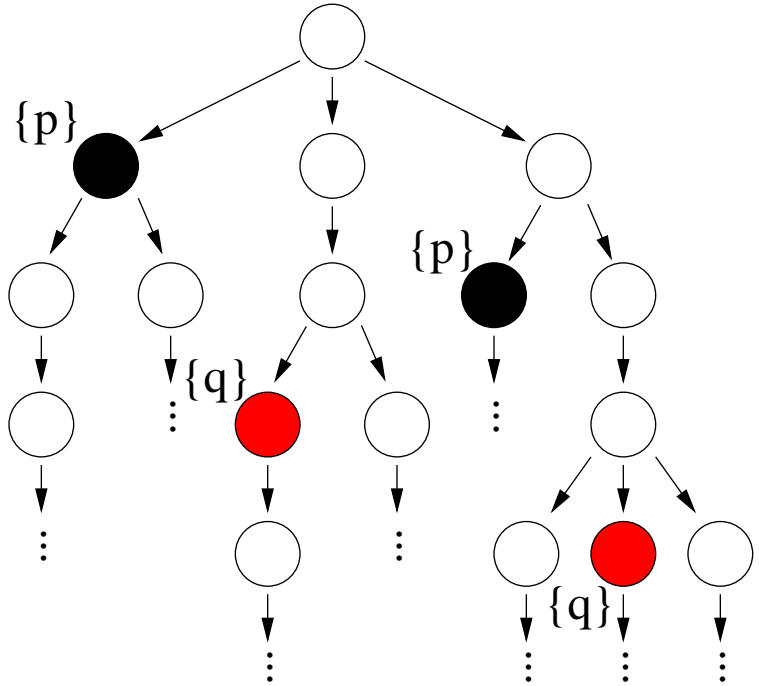
Dans le suivant, on omet l'indice $\mathcal{K}$ de $[\![\cdot]\!]_{\mathcal{K}}$ si $\mathcal{K}$ est évident.

CTL: Syntaxe étendue

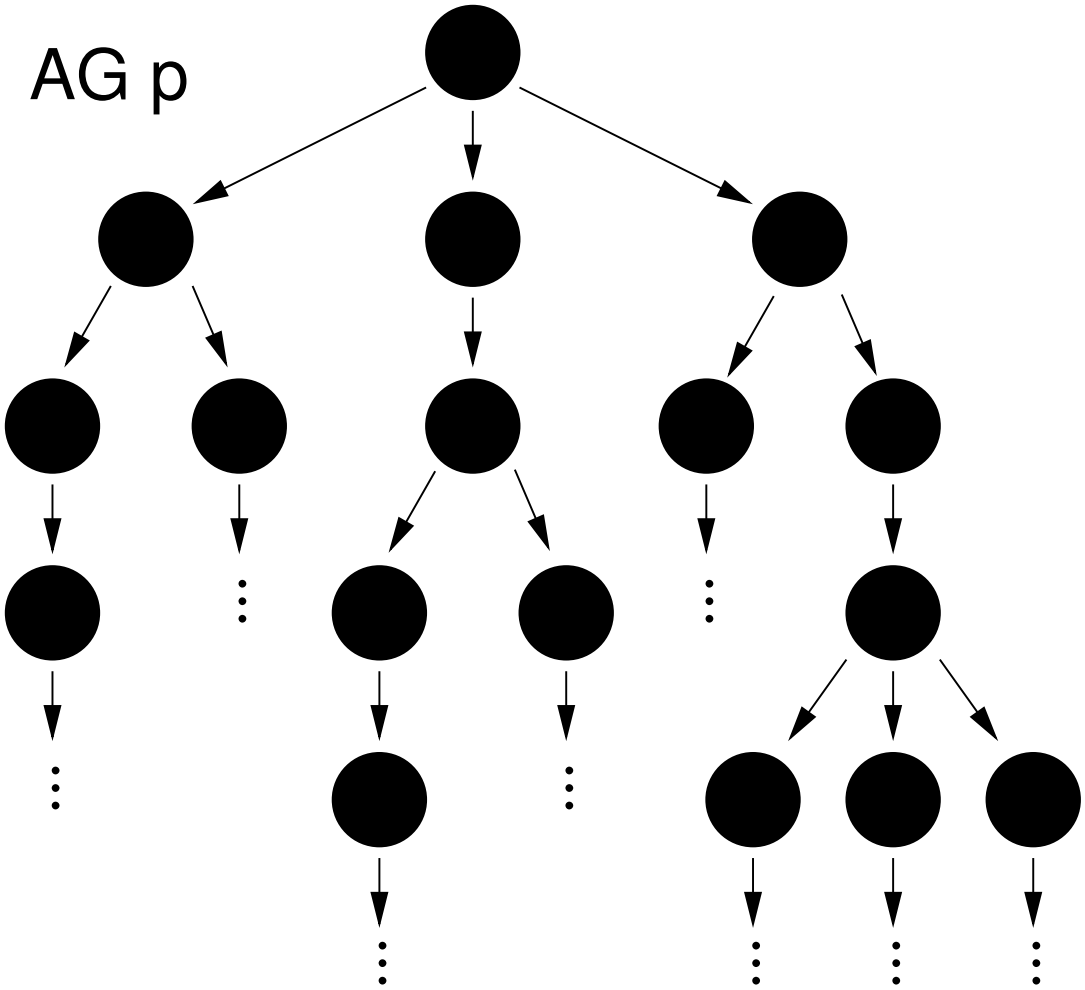
$$
\begin{aligned}
\phi_1 \wedge \phi_2 &\equiv \neg(\neg\phi_1 \vee \neg\phi_2) \\
\mathbf{true} &\equiv a \vee \neg a \\
\mathbf{false} &\equiv \neg\mathbf{true} \\
\phi_1\,\mathbf{EW}\,\phi_2 &\equiv \mathbf{EG}\,\phi_1 \vee (\phi_1\,\mathbf{EU}\,\phi_2) \\
\mathbf{EF}\,\phi &\equiv \mathbf{true}\,\mathbf{EU}\,\phi
\end{aligned}
\qquad
\begin{aligned}
\mathbf{AX}\,\phi &\equiv \neg\,\mathbf{EX}\,\neg\phi \\
\mathbf{AG}\,\phi &\equiv \neg\,\mathbf{EF}\,\neg\phi \\
\mathbf{AF}\,\phi &\equiv \neg\,\mathbf{EG}\,\neg\phi \\
\phi_1\,\mathbf{AW}\,\phi_2 &\equiv \neg(\neg\phi_2\,\mathbf{EU}\,\neg(\phi_1 \vee \phi_2)) \\
\phi_1\,\mathbf{AU}\,\phi_2 &\equiv \mathbf{AF}\,\phi_2 \wedge (\phi_1\,\mathbf{AW}\,\phi_2))
\end{aligned}
$$

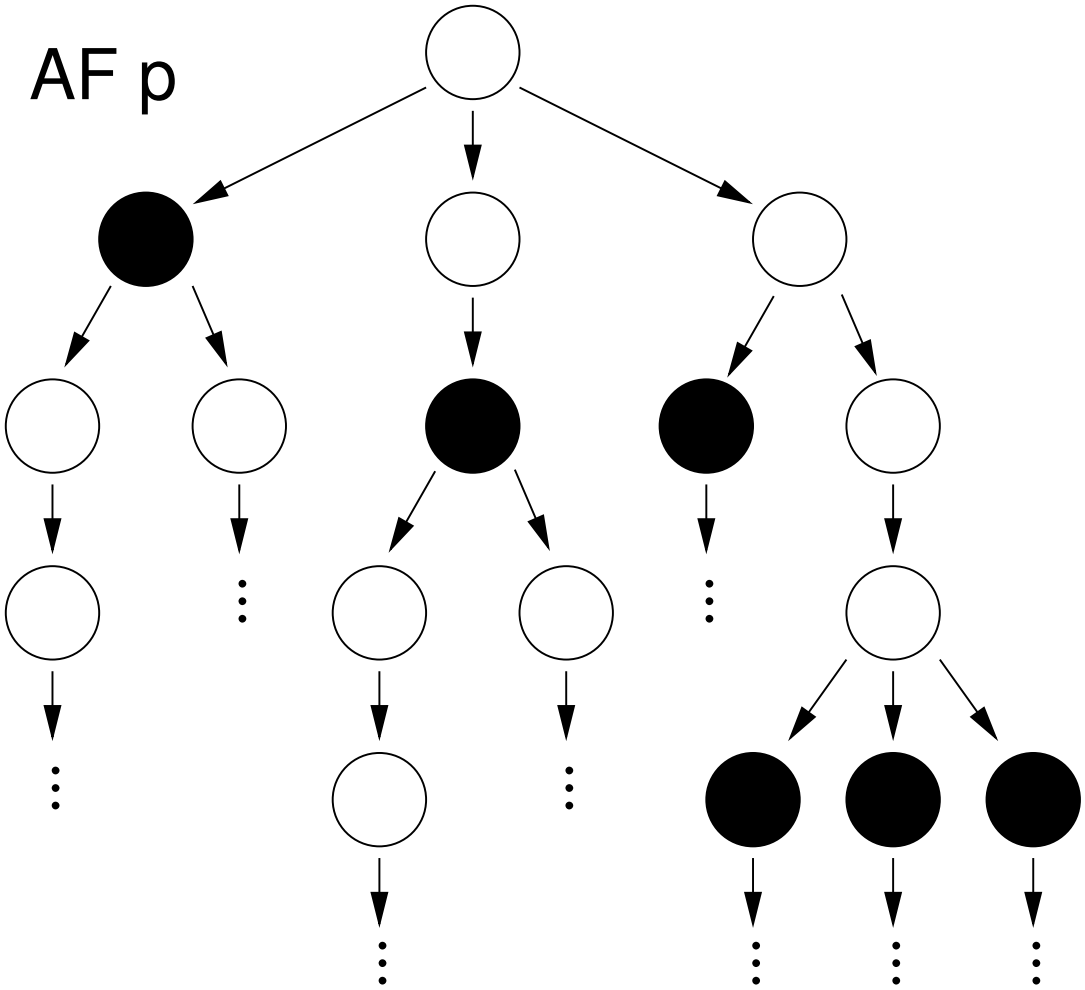
CTL: Exemples

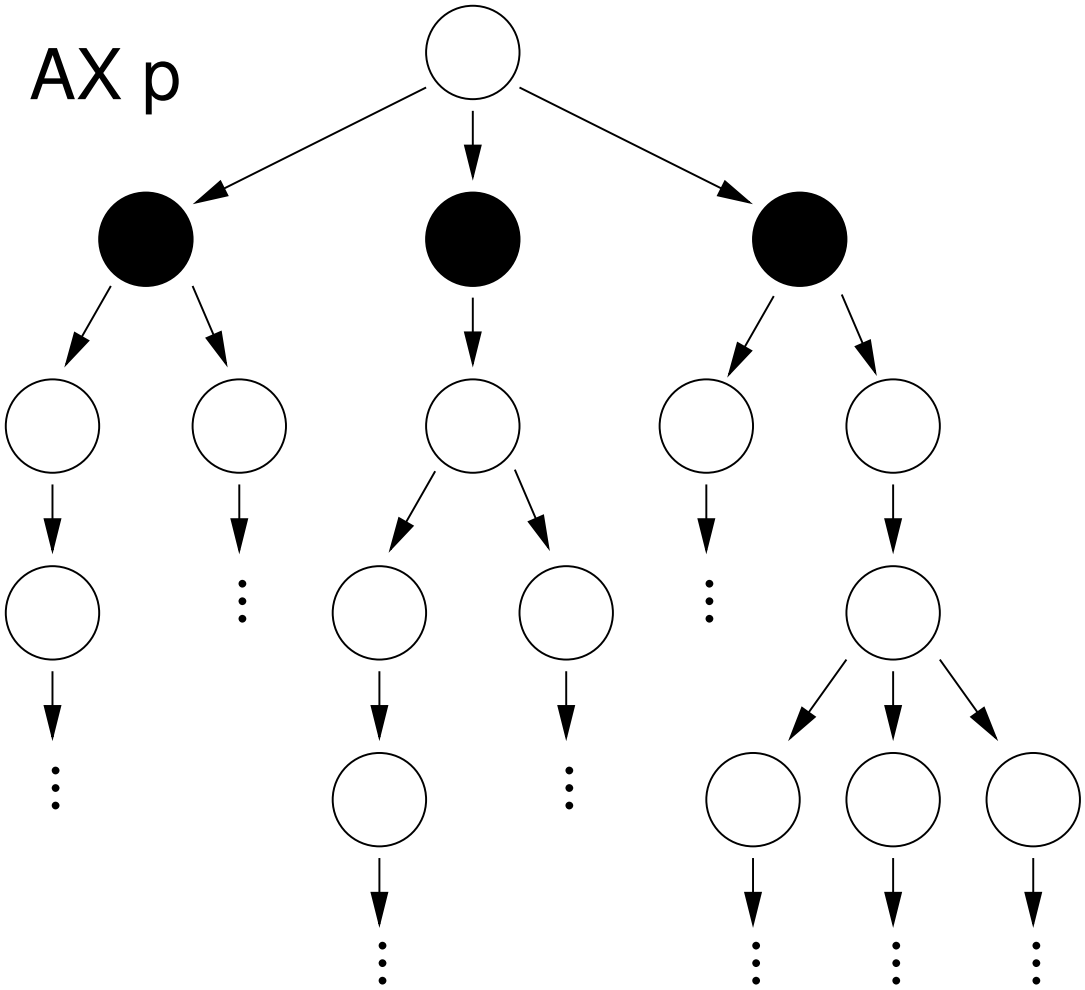
On utilisera l'arbre de calcul suivant comme exemple (avec des distributions variables d'états rouges et noirs):

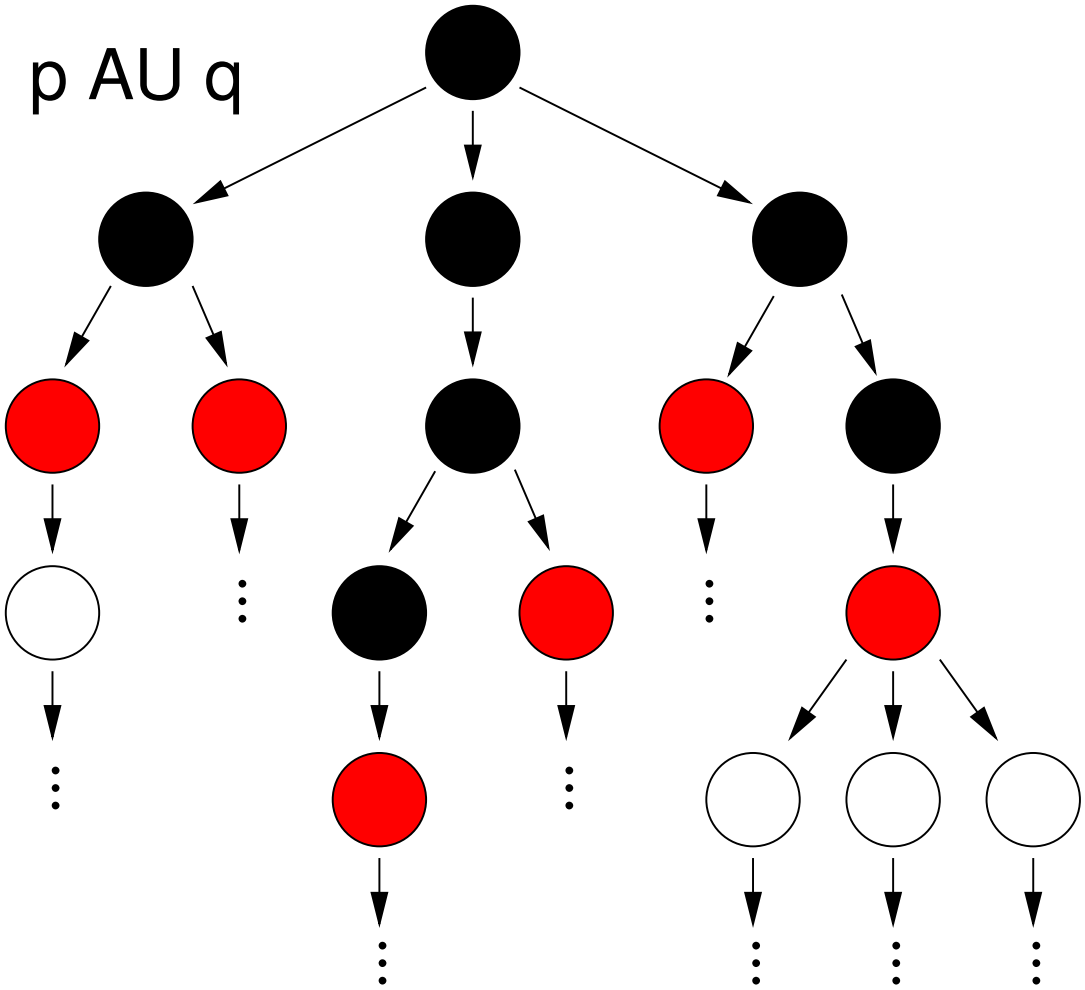


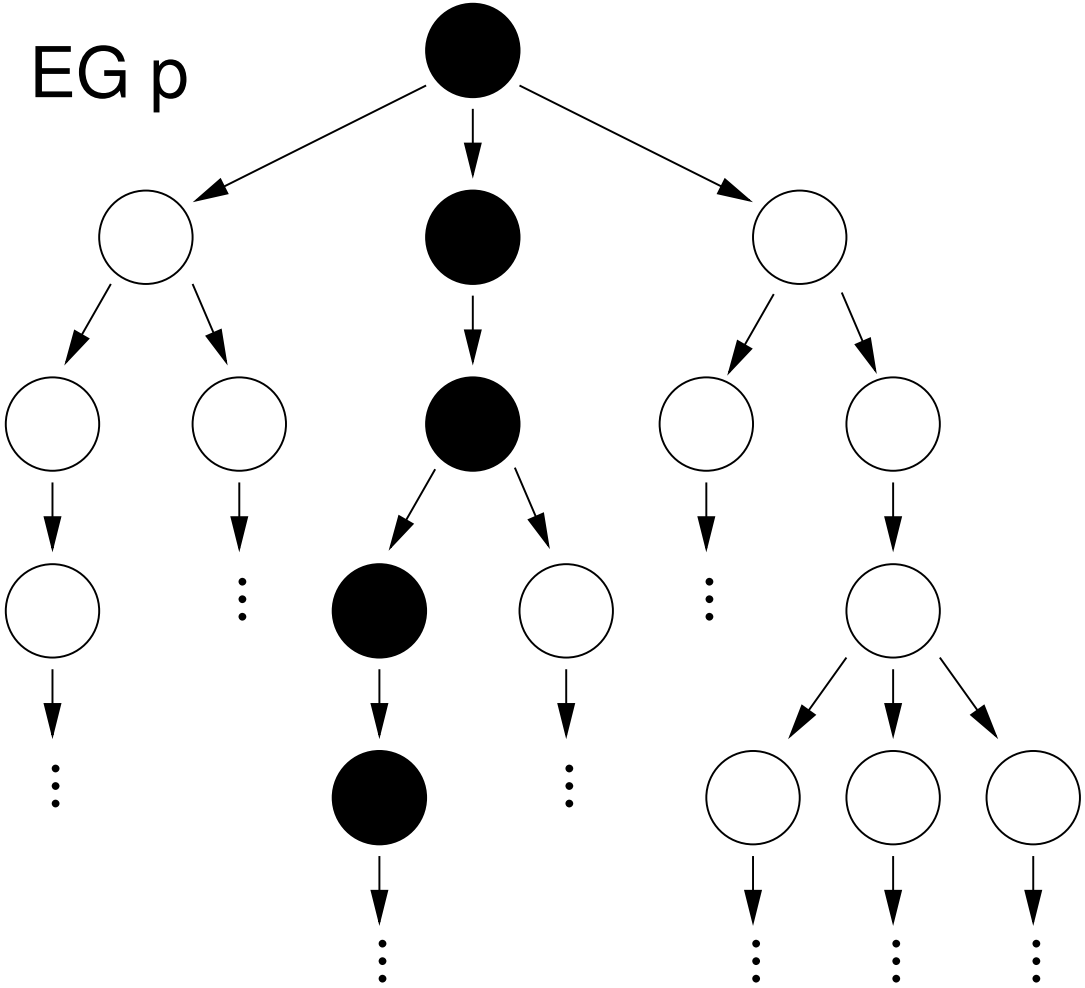
Dans les transparents suivants, la racine de l'arbre satisfait la formule donnée si les états noirs satisfont p et les rouges satisfont q.

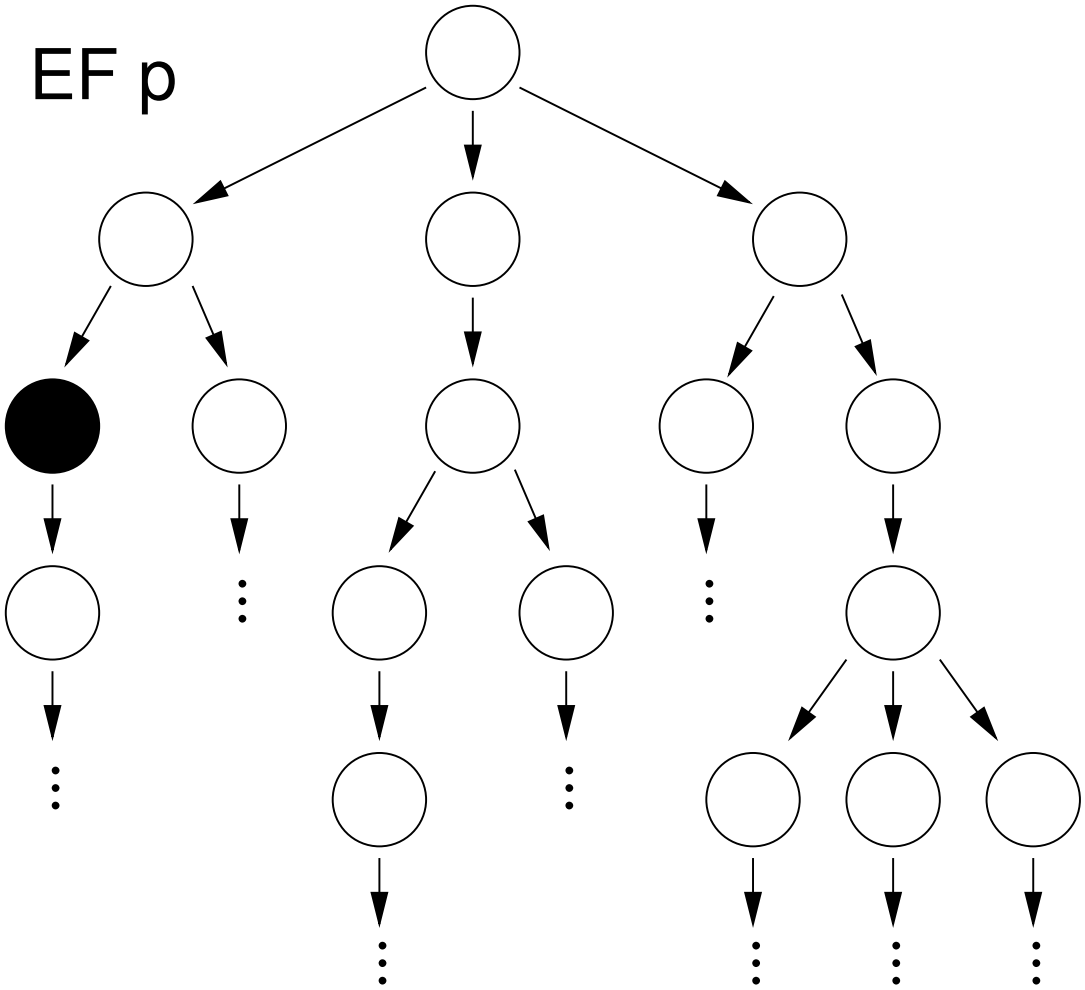

AG p

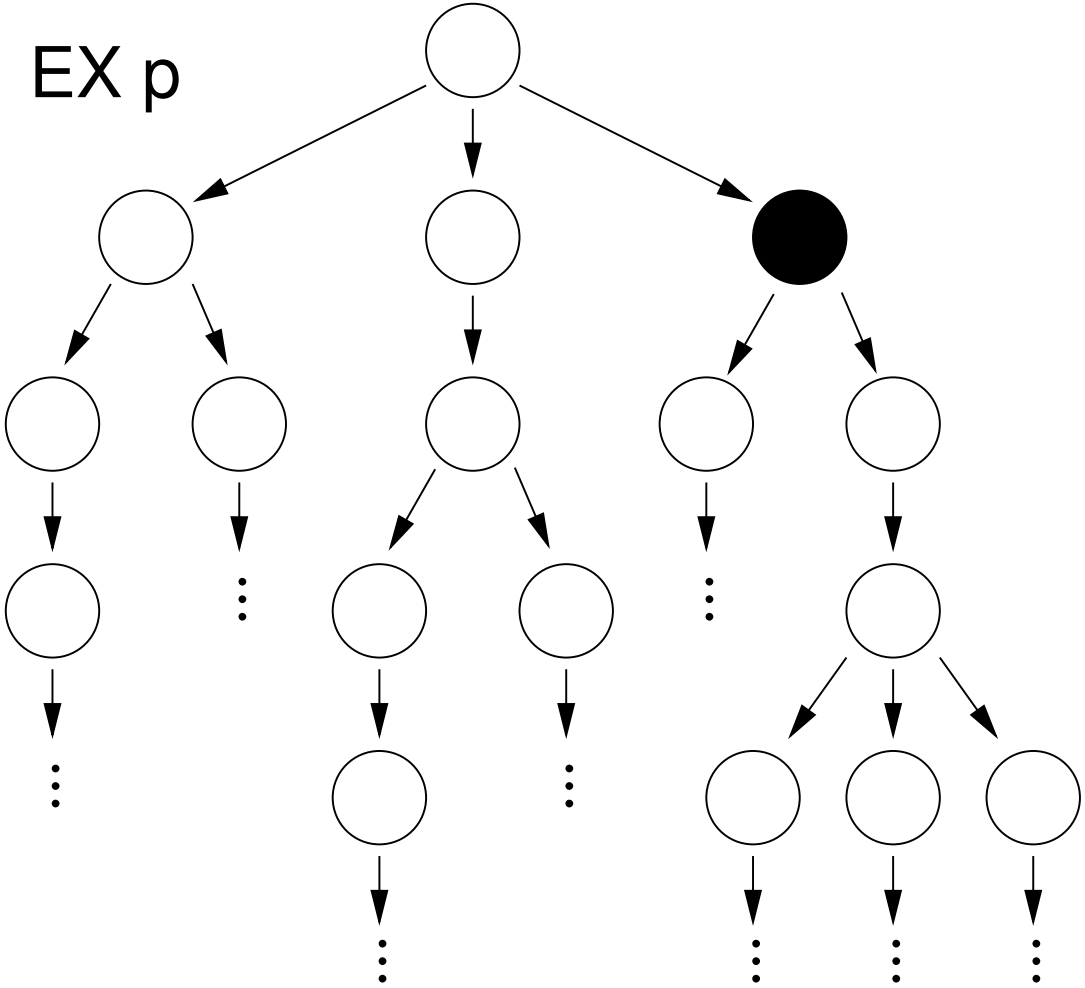

AF p

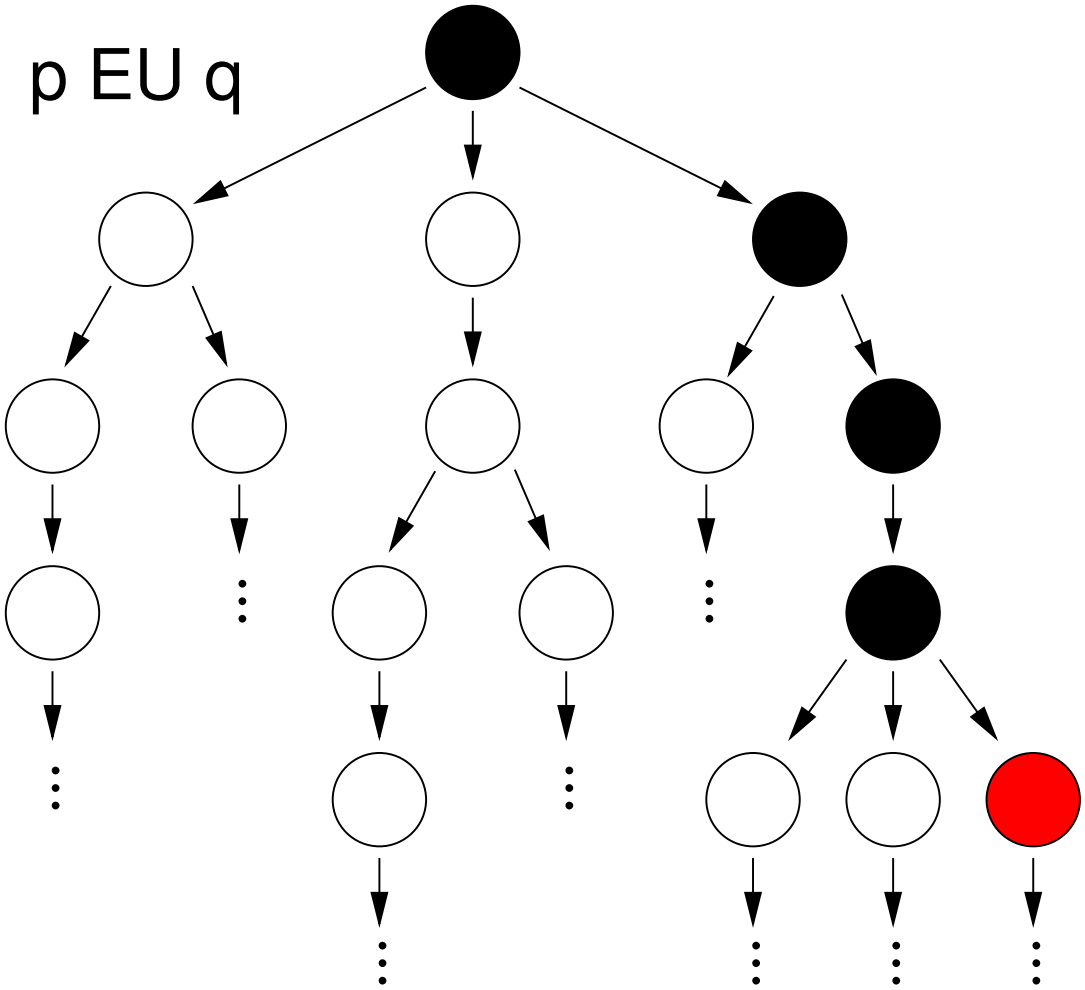

AX p


p AU q

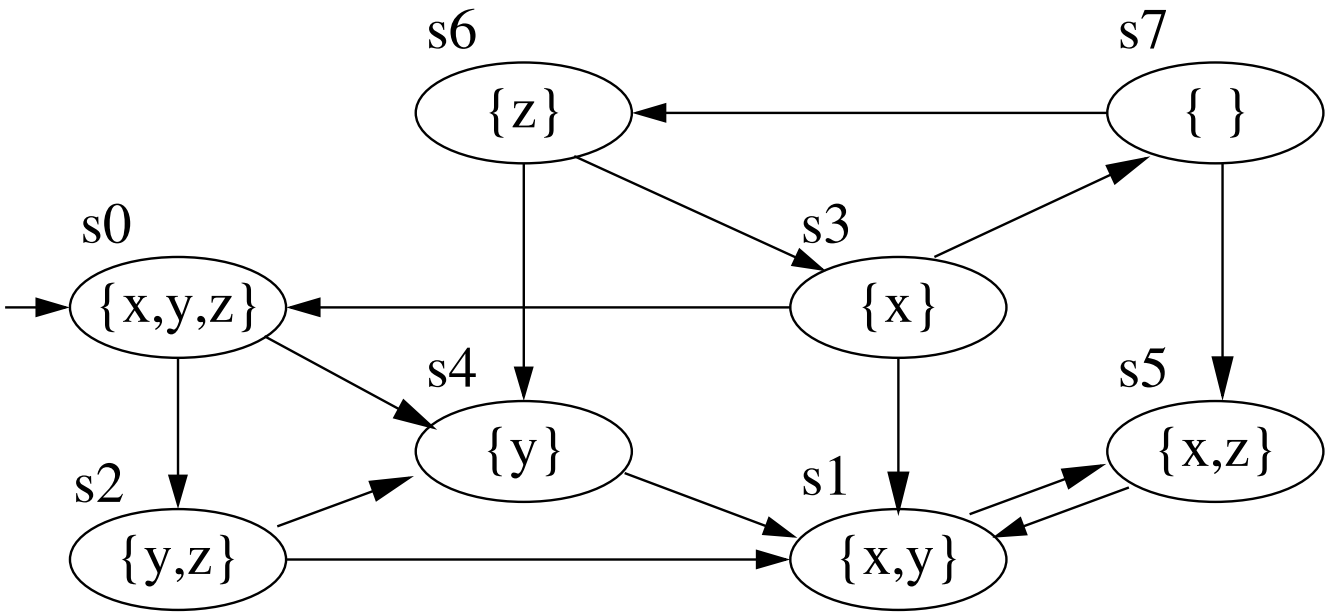

EG p


EF p


EX p


p EU q

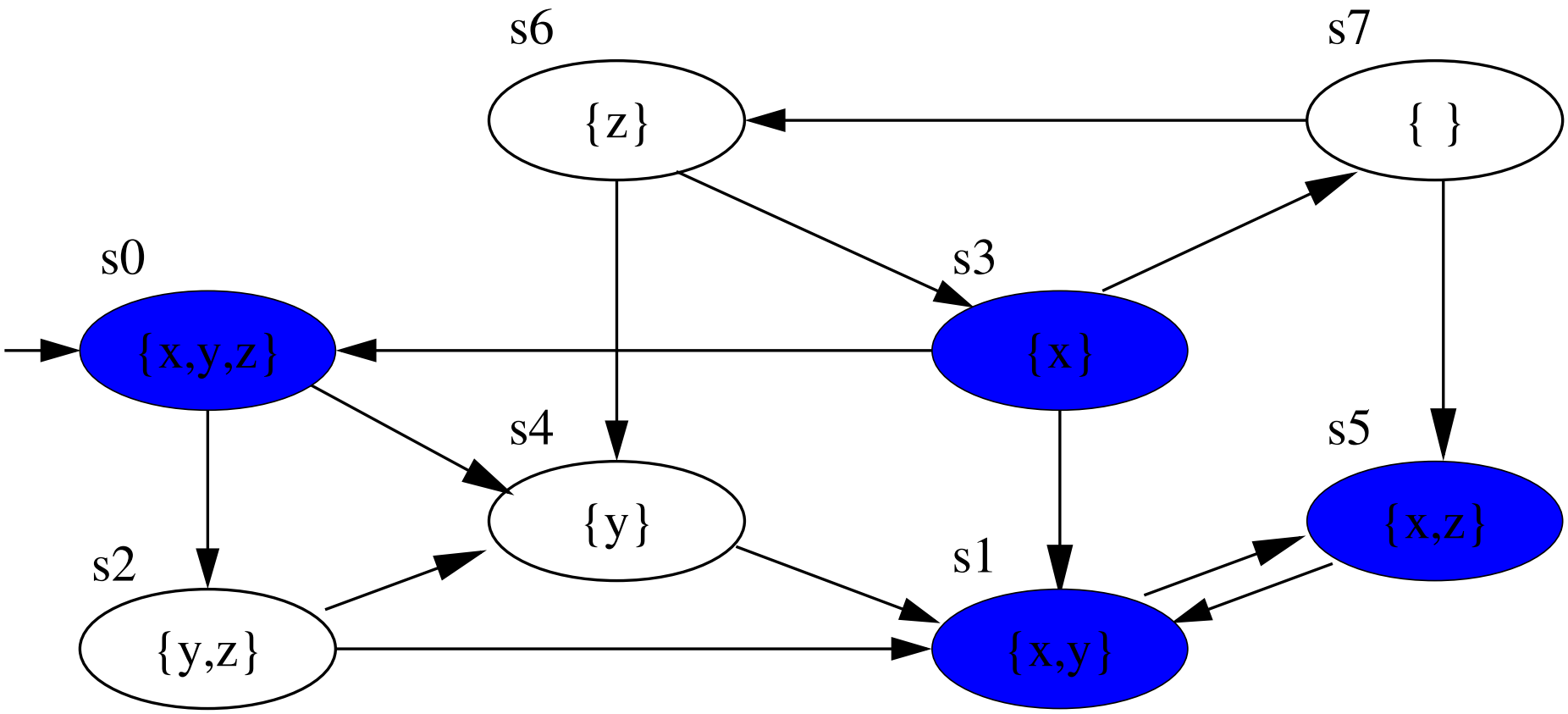
Résoudre les formules imbriquées : $s_0 \in [\![\mathbf{AF}\,\mathbf{AG}\,x]\!]$?



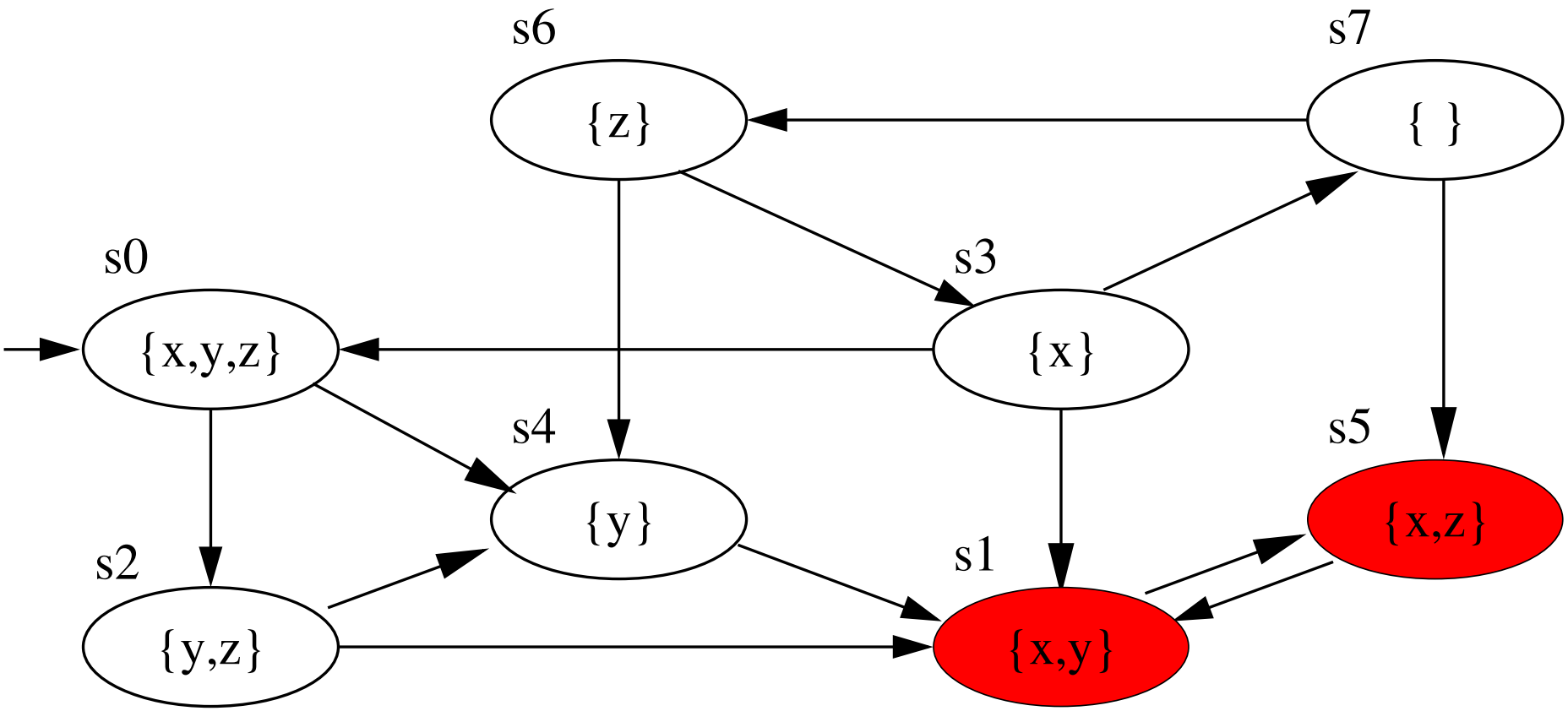
On calcule d'abord les états satisfaisant les sous-formules intérieures, ensuite progressivement les formules plus complexes.

Dans cet exemple, on commence par $[\![x]\!]$, puis $[\![\mathbf{AG}\,x]\!]$, et enfin $[\![\mathbf{AF}\,\mathbf{AG}\,x]\!]$.

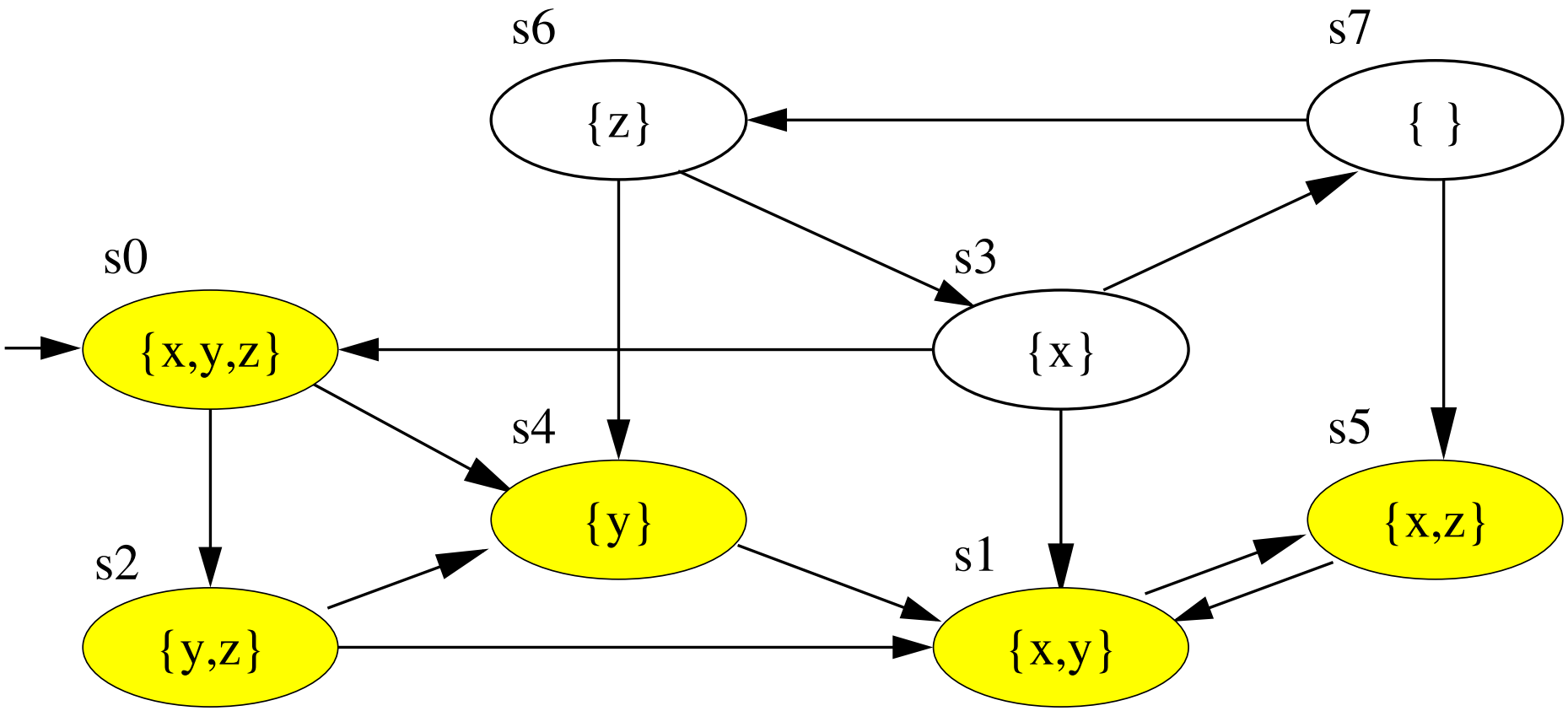
Méthode “bottom-up” : calcul de $[\![x]\!]$



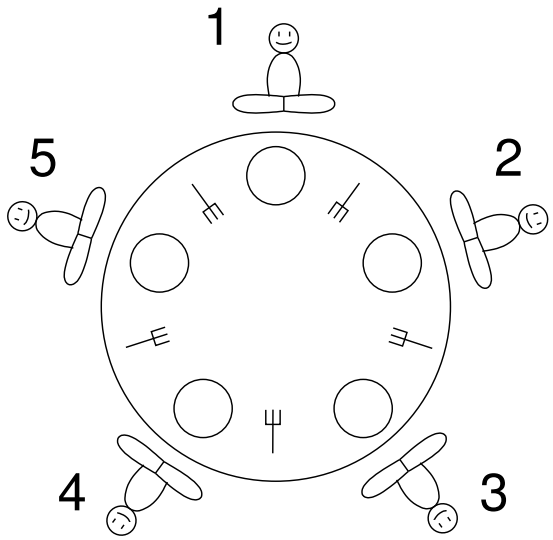
Méthode “bottom-up” : calcul de $[\![\mathbf{AG}\ x]\!]$



Méthode “bottom-up” : calcul de $[\![\mathbf{AF\ AG}\ x]\!]$



Exemple: Diner des philosophes



On a plusieurs philosophes autour d'une table qui mangent et pensent.

On exprimera quelques propriétés en CTL, à l'aide des prédicats suivants:

$e_i \widehat{=}$ le philosophe i mange

Diner des philosophes : propriétés

“Les philosophes 1 et 4 ne mangent jamais en même temps.”

Diner des philosophes : propriétés

“Les philosophes 1 et 4 ne mangent jamais en même temps.”

$$\mathbf{AG}\,\neg(e_1 \wedge e_4)$$

“Il est possible que le philosophe 3 ne mange jamais.”

Diner des philosophes : propriétés

“Les philosophes 1 et 4 ne mangent jamais en même temps.”

$$\mathbf{AG}\,\neg(e_1 \wedge e_4)$$

“Il est possible que le philosophe 3 ne mange jamais.”

$$\mathbf{EG}\,\neg e_3$$

“On peut toujours atteindre un état où seul le philosophe 2 mange.”

Diner des philosophes : propriétés

“Les philosophes 1 et 4 ne mangent jamais en même temps.”

$$\mathbf{AG}\,\neg(e_1 \wedge e_4)$$

“Il est possible que le philosophe 3 ne mange jamais.”

$$\mathbf{EG}\,\neg e_3$$

“On peut toujours atteindre un état où seul le philosophe 2 mange.”

$$\mathbf{AG}\,\mathbf{EF}(\neg e_1 \wedge e_2 \wedge \neg e_3 \wedge \neg e_4 \wedge \neg e_5)$$

Partie 7: Algorithmes pour CTL

Algo de model-checking pour CTL

Dans le suivant, soit $\mathcal{K} = (S, \rightarrow, r, AP, \nu)$ une SK (avec S fini) et ϕ une formule de CTL sur AP.

On va résoudre le problème de model-checking *global* pour CTL, à savoir calculer $[\![\phi]\!]_{\mathcal{K}}$ (tous les états de $\mathcal{K}$ dont l'arbre de calcul satisfait ϕ).

Cette solution met en œuvre les détails de la méthode "bottom-up".

Ici, on ne considère que la syntaxe minimale ; pour une meilleure efficacité on peut traiter d'autres cas directement.

Algorithme ‘bottom-up’ pour CTL

L’algorithme réduit ϕ progressivement vers un seul prédicat. Rappel: $[\![p]\!]_{\mathcal{K}} = \{\, s \mid p \in \nu(s) \,\}$ pour $p \in AP$. Dans le suivant, on denote cet ensemble par $\mu(p)$.

1. Vérifier si $\phi = p$, où $p \in AP$. Dans ce cas, on termine avec $\mu(p)$.

2. Sinon, ϕ contient une sous-formule ψ de la forme $\neg p$, $p \vee q$, $\mathbf{EX}\, p$, $\mathbf{EG}\, p$ ou $p\, \mathbf{EU}\, q$, avec $p, q \in AP$. On calcule $[\![\psi]\!]_{\mathcal{K}}$ à l’aide des transparents suivants.

3. Soit $p' \notin AP$ un prédicat “frais”. Ajouter p' à AP avec $\mu(p') := [\![\psi]\!]_{\mathcal{K}}$. Ensuite, on remplace toute occurrence de ψ dans ϕ par p' et on itère.

Calcul de $[\![\psi]\!]_{\mathcal{K}}$: cas simples

Cas 1: $\psi \equiv \neg p, \quad p \in AP$

Par définition, $[\![\psi]\!]_{\mathcal{K}} = S \setminus \mu(p)$.

Cas 2: $\psi \equiv p \vee q, \quad p, q \in AP$

Alors $[\![\psi]\!]_{\mathcal{K}} = \mu(p) \cup \mu(q)$.

Cas 3: $\psi \equiv \mathbf{EX}\, p, \quad p \in AP$

Écrivons $pre(X)$, pour $X \subseteq S$, pour l'ensemble

$$pre(X) := \{\, s \mid \exists t \in X : s \to t \,\}.$$

Alors par définition $[\![\psi]\!]_{\mathcal{K}} = pre(\mu(p))$.

Calcul de $[\![\psi]\!]_{\mathcal{K}}$: $\mathbf{EU}$ et $\mathbf{EG}$

D'abord on caractérise $\mathbf{EU}$ et $\mathbf{EG}$ par des points fixes.

$\mathbf{EU}$ correspond à un point fixe minimal: On démarre avec l'hypothèse qu'aucun état ne satisfait la modalité EU, ensuite on identifie ces états un par un.

Au contraire, $\mathbf{EG}$ est un point fixe maximal: On démarre avec tous les états, puis on élimine ceux que ne satisfont surement pas la modalité EG.

Les points fixes nous donnent donc des algorithmes pour le calcul récursif.

Calcul de $[\![\psi]\!]_{\mathcal{K}}$: $\mathbf{EG}$

Cas 4: $\psi \equiv \mathbf{EG}\, p$, $\quad p \in AP$

Lemme 1: $[\![\mathbf{EG}\, p]\!]_{\mathcal{K}}$ est la solution maximale (avec $\subseteq$) de

$$X = \mu(p) \cap pre(X).$$

Preuve: On procède en deux étapes:

1. On montre que $[\![\mathbf{EG}\, p]\!]_{\mathcal{K}}$ est effectivement une solution possible de cette équation:

$$[\![\mathbf{EG}\, p]\!]_{\mathcal{K}} = \mu(p) \cap pre([\![\mathbf{EG}\, p]\!]_{\mathcal{K}}).$$

Rappel: $[\![\mathbf{EG}\, p]\!]_{\mathcal{K}} = \{\, s \mid \exists \rho\colon \rho(0) = s \wedge \forall i \geq 0\colon \rho(i) \in \mu(p) \,\}$.

"$\Rightarrow$" Soit $s \in [\![\mathbf{EG}\, p]\!]_{\mathcal{K}}$ et ρ un chemin "temoin". Alors évidemment $s \in \mu(p)$. D'ailleurs, $\rho(1) \in [\![\mathbf{EG}\, p]\!]_{\mathcal{K}}$ (en raison de ρ^1), du coup $s \in pre([\![\mathbf{EG}\, p]\!]_{\mathcal{K}})$.

Continuation de la preuve du Lemme 1:

1. "$\Leftarrow$" Soit $s \in \mu(p) \cap pre([\![\mathbf{EG}\, p]\!]_{\mathcal{K}})$. Alors s possède un successeur direct t, qui est le point de départ d'un chemin ρ témoin pour $t \in [\![\mathbf{EG}\, p]\!]_{\mathcal{K}}$. Du coup, $s\rho$ est un chemin témoin prouvant que $s \in [\![\mathbf{EG}\, p]\!]_{\mathcal{K}}$.

2. On montre maintenant que $[\![\mathbf{EG}\, p]\!]_{\mathcal{K}}$ est la solution *la plus grande*, autrement dit si M est une solution de l'équation, alors $M \subseteq [\![\mathbf{EG}\, p]\!]_{\mathcal{K}}$.

Soit $M \subseteq S$ une solution, alors $M = \mu(p) \cap pre(M)$. Soit $s \in M$. On montre que $s \in [\![\mathbf{EG}\, p]\!]_{\mathcal{K}}$.

- $s \in M$, alors $s \in \mu(p)$ et $s \in pre(M)$.
- $s \in pre(M)$, du coup il existe $s_1 \in M$ avec $s \to s_1$.
- Par itération, on construit un chemin infini $\rho = ss_1 \cdots$ dans lequel tous les états sont dans $\mu(p)$. Du coup, $s \in [\![\mathbf{EG}\, p]\!]_{\mathcal{K}}$.

Lemme 2: On considère la séquence $S,\ \pi(S),\ \pi(\pi(S)),\ \ldots$, où $\pi(X) := \mu(p) \cap pre(X)$. Pour tout $i \geq 0$ on a $\pi^i(S) \supseteq [\![\mathbf{EG}\, p]\!]_{\mathcal{K}}$.

Les deux propriétés suivantes sont évidentes:

(1) π est *monotone*: si $X \supseteq X'$, alors $\pi(X) \supseteq \pi(X')$.

(2) La séquence est *descendante*: $S \supseteq \pi(S) \supseteq \pi(\pi(S)) \ldots$ (conséquence de (1)).

Preuve du Lemme 2: (par récurrence sur i)
Base: $i = 0$: évident.
Récurrence: $i \rightarrow i + 1$:

$$\begin{aligned} \pi^{i+1}(S) &= \mu(p) \cap pre(\pi^i(S)) \\ &\supseteq \mu(p) \cap pre([\![\mathbf{EG}\, \varphi]\!]_{\mathcal{K}}) \quad \text{(hypothèse et monotonicité)} \\ &= [\![\mathbf{EG}\, p]\!]_{\mathcal{K}} \end{aligned}$$

Lemma 3: Il existe un i tel que $\pi^i(S) = \pi^{i+1}(S)$, et $[\![\mathbf{EG}\, p]\!]_{\mathcal{K}} = \pi^i(S)$.

Preuve: Comme S est fini, la séquence descendante doit atteindre un point fixe, disons après i étapes. Alors on a $\pi^i(S) = \pi(\pi^i(S)) = \mu(p) \cap pre(\pi^i(S))$. Du coup, $\pi^i(S)$ est une solution de l'équation du Lemme (1).

Par le Lemme 1, on a $\pi^i(S) \subseteq [\![\mathbf{EG}\, p]\!]_{\mathcal{K}}$.
Par le Lemme 2, on a $\pi^i(S) \supseteq [\![\mathbf{EG}\, p]\!]_{\mathcal{K}}$.

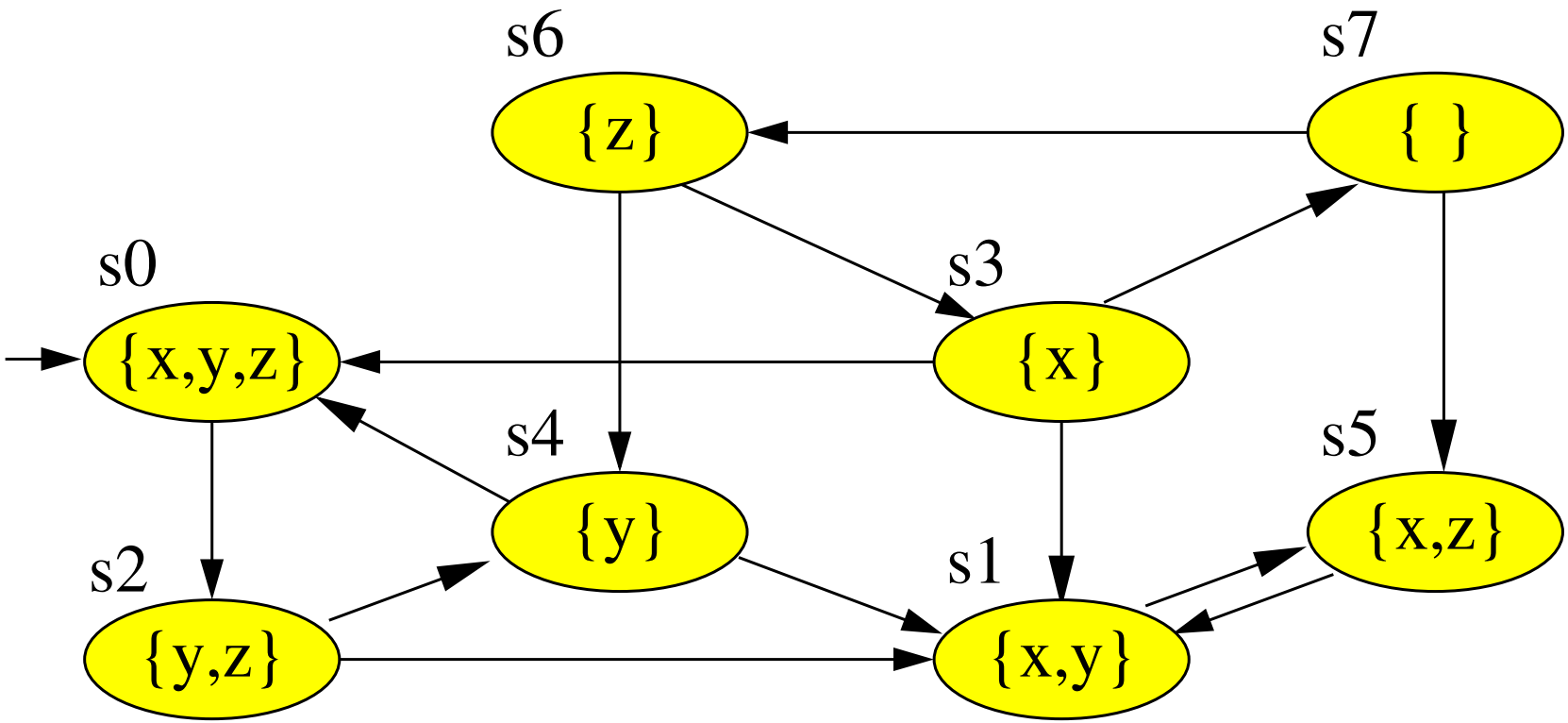
Un algorithme pour $\mathbf{EG}$

Le Lemme 3 nous donne une stratégie pour calculer $[\![\mathbf{EG}\, p]\!]_{\mathcal{K}}$: on calcule la séquence $S,\ \pi(S),\ \cdots$ jusqu'à ce qu'on atteint un point fixe.

En pratique, on démarre avec $X := \mu(p)$. Puis, à chaque tour, on élimine les états sans successeur dans X.

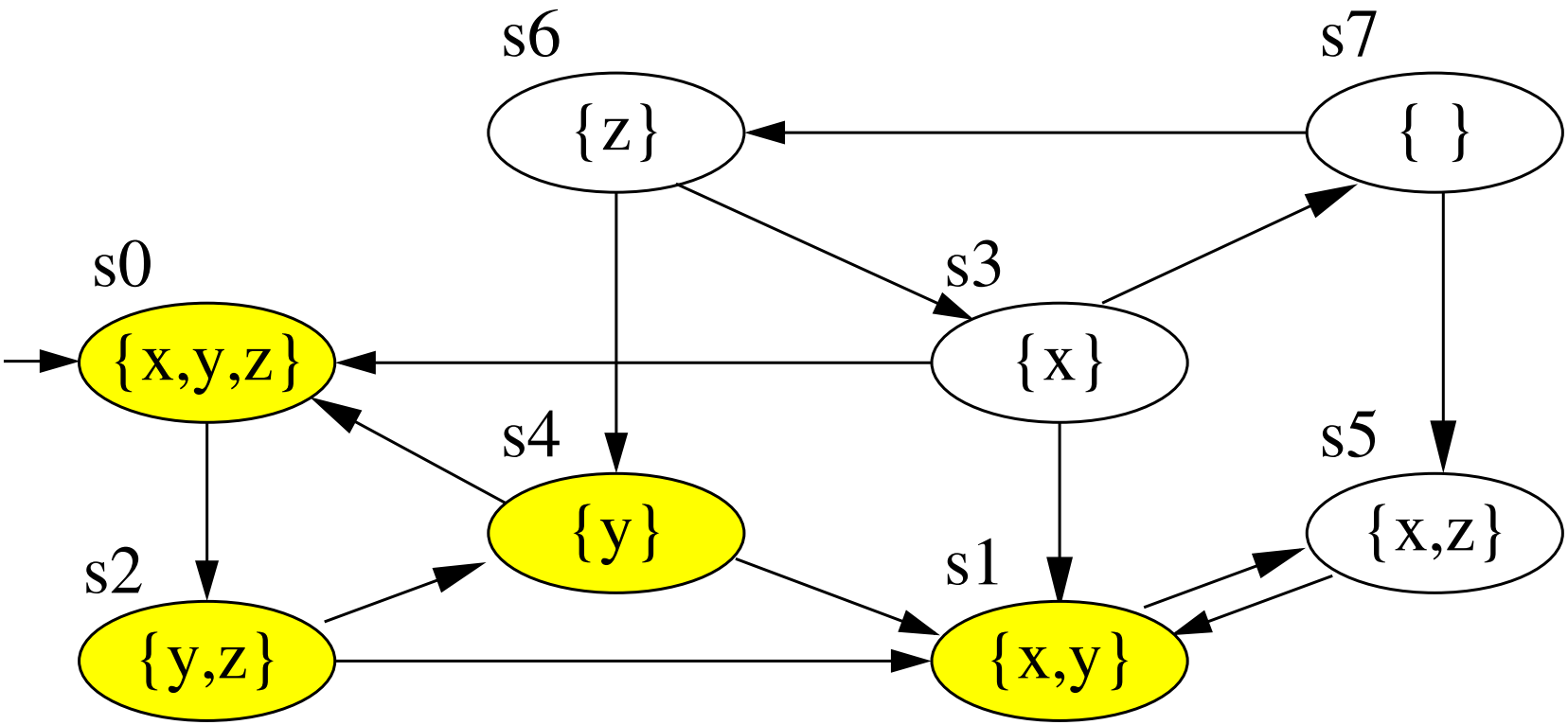
Ceci peut être réalisé en temps $\mathcal{O}(|\mathcal{K}|)$ (compter les successeurs dans X).

Exemple: Calcul de $[\![\mathbf{EG}\ y]\!]_{\mathcal{K}}$ (1/4)



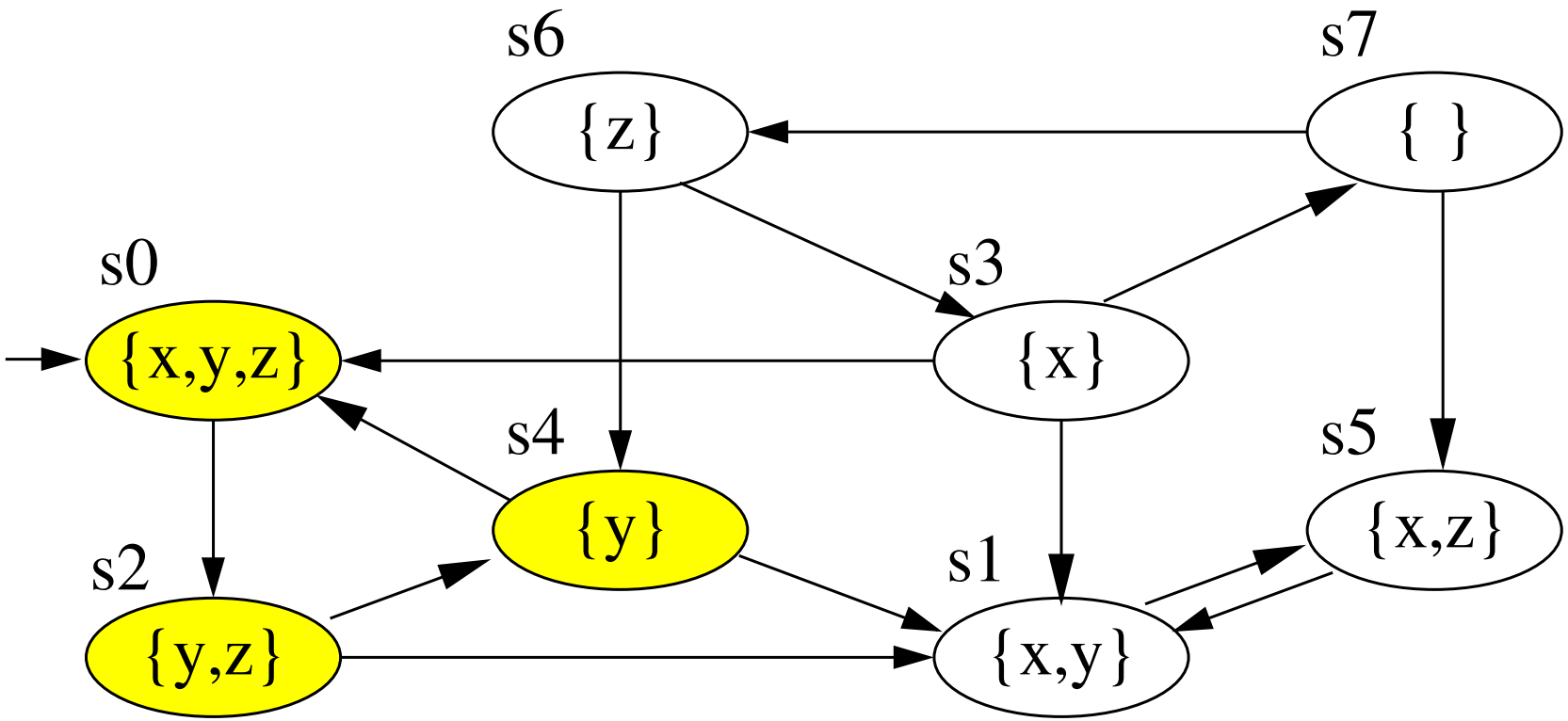
$\pi^0(S) \;=\; S$

Exemple: Calcul de $[\![\mathbf{EG}\ y]\!]_{\mathcal{K}}$ (2/4)



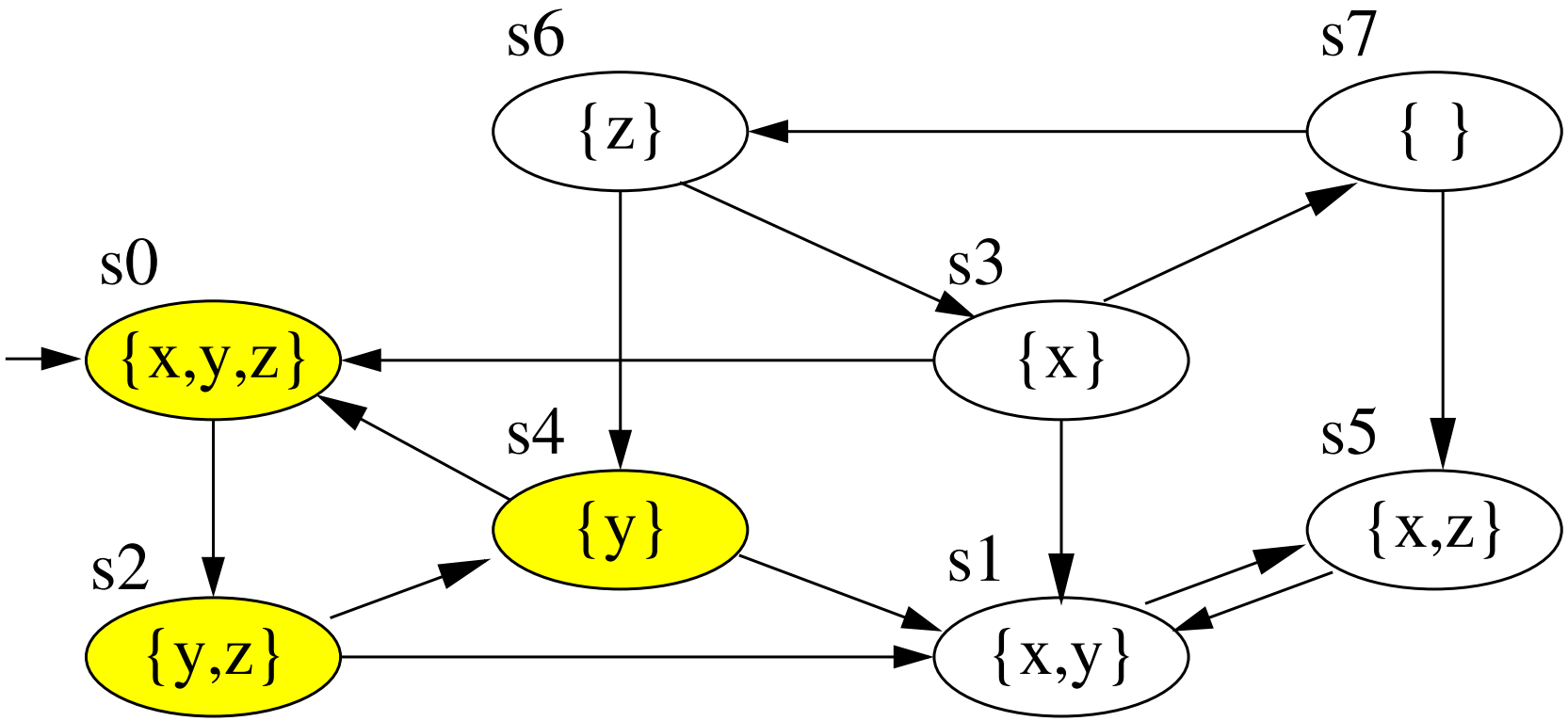
$\pi^1(S) \quad = \quad \mu(y) \cap pre(S)$

Exemple: Calcul de $[\![\mathbf{EG}\ y]\!]_{\mathcal{K}}$ (3/4)



$\pi^2(S) \;=\; \mu(y) \cap pre(\pi^1(S))$

Exemple: Calcul de $[\![\mathbf{EG}\, y]\!]_{\mathcal{K}}$ (4/4)



$\pi^3(S) \;=\; \mu(y) \cap pre(\pi^2(S)) \;=\; \pi^2(S)$: $[\![\mathbf{EG}\, y]\!]_{\mathcal{K}} = \{s_0, s_2, s_4\}$

Calcul de $\mathbf{EU}$

Cas 5: $\psi \equiv p\,\mathbf{EU}\,q$, $\quad p, q \in AP$

Analogue à $\mathbf{EG}$ (on omet les preuves):

Lemme 4: $[\![p\,\mathbf{EU}\,q]\!]_{\mathcal{K}}$ est la solution minimale (avec $\subseteq$) de

$$X = \mu(q) \cup (\mu(p) \cap pre(X)).$$

Lemme 5: $[\![p\,\mathbf{EU}\,q]\!]_{\mathcal{K}}$ est le point fixe de la séquence

$$\emptyset,\ \ \xi(\emptyset),\ \ \xi(\xi(\emptyset)),\ \ \ldots \text{où} \quad \xi(X) := \mu(q) \cup (\mu(p) \cap pre(X))$$

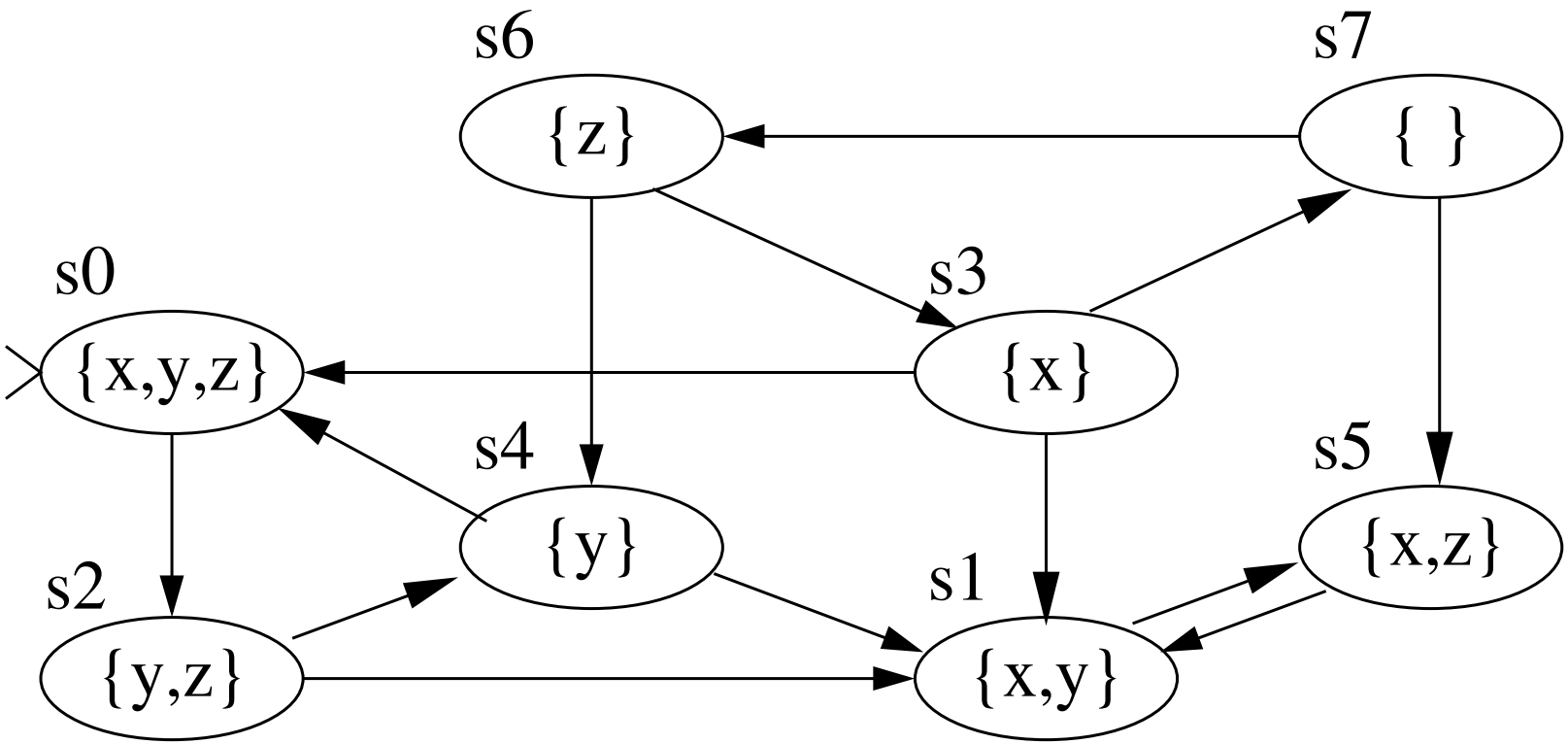
Un algorithme pour $\mathbf{EU}$

Le Lemme 5 propose une stratégie: Calculer la séquence $\emptyset,\ \xi(\emptyset),\ \cdots$ pour atteindre un point fixe.

On démarre avec $X := \mu(q)$; ensuite on ajoute les prédecesseurs directs qui sont dans $\mu(p)$.

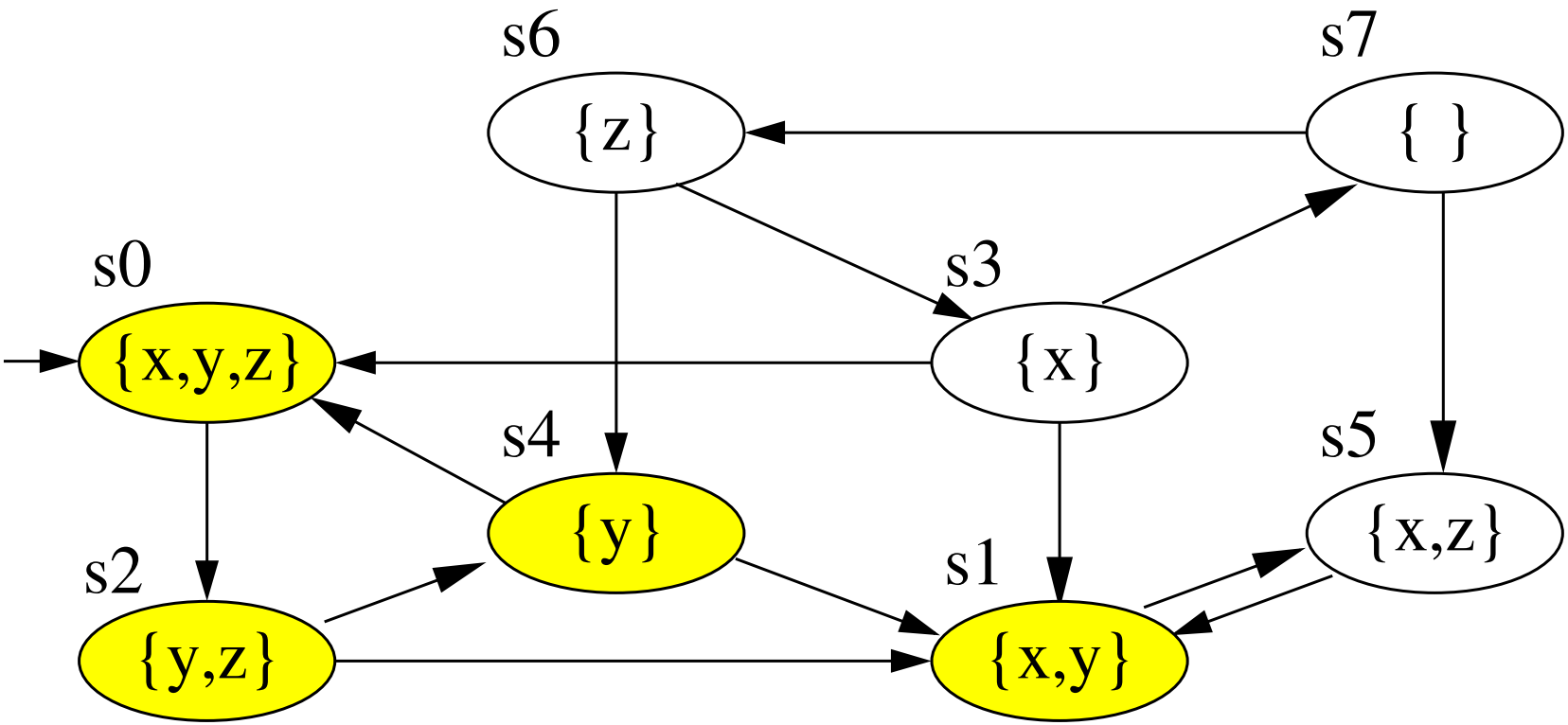
Ceci peut être effectué en temps $\mathcal{O}(|\mathcal{K}|)$
(traversée de graphe par profondeur/largeur en arrière).

Exemple: Calcul de $[\![z \ \mathbf{EU} \ y]\!]_{\mathcal{K}}$ (1/4)



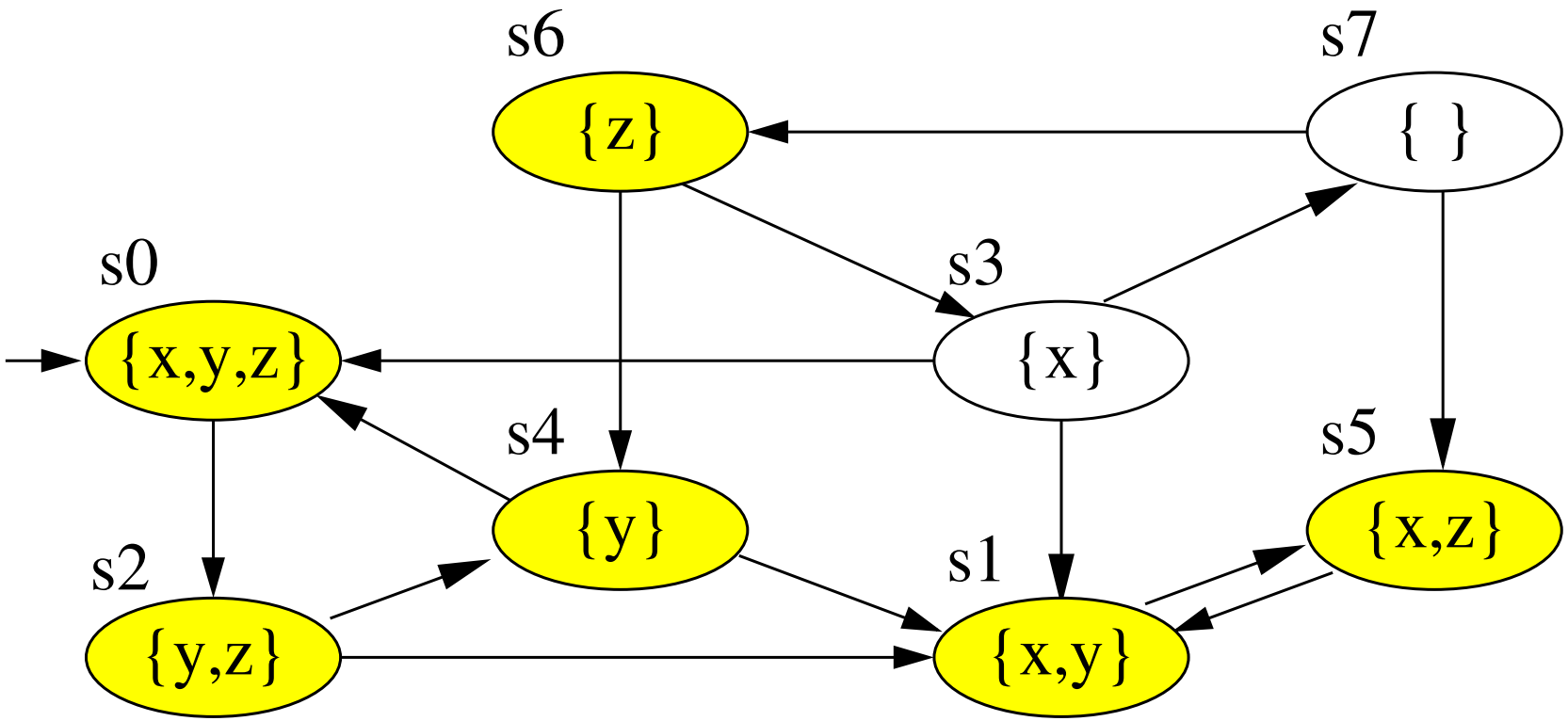
$\xi^0(\emptyset) \ = \ \emptyset$

Exemple: Calcul de $[\![z \ \mathbf{EU} \ y]\!]_{\mathcal{K}}$ (2/4)



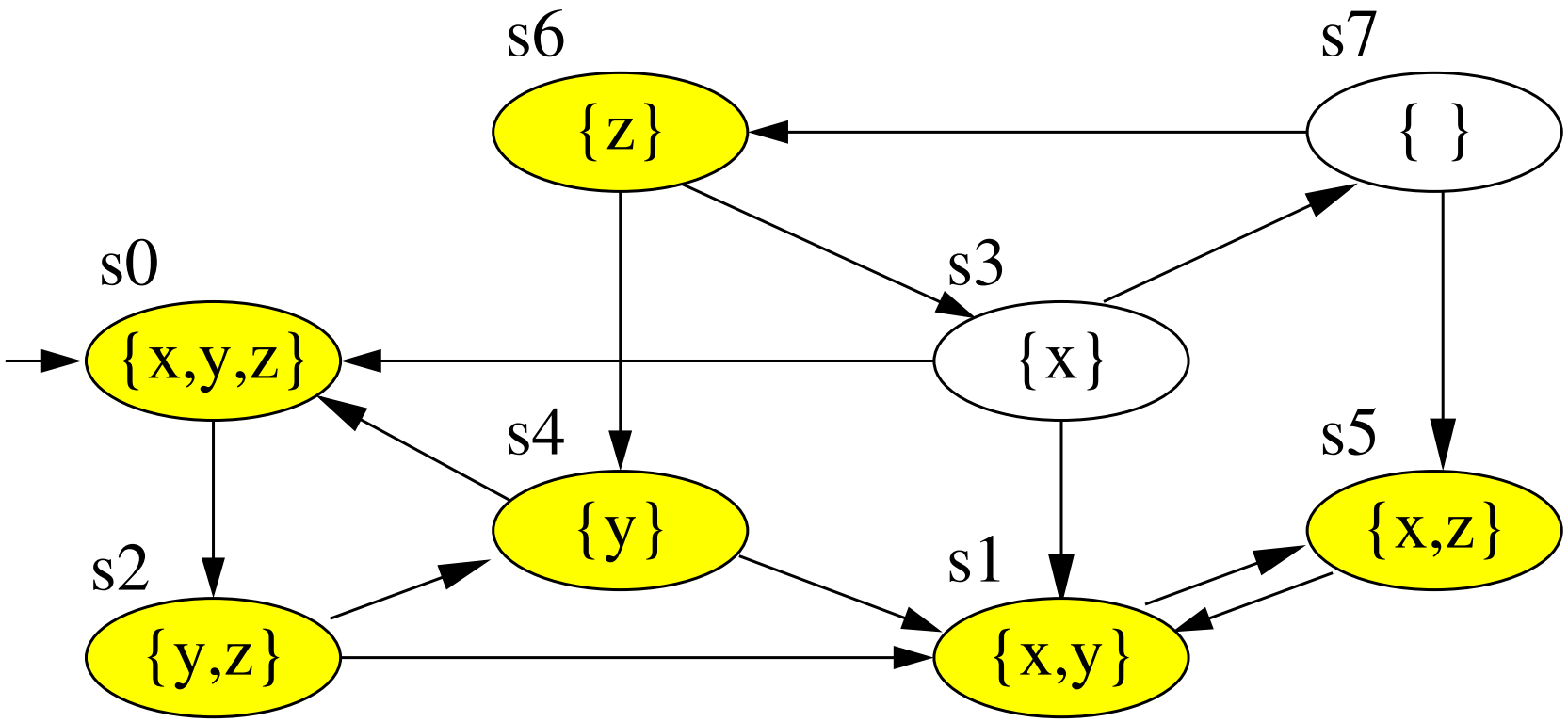
$\xi^1(\emptyset) \ = \ \mu(y) \cup (\mu(z) \cap pre(\xi^0(\emptyset)))$

Exemple: Calcul de $[\![z \ \mathbf{EU} \ y]\!]_{\mathcal{K}}$ (3/4)



$\xi^2(\emptyset) \ = \ \mu(y) \cup (\mu(z) \cap pre(\xi^1(\emptyset)))$

Exemple: Calcul de $[\![z\ \mathbf{EU}\ y]\!]_{\mathcal{K}}$ (4/4)



$\xi^3(\emptyset) \;=\; \mu(y) \cup (\mu(z) \cap pre(\xi^2(\emptyset))) \;=\; \xi^2(\emptyset)$

$[\![z\ \mathbf{EU}\ y]\!]_{\mathcal{K}} = \{s_0, s_1, s_2, s_4, s_5, s_6\}$

Comparaison de CTL et LTL

Pas mal de propriétés s'expriment dans les deux logiques, p.ex.:

Invariants ("p ne tient jamais.")

$$\mathbf{AG}\,\neg p \qquad \text{ou} \qquad \mathbf{G}\,\neg p$$

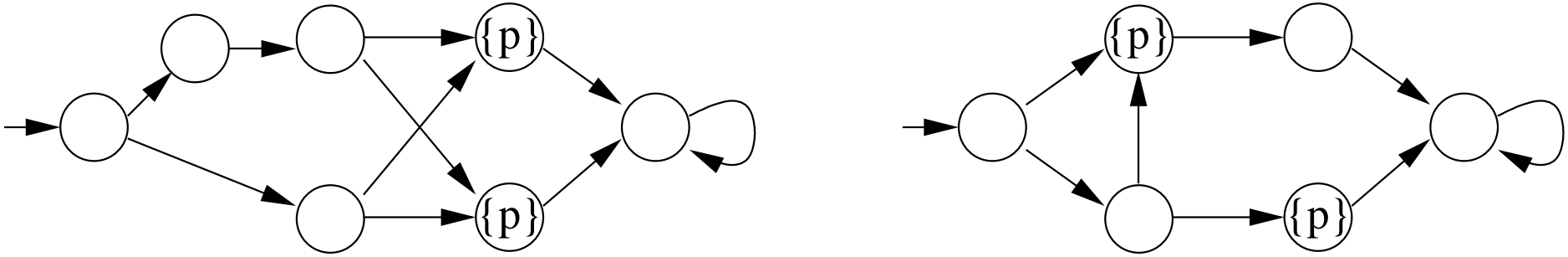
Réactivité ("Tout p est suivi d'un q.")

$$\mathbf{AG}(p \rightarrow \mathbf{AF}\,q) \qquad \text{ou} \qquad \mathbf{G}(p \rightarrow \mathbf{F}\,q)$$

CTL considère l'arbre de calcul entier, LTL les calculs individuels. Du coup, CTL peut raisonner sur les *possibilities*, ce qui est impossible dans LTL. Exemples:

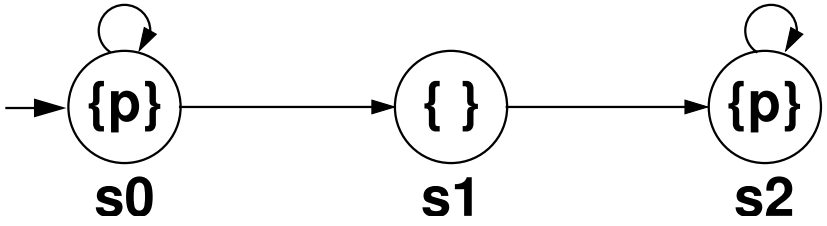
La formule $\mathbf{AG}\,\mathbf{EF}\,p$ ("reset") n'est pas expressible dans LTL.

$\mathbf{AF}\,\mathbf{AX}\,p$ distingue entre ces deux structures, mais pas $\mathbf{F}\,\mathbf{X}\,p$:



Inversement, LTL arrive à exprimer certaines propriétés non-expressibles en CTL. Du coup, les deux logiques sont *incomparable* dans leur pouvoir d'expression :

$\mathbf{F}\,\mathbf{G}\,p$ n'est pas expressible dans CTL:



$\mathcal{K} \models \mathbf{F}\,\mathbf{G}\,p$ mais $\mathcal{K} \not\models \mathbf{AF}\,\mathbf{AG}\,p$

(Emerson, Halpern 1986)

Complexité

Model-checking pour CTL: $\mathcal{O}(|\mathcal{K}| \cdot |\phi|)$
Model-checking pour LTL: $\mathcal{O}(|\mathcal{K}| \cdot 2^{|\phi|})$

Est-ce que ça veut dire que CTL est plus efficace?

Réponse: pas nécessairement

LTL a certains avantages algorithmiques (p.ex. exploration à la volée et une seule traversée de $\mathcal{K}$).

CTL: toute la structure doit être traversée plusieurs fois.

On va étudier une amélioration algorithme pour CTL (les BDD).

Littérature

Papier de recherche:

M. Vardi, *Branching vs. Linear Time: Final Showdown*, 2001

Specification Patterns Database:

`https://matthewbdwyer.github.io/psp/`

Démonstration d'un outil: Spin

L'outil Spin

Spin est un outil de model-checking écrit par Gerard Holzmann aux Bell Labs.

Lauréat du ACM Software System Award en 2002

URL: http://spinroot.com
, sources: https://github.com/nimble-code/Spin

Littérature: Holzmann, The Spin Model Checker

Modélisation avec Spin

Descriptif du système en Promela (Protocol Meta Language)

Adapté pour spécifier des systèmes finis

Processus concurrents, communication synchrone/asynchrone, variables et d'autres types de données finis

Model-checking pour LTL (avec fairness)

Exemple: Algorithme de Dekker

Encore un algorithme pour l'exclusion mutuel:

```
bit turn;
bool flag0, flag1;
bool crit0, crit1;

active proctype p0() {
...
}

active proctype p1() {
...
}
```

Dekker: contenu de p0

```
active proctype p0() {
again:  flag0 = true;
        do
        :: flag1 ->
                if
                :: turn == 1 ->
                        flag0 = false;
                        (turn != 1) -> flag0 = true;
                :: else -> skip;
                fi
        :: else -> break;
        od;

        crit0 = true; /* critical section */ crit0 = false;

        turn = 1; flag0 = false;
        goto again;
}
```

p1: identique, mais en échangeant les 0 et 1

Promela I

Déclarations de variables:

```
bit turn;
bool flag0, flag1;
bool crit0, crit1;
```

`turn` prend des valeurs 0 et 1.

`flag1` prend des valeurs `true` et `false`.

valeurs initiaux: à défaut, 0 et `false`, resp.

D'autres types : `byte`, types défini par utilisateur, ...

Promela II

Déclaration d'un processus:

```
active proctype p0() {
...
}
```

`proctype` définit un *type* de processus. `active` veut dire qu'il en existe une instance initialement. On peut aussi en créer plusieurs :

```
active [2] proctype my_process() {
...
}
```

Le comportement du système est construit par entrelacement: chaque transition du système est une transition d'un seul processus, les autres restent inchangés.

Promela III

étiquetes / affectations / sauts

```
again:  flag0 = true;
...
        goto again;
```

instruction vide

```
skip
```

Promela IV

Boucle:

```
do
:: flag1 -> ...
:: else -> break;
od;
```

`flag1` et `else` sont des “gardes”

L’exécution prend une branche de façon non-déterministe si plusieurs gardes sont satisfaites.

La branche `else` n’est prise que si aucune garde n’est satisfaite.

`break` quitte un bloc de `do`.

Promela V

Branchement:

```
if
:: turn == 1 -> ...
:: else -> ...;
fi
```

Même syntaxe et semantique que `do` mais sans répétition.

```
(turn != 1) -> ...
```

Instruction gardée : on bloque jusqu'à ce que la garde est satisfaite.

Model checking avec Spin

Dans l'algorithme de Dekker l'exclusion mutuelle sur les zone critiques doit être respectée:

$$\mathbf{G}\ \neg(\texttt{crit0} \wedge \texttt{crit1})$$

(où les prédicates `crit0` et `crit1` veulent dire que les variables correspondants sont vraies).

Syntaxe dans Spin: `[] !(crit0 && crit1)`

Tester la propriété dans Spin (script `spinLTL`):

Formule satisfaite !

Model checking avec Spin

On veut aussi qu'un processus réussisse à entrer dans la zone critique :

$$\mathbf{G}(\texttt{flag0} \rightarrow \mathbf{F}\,\texttt{crit0})$$

Syntaxe : `[] (flag0 -> <> crit0)`

On teste la propriété :

Pas satisfaite !

Model checking avec fairness

La formule précédent n'est pas satisfaite si `p1` n'a pas de temps pour remettre `flag1` à `false`.

Une telle exécution n'est pas "juste".

Hypothèse de justesse (fairness) : On ne considère que les exécutions telles que tout processus peut progresser infiniment souvent.

Spin possède une option spéciale pour cela (voir spinFairLTL).

Partie 8: Diagrammes de Décision Binaires

Ensembles d'états

On a vu que les algorithmes pour model-checking de CTL s'expriment naturellement comme des opérations sur des *ensembles d'états*:

états satisfaisant un prédicat : $\mu(p)$ pour $p \in AP$

états satisfaisant une sous-formule : $[\![\psi]\!]_{\mathcal{K}}$

calcul par opérations ensemblistes : pre, $\cap$, $\cup$, ...

Comment représenter de tels ensembles :

liste explicite : $S = \{s_1, s_2, s_4, \ldots\}$

représentation symbolique: notation ou structure de données compacte

Ici: On s'intéressera à représenter des états avec variables booléennes.

Soit S un ensemble de vecteurs booléens:

$$S = \{0,1\}^m \qquad \text{pour un } m \geq 1$$

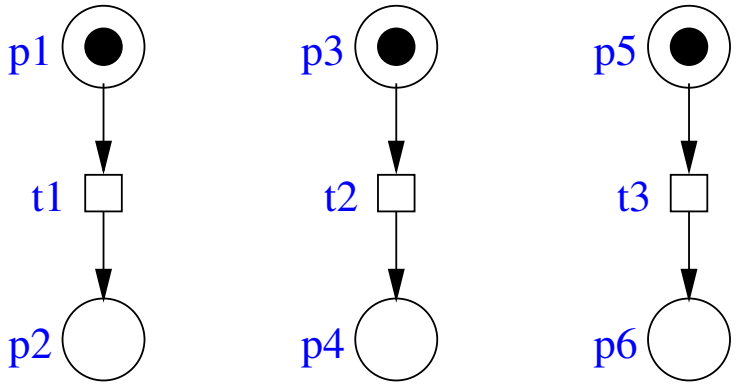
Exemples :

programmes avec variables booléennes

circuits logiques (tout signal est 0 ou 1)

En principe, *tout* espace d'état finis peut être représenté ainsi, mais cela n'est pas toujours adéquat.

Exemple 1: Automates concurrents

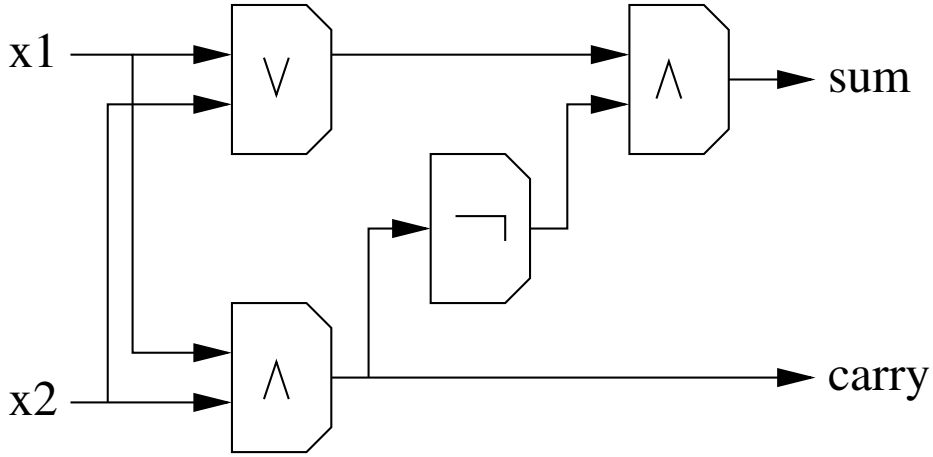


Un état s'écrit comme $(p_1, p_2, \ldots, p_6)$, où p_i, $1 \leq i \leq 6$ indique que si l'un des automates se trouve dans l'état local P_i.

État initial $(1, 0, 1, 0, 1, 0)$;
d'autres états accessibles, p.ex., $(0, 1, 1, 0, 1, 0)$ ou $(1, 0, 0, 1, 0, 1)$.

Exemple 2: Circuit logique

Demi-additionneur:



Ce circuit a deux signaux d'entrée (x_1, x_2) et deux de sortie ($carry, sum$). Leurs combinaisons valables se denotent par des vecteurs de dimension 4., p.ex. $(1, 0, 0, 1)$ ($x_1 = 1$, $x_2 = 0$, $carry = 0$, $sum = 1$).

Graphes de décision binaire

Soit V un ensemble de variables et $<$ un ordre total sur V, p.ex.

$$x_1 < x_2 < \mathit{carry} < \mathit{sum}$$

(Il s'agit d'ordonner les variables elles-mêmes, pas leurs valeurs !)

Un graphe de décision binaire (sur $<$) est un graphe dirigé acyclique connexe tq:

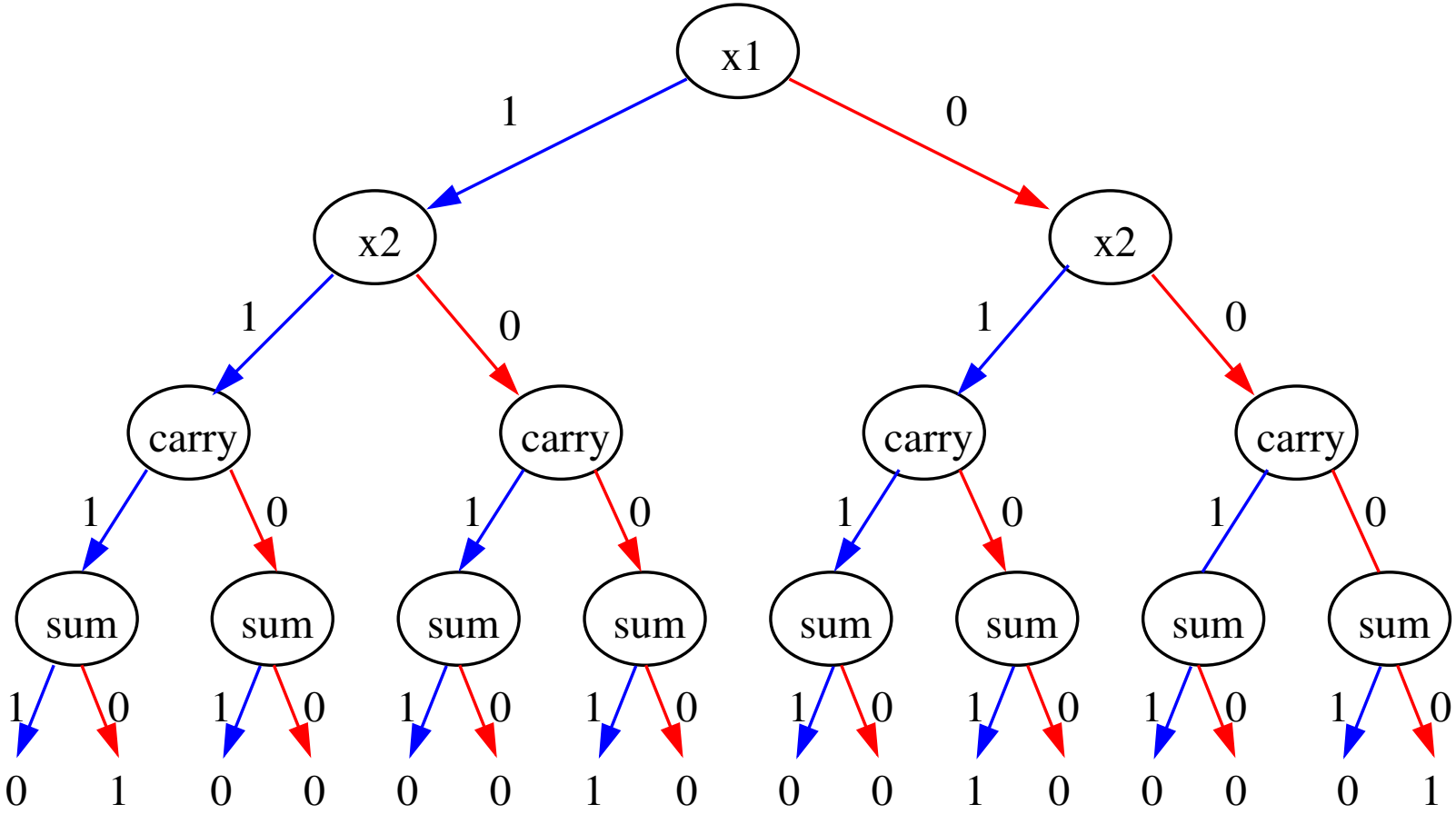
il y a une seule racine (sommet sans arête entrante);

il y a au plus deux feuilles $\mathbf{0}$ et/ou $\mathbf{1}$;

tout sommet qui n'est pas une feuille est étiqueté par une variable de V
et possède deux arêtes sortantes étiquetés par 0 et 1;

une arête d'un sommet étiqueté x vers un sommet étiqueté y implique $x < y$.

Exemple 2: Graphe de décision binaire complet



Les chemins menant à in 1 correspondent aux vecteurs qui sont dans *C*.

Diagramme de décision binaire

Un diagramme de décision binaire (binary decision diagram, BDD) et un graphe de décision binaire avec deux propriétés supplémentaires:

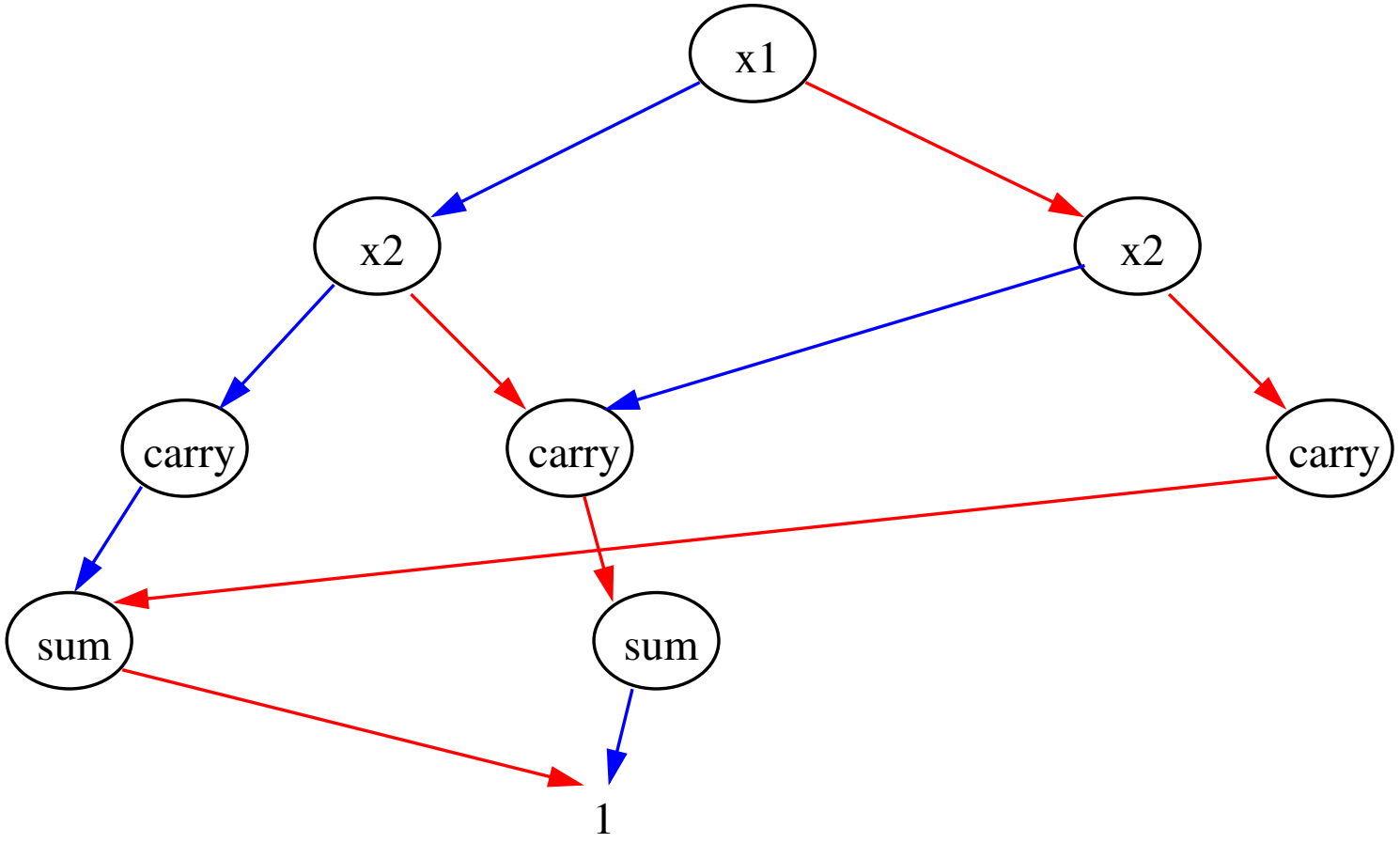
aucune paire de sous-graphes distincts n'est isomorphe;

aucun sommet n'est *redondant*: les deux arêtes sortants de tout sommet mènent à des sommets différents.

(Graphiquement, on se permet d'omettre la feuille $\mathbf{0}$ et les arêtes qui y mènent.)

Remarque: Sur les transparents suivants, les arêtes bleues sont étiquetés par 1, les arêtes rouges par 0.

Exemple 2: Diagramme de décision binaire



Voici le BDD correspondant au diagramme précédente.

Littérature et outils

Introduction compacte :

H.R. Andersen, *An Introduction to Binary Decision Diagrams*, Lecture notes, Department of Information Technology, IT University of Copenhagen

Disponible dans le web (chercher le titre).

Outils/bibliothèque:

Colorado University Decision Diagrams (CUDD library)

Plan

Dans le suivant, on étudie les opérations sur les BDDs réquises pour réaliser du model-checking:

- Construction d'un BDD (à partir d'une formule de calcul propositionnel)
- Test d'équivalence
- Intersection, complément, union
- Relations, calcul de prédécesseurs

Substitution

Soient F et G formules de CP et x un prédicat.

$F[x/G]$ note la formule obtenue en remplaçant toute occurrence de x dans F par G. En particulier, on utilisera $F[x/0]$ et $F[x/1]$.

Exemple: Soit $F = x \wedge y$. Alors $F[x/1] = 1 \wedge y \equiv y$ et $F[x/0] = 0 \wedge y \equiv 0$.

If-then-else

On va introduire un opérateur booléen *ternaire* qui s'appelle *ite* (if-then-else).

Note: *ite* n'étend par l'expressivité des formule, mais il sert comme raccourci.

Soient F, G, H des formules. On définit

$$ite(F, G, H) := (F \wedge G) \vee (\neg F \wedge H).$$

Les formules normales INF (if-then-else normal form) sont les suivantes :

- $\mathbf{0}$ et $\mathbf{1}$ sont INF;
- si $x \in V$ et G, H sont INF, alors $ite(x, G, H)$ est INF.

Partition de Shannon

Soit F une formule et x un prédicat. On a:

$$F \equiv ite(x, F[x/1], F[x/0])$$

Preuve: étudier les affectation des deux côtés

Corollaire: Toute formule est équivalente à une formule INF.
(Preuve: appliquer l'équivalence ci-dessus à toute variable et toute sous-formule.

Construction d'un BDD

Soit F une formule. Nous allons construire un BDD représentant F pour un ordre $<$ donné:

SI F ne contient aucune variable, alors soit $F \equiv \mathbf{0}$ soit $F \equiv \mathbf{1}$, et le BDD ne contient que la feuille correspondante.

Sinon, soit x la variable la plus petite (dans $<$) qui apparaît dans F. Construire des BDD B_0 et B_1 pour $F[x/1]$ et $F[x/0]$.

La partition de Shannon implique que F est représentable par un graphe avec une racine x et deux-sous-arbres B_1 et B_0. Pour obtenir un BDD, on teste si B_0 et B_1 sont égaux, dans ce cas on prend simplement B_0. Sinon, on fusionne tous les sous-graphes isomorphes dans B_0 et B_1.

Exemple: Construction d'un BDD

On considère à nouveau la formule de l'Exemple 2:

$$F \equiv \Big(carry \leftrightarrow (x_1 \wedge x_2)\Big) \wedge \Big(sum \leftrightarrow (x_1 \vee x_2) \wedge \neg carry\Big)$$

On a, p.ex.:

$F[x_1/0] \equiv \neg carry \wedge (sum \leftrightarrow x_2)$

$F[x_1/1] \equiv (carry \leftrightarrow x_2) \wedge (sum \leftrightarrow \neg carry)$

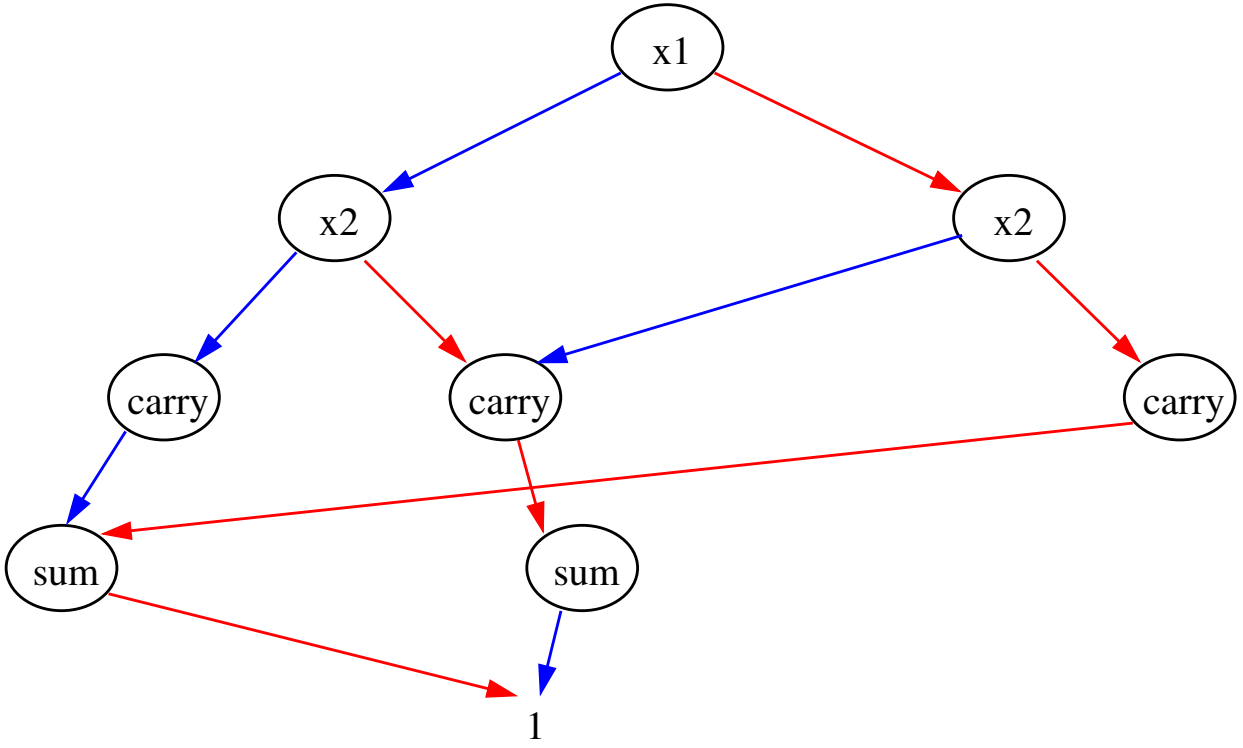
$F[x_1/0][x_2/0] \equiv \neg carry \wedge \neg sum$

$F[x_1/0][x_2/1] \equiv F[x_1/1][x_2/0] \equiv \neg carry \wedge sum$

$F[x_1/1][x_2/1] \equiv carry \wedge sum$

Exemple: Construction d'un BDD

En appliquant la procédure de construction, on obtient le même BDD qu'avant:



Canonicité des BDD

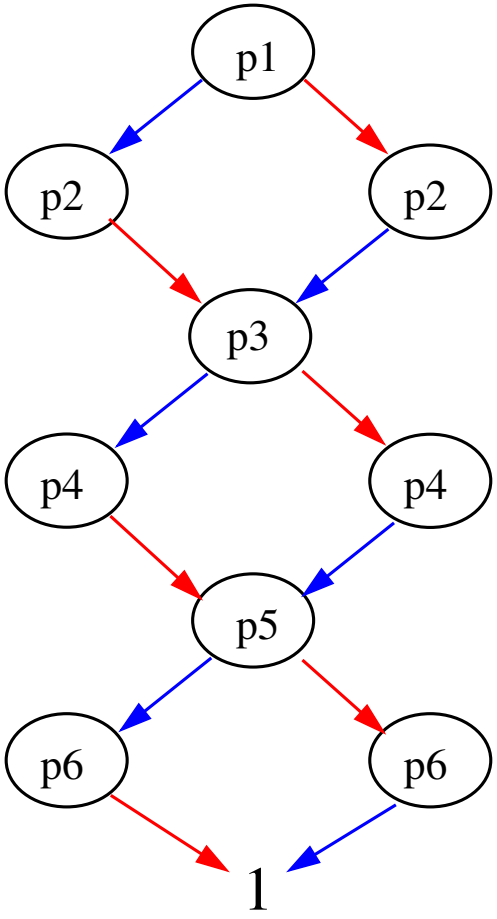
Remarque: Le résultat de la construction précédente est unique (sauf isomorphisme).

Par conséquent, étant donné F et $<$, il existe exactement un BDD sur $<$ qui représente F.

Remarque: Changer $<$ donne un autre BDD, parfois avec une différence de taille énorme!

Exemple: Ordre de variables

Voici un BDD respectant l'ordre $p_1 < p_2 < p_3 < p_4 < p_5 < p_6$:



Lorsqu'on généralise cette formule de 6 à $2n$ variables, la taille du BDD sera linéaire dans n.

Un BDD de taille m peut donc représenter un ensemble de 2^m affectations (même plus, si certains variables sont redondantes).

Par contre, si on considère l'ordre ci-dessous, la taille du BDD devient exponentielle !

$$p_1 < p_3 < p_5 < p_2 < p_4 < p_6.$$

Test d'équivalence

Problème: Soient B et C des BDD (avec le même ordre). B et C représentent-ils la même formule ?

Solution: (naïve) Tester si B et C sont isomorphes.

Cas spécial:

Satisfaisabilité : Tester si la racine est $\mathbf{0}$.

Tautologie: Tester si la racine est $\mathbf{1}$.

Réalisation avec tables de hachage

Typiquement, on souhaite manipuler plusieurs BDD en même temps.

Une réalisation efficace de BDD va exploiter la canonicité des BDD: Tous les sommets de tous les BDD seront stockés dans un grand table de hachage.

Tout sommet peut être identifié avec le BDD dont il est la racine.
Du coup, tout BDD est représentable par un pointer vers sa racine.

Initialement, le table de hachage possède deux éléments, les feuilles 0 et 1.

Tout autre sommet est identifié par un triplet (x, B_1, B_0), où x est une variable et B_1, B_0 sont les pointers vers les racines des sous-arbres.

Typiquement, on réalise une fonction $mk(x, B_1, B_0)$ qui teste si un tel sommet existe déjà; si c'est le cas, on le renvoye, sinon on crée un nouveau sommet.

Du coup, un ensemble de BDD est alors stocké sous forme de "forêt" avec plusieurs pointers sur les racines qui nous intéressent.

Problème: ramasse-miettes (avec compteurs de référence)

Test d'équivalence (revisé)

Si B et C sont stockés dans un table de hachage, alors ils sont représentables par des pointeurs vers leurs racines.

On teste donc simplement si ces pointeurs sont égaux, en temps constant.

Opérations logiques I : Complément

Soit F une formule et B un BDD pour F.

Problème: Calculer un BDD pour $\neg F$.

Solution: Repliquer tout sommet de B, en échangeant les deux feuilles.

Remarque: Il existe une amélioration des BDD qui réalisent la négation en temps constant.

Opérations logiques II: Intersection

Soient F, G des formules et B, C leurs BDD.

Problème: Calculer un BDD pour $F \wedge G$ à partir de B et C.

On utilise l'équivalence suivante:

$$F \wedge G \equiv ite(x, (F \wedge G)[x/1], (F \wedge G)[x/0]) \equiv ite(x, F[x/1] \wedge G[x/1], F[x/0] \wedge G[x/0])$$

Si x est la plus petite variable apparaissant dans F et G, alors $F[x/1], F[x/0], G[x/1], G[x/0]$ sont des sommets de B et C (les racines ou leurs fils).

On construit alors un BDD pour $F \wedge G$ avec la stratégie recursive suivante :

Si B et C sont égaux, renvoyer B.

Si soit B ou C sont $\mathbf{0}$, renvoyer $\mathbf{0}$.

Si soit B ou C sont $\mathbf{1}$, renvoyer l'autre.

Sinon, soit x la plus petite variables qui étiquete les racines de B et C.

Si la racine de B est étiqueté par x, soit B_1, B_0 les fils de B; sinon soit $B_1, B_0 := B$. (analogue pour C_1, C_0.

Appliquer cette stratégie récursivement à B_1, C_1 et B_0, C_0, ce qui donne des BDD E et F. Si $E = F$, renvoyer E, sinon $mk(x, E, F)$.

Opérations logiques III: Union

Soient F, G des formules et B, C des BDD correspondants.

Problème: Calculer un BDD pour $F \vee G$ à partir de B et C.

Solution: Similaire à l'intersection, avec les règles de base pour $\mathbf{1}$ et $\mathbf{0}$ adaptées.

Complexité: Avec programmation dynamique: $\mathcal{O}(|B| \cdot |C|)$ (toute paire de sommets une fois).

Calcul de prédéceseurs

Il nous manque encore une stratégie pour calculer

$$pre(M) = \{\, s \mid \exists s' : (s, s') \in \rightarrow \ \wedge \ s' \in M \,\}.$$

On note qu'une relation $\rightarrow$ est un sous-ensemble de $S \times S$ tandis que $M \subset S$.

On représente donc M par un BDD avec variables $y_1, \ldots, y_m$.

$\rightarrow$ sera représenté par un BDD avec $x_1, \ldots, x_m$ et $y_1, \ldots, y_m$
(l'état "avant" et "après").

Remarque: Tout BDD pour M est également un BDD représentant $S \times M$!

Du coup, on réécrit $pre(M)$ comme suit:

$$\{ s \mid \exists s' : (s, s') \in \rightarrow \ \cap \ (S \times M) \}$$

pre se réduit alors aux problèmes d'intersection et abstraction existentielle.

Abstraction existentielle

L'abstraction existentielle de x sur F est définie comme suit :

$$\exists x : F \quad \equiv \quad F[x/0] \vee F[x/1]$$

Donc $\exists x :\ F$ est vraie pour les affectation qui pourront être étendue avec une valeur pour x qui rend F vraie.

Exemple: Soit $F \equiv (x_1 \wedge x_2) \vee x_3$. Alors

$$\exists x_1 : F \equiv F[x_1/0] \vee F[x_1/1] \equiv (x_3) \vee (x_2 \vee x_3) \equiv x_2 \vee x_3$$

Par extension, on peut définir l'abstraction existentielle d'un ensemble de variables (une par une).

Partie 9: Abstraction

Exemple 1 (boucle)

On considère le programme suivant avec trois variables x, y, z.

ℓ_1: `y = x+1;`

ℓ_2: `z = 0;`

ℓ_3: `while (z < 100) z = z+1;`

ℓ_4: `if (y < x) error;`

Question: L'erreur, est-elle atteignable ?

Exemple 2 (Trier)

Un autre programme avec trois variables x, y, z.

```
ℓ1: if x > y then  swap x,y  else  skip;

ℓ2: if y > z then  swap y,z  else  skip;

ℓ3: if x > y then  swap x,y  else  skip;

ℓ4: skip
```

Hypothèse: soient x, y, z toutes différents au départ

Question: Est-ce qu'on trie x, y, z en ordre ascendant ?

Question 3 (pilote)

Code en C d'un pilote pour Windows

Opérations sur un sémaphor: acquérir (lock), relâcher (release)

Ces opérations doivent être utilisées tour à tour !

- ne jamais acquérir le lock deux fois de suite ;
- ne jamais relâcher le lock deux fois de suite.

Abstraction

On souhaite vérifier des systèmes avec nombre infini d'états,
ou nombre états fini mais trop important.

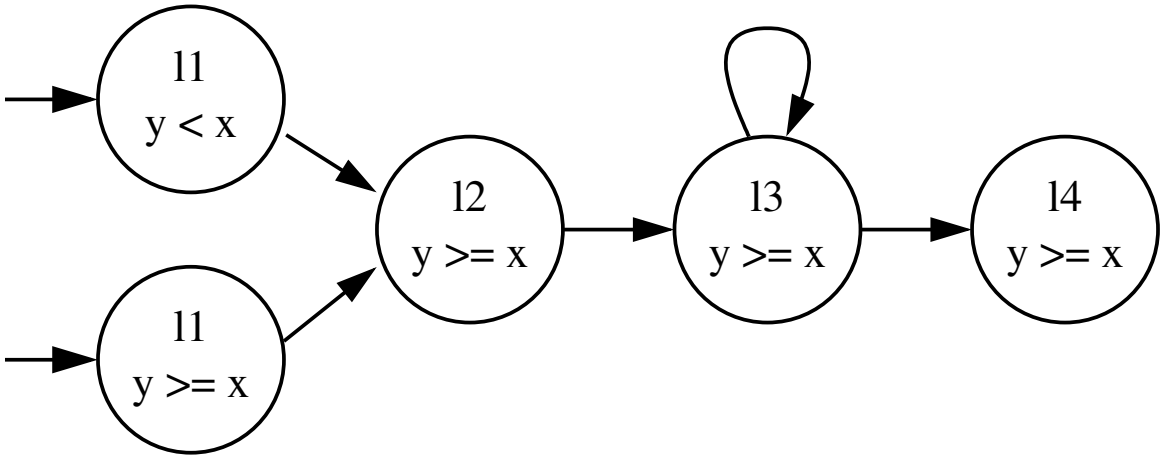
Idée: supprimer de l'information "non importante",
vérifier un système plus petit mais "equivalent".

Perspective différente : fusionner des états "équivalents"

Exemple 1

Oublier les valeurs concrètes de `x,y,z`; on ne retient que le compteur de programme et le prédicat $y < x$

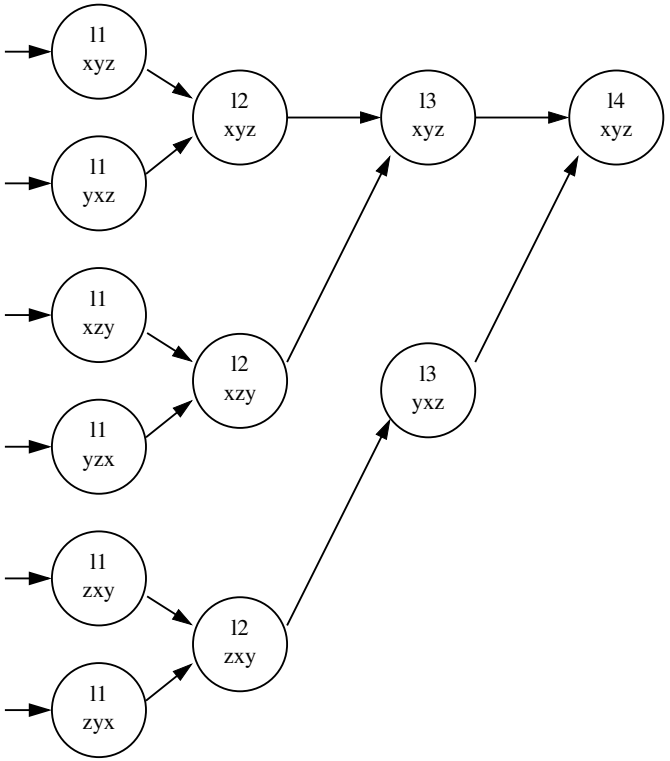
Structure (abstraite) résultante :



Résultat: ℓ_4 est atteignable seulement avec $y \geq x$; l'erreur n'arrive pas.

Exemple 2

Oublier les valeurs de `x,y,z`; ne retenir que le compteur de programme est une permutation de of x, y, z



Résultat: seul *xyz* atteint ℓ_4.

Simulation

Soient $\mathcal{K}_1 = (S, \rightarrow_1, s_0, AP, \nu)$ et $\mathcal{K}_2 = (T, \rightarrow_2, t_0, AP, \mu)$ deux SK (peut-être infinis), et soit $H \subseteq S \times T$ une relation.

H est une simulation de $\mathcal{K}_1$ à $\mathcal{K}_2$ si

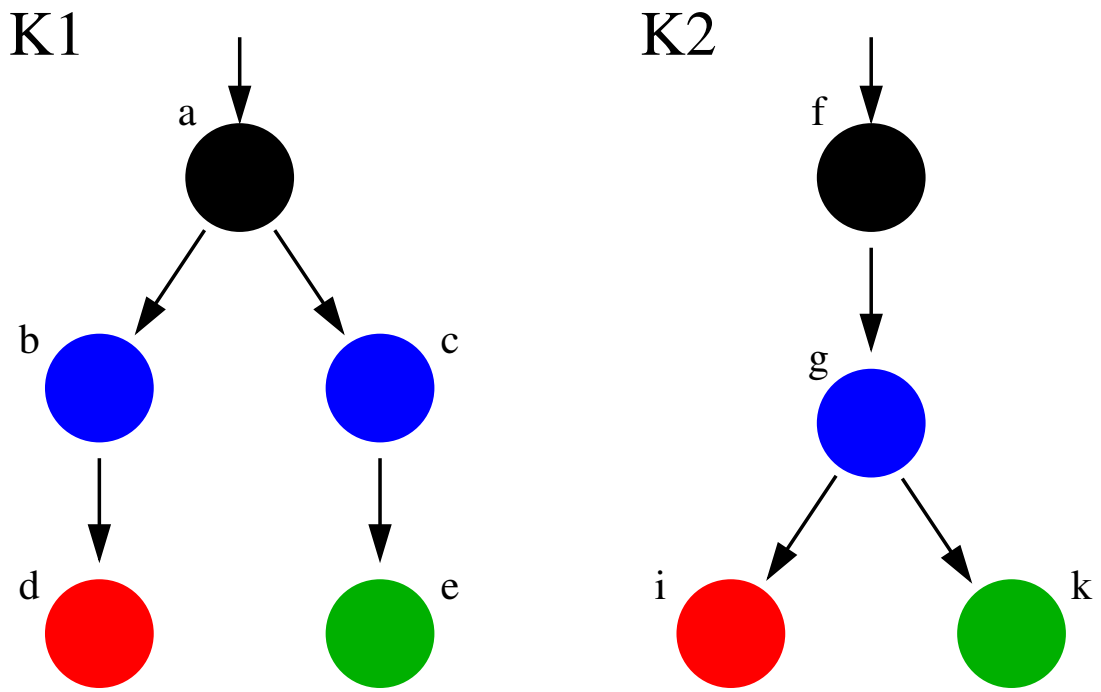
(i) $(s_0, t_0) \in H$;

(ii) pour tout $(s, t) \in H$ on a $\nu(s) = \mu(t)$;

(iii) si $(s, t) \in H$ et $s \rightarrow_1 s'$, alors il existe t' tel que $t \rightarrow_2 t'$ et $(s', t') \in H$.

On dit que $\mathcal{K}_2$ simule $\mathcal{K}_1$ (noté $\mathcal{K}_1 \leq \mathcal{K}_2$) s'il existe une telle relation H.

Intuition: $\mathcal{K}_2$ peut faire tout ce qui est possible dans $\mathcal{K}_1$.



$\mathcal{K}_2$ simule $\mathcal{K}_1$ (avec $H = \{(a, f), (b, g), (c, g), (d, i), (e, k)\}$).

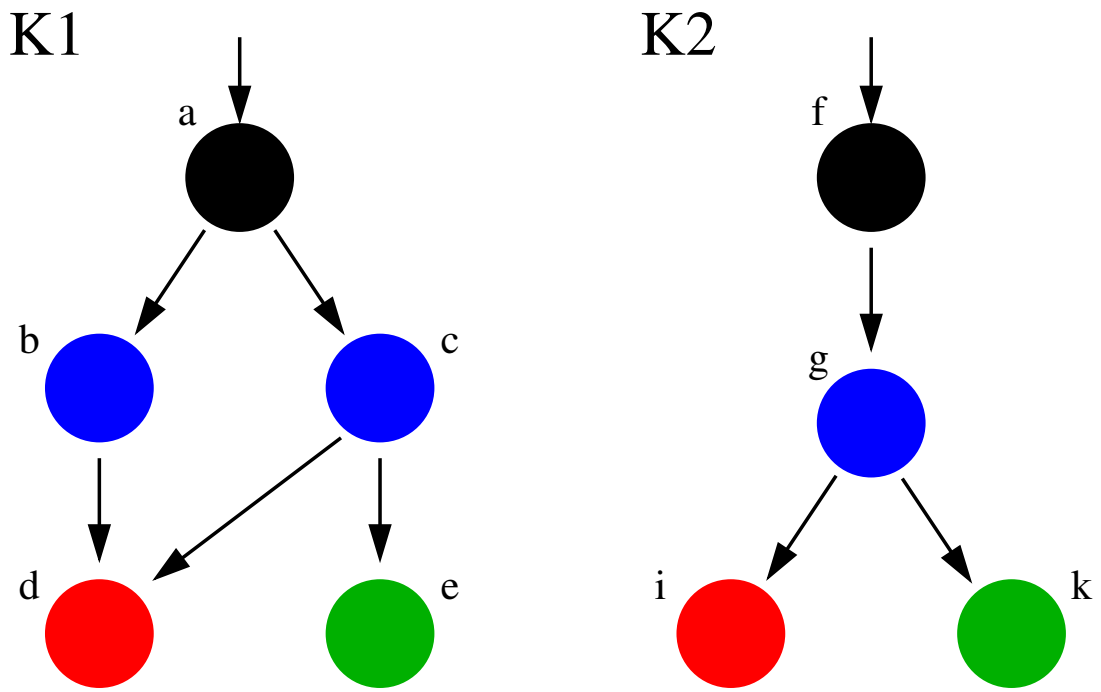
Par contre, $\mathcal{K}_1$ ne simule pas $\mathcal{K}_2$.

Bisimulation

H s'appelle bisimulation entre $\mathcal{K}_1$ et $\mathcal{K}_2$ si H est une simulation de $\mathcal{K}_1$ à $\mathcal{K}_2$ et $\{ (t, s) \mid (s, t) \in H \}$ est une simulation de $\mathcal{K}_2$ à $\mathcal{K}_1$.

On appelle $\mathcal{K}_1$ et $\mathcal{K}_2$ bisimilaires (noté $\mathcal{K}_1 \equiv \mathcal{K}_2$) si une telle relation H existe.

Attention: En général, $\mathcal{K}_1 \leq \mathcal{K}_2$ et $\mathcal{K}_2 \leq \mathcal{K}_1$ n'impliquent pas $\mathcal{K}_1 \equiv \mathcal{K}_2$!



(Bi-)Simulation et model checking

Soit $\mathcal{K}_1 \leq \mathcal{K}_2$ et ϕ une formule de LTL.

Alors $\mathcal{K}_2 \models \phi$ implique $\mathcal{K}_1 \models \phi$.

Soit $\mathcal{K}_1 \equiv \mathcal{K}_2$ et ϕ une formule de CTL.

Alors $\mathcal{K}_1 \models \phi$ ssi $\mathcal{K}_2 \models \phi$.

Si $\mathcal{K}_1 \not\equiv \mathcal{K}_2$, alors il existe une formule ϕ de CTL avec $\mathbf{EX}$ comme seule modalité, telle que $\phi \models \mathcal{K}_1$ et $\phi \not\models \mathcal{K}_2$.

Abstraction existentielle

Soit $\mathcal{K} = (S, \to, r, AP, \nu)$ une SK (structure concrète).

Soit $\approx$ une relation d'équivalence sur S telle que pour tout $s \approx t$ on a $\nu(s) = \nu(t)$ (on dit que $\approx$ respecte ν).

On note $[s] := \{ t \mid s \approx t \}$ la classe d'équivalence de s;
$[S]$ note l'ensemble de ces classes.

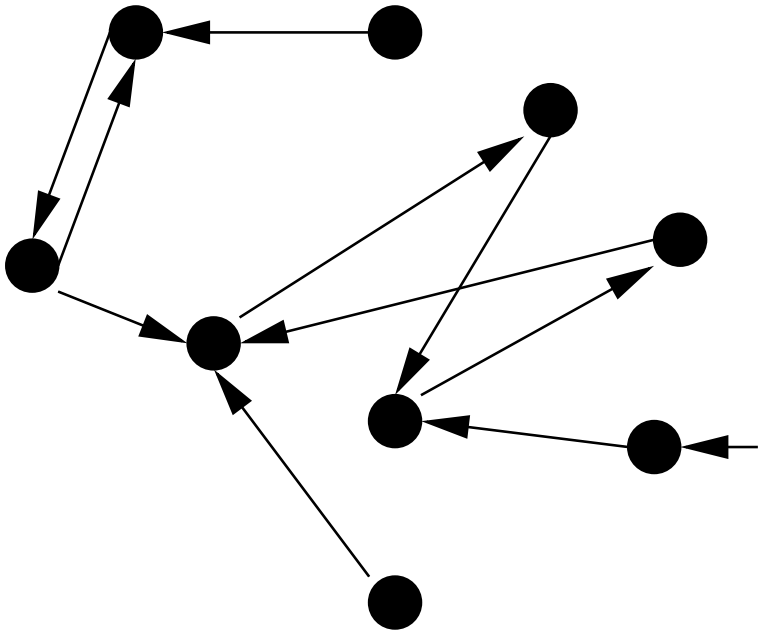
L'abstraction de S denote alors la structure $\mathcal{K}' = ([S], \to', [r], AP, \nu')$ où

$[s] \to' [t]$ pour tout $s \to t$;

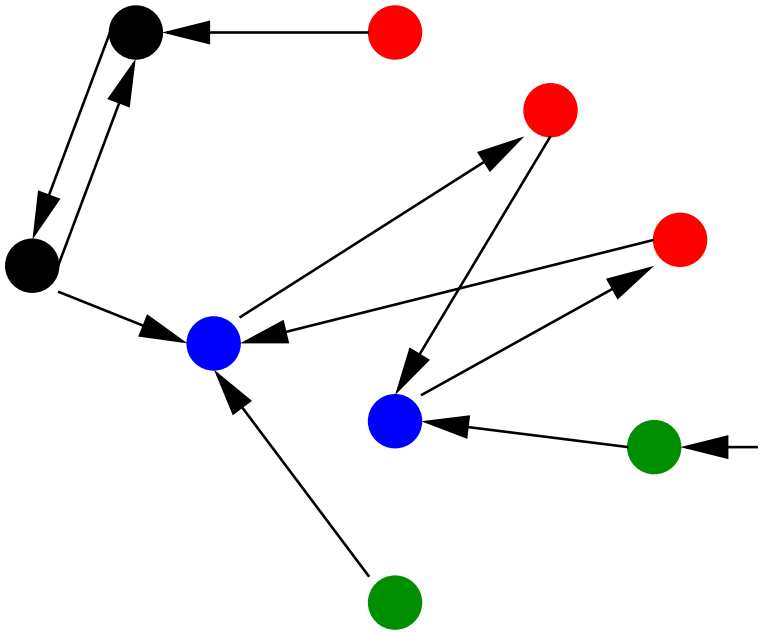
$\nu'([s]) = \nu(s)$ (bien défini !)

Exemple

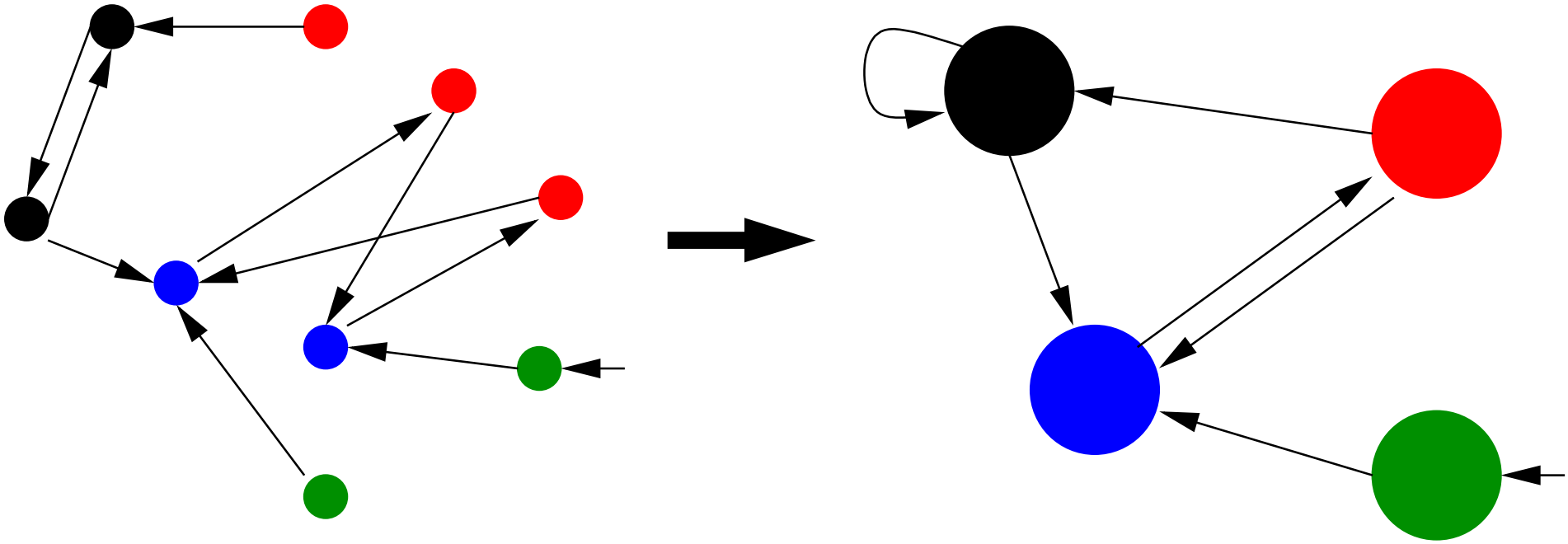
On considère la structure suivante :



On partitionne les états en classes :



Structure abstraite obtenue en fusionnant les états :



Soit $\mathcal{K}'$ une structure obtenue par abstraction existentielle depuis $\mathcal{K}$.

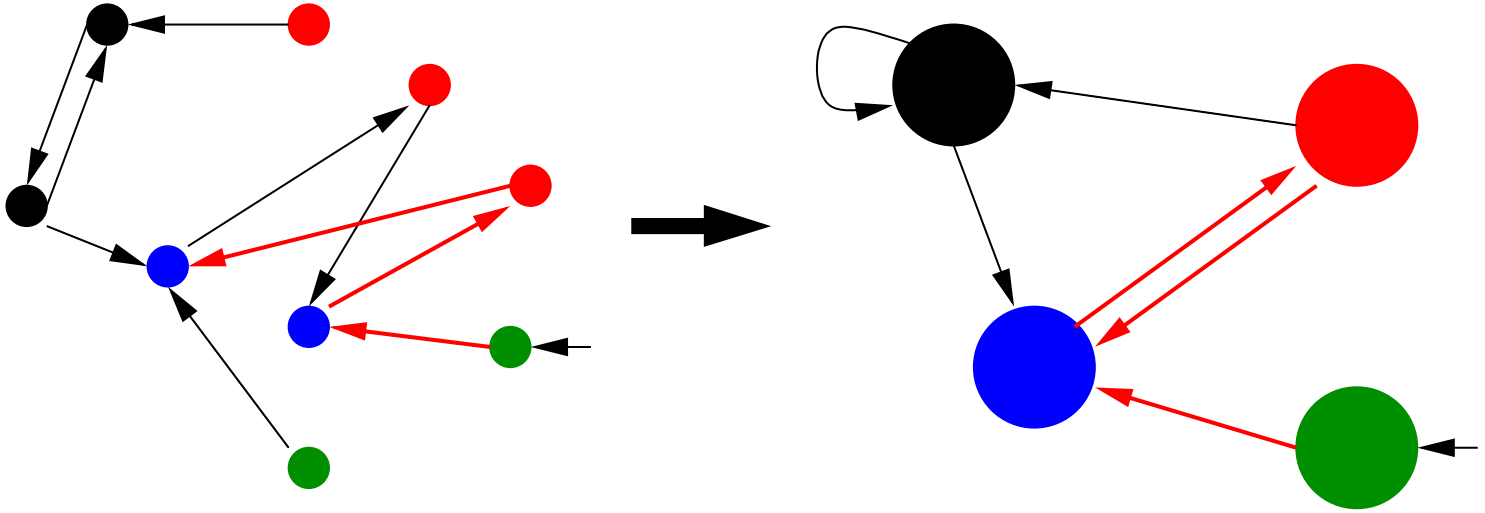
Alors on a $\mathcal{K} \leq \mathcal{K}'$.

Du coup, si $\mathcal{K}'$ satisfait une formule de LTL, $\mathcal{K}$ le fait aussi.

Par contre, le sens invers n'est pas vrai en général.

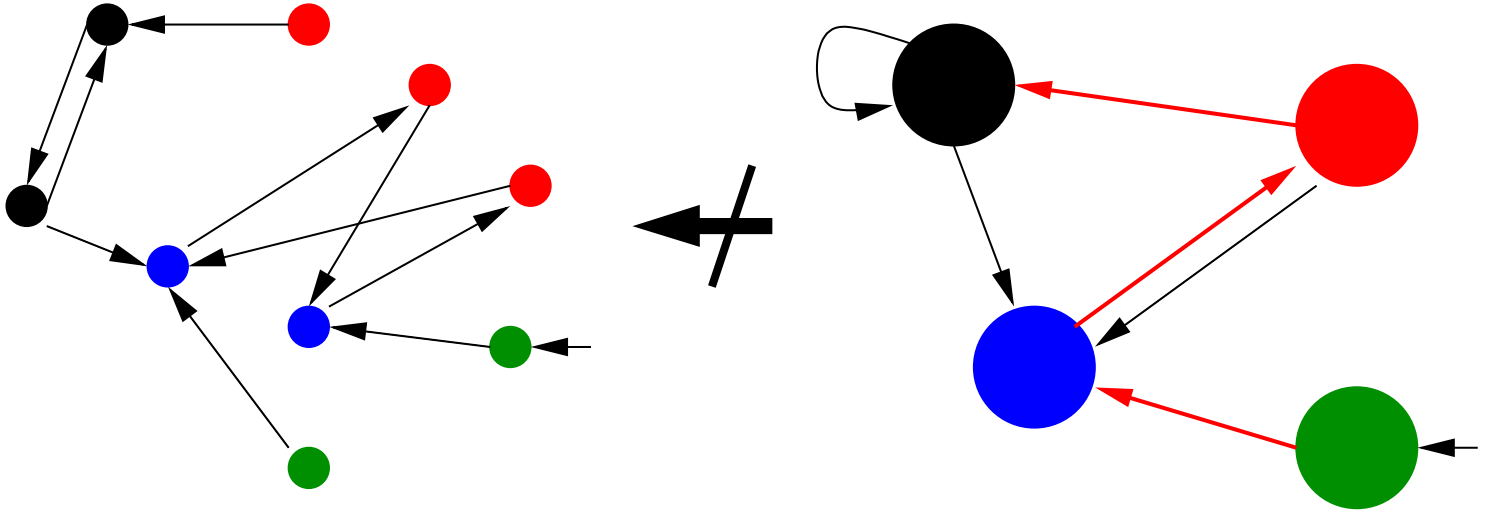
L’abstraction peut introduire des chemins supplémentaires :

Tout calcul concret a une correspondence dans la structure abstraite . . .



L'abstraction peut introduire des chemins supplémentaires :

. . . mais certains calculs abstraits n'ont pas de correspondances concrètes.



Supposons que $\mathcal{K}' \not\models \phi$, où ρ est un contre-exemple, un calcul ne satisfaisant pas ϕ.

On vérifie s'il existe un calcul dans $\mathcal{K}$ qui correspond à ρ.

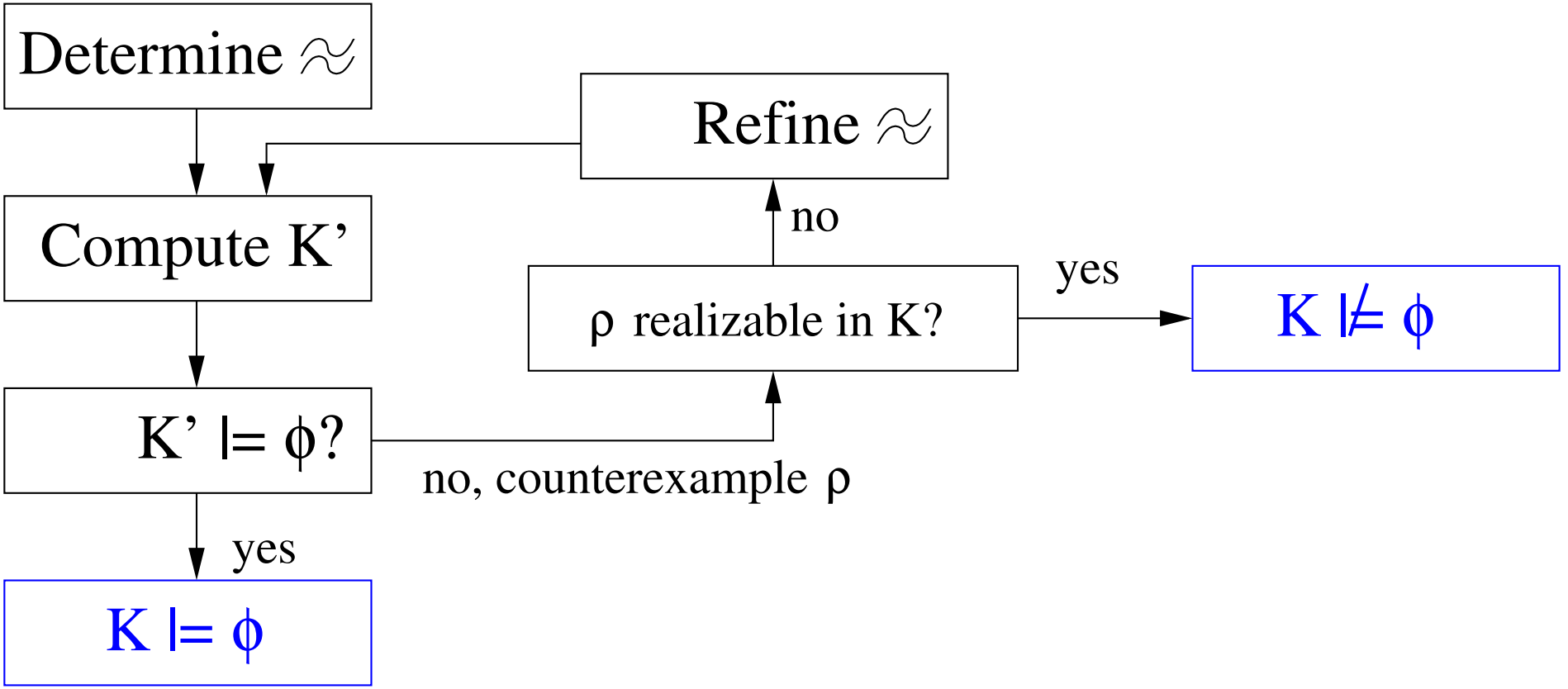
Si c'est le cas, alors $\mathcal{K} \not\models \phi$.

Sinon, en peut utiliser ρ pour raffiner l'abstraction, en supprimant quelques équivalences dans H pour rendre ρ impossible.

Ce raffinement peut être répété jusqu'à ce qu'on obtient une réponse définitive (positive ou négative) au problème $\mathcal{K} \models \phi$. Ce technique s'appelle (en anglais) counterexample-guided abstraction refinement (CEGAR) [Clarke et al, 1994].

Le cycle d'abstraction et raffinement

Données: $\mathcal{K}, \phi$



Simulation de ρ

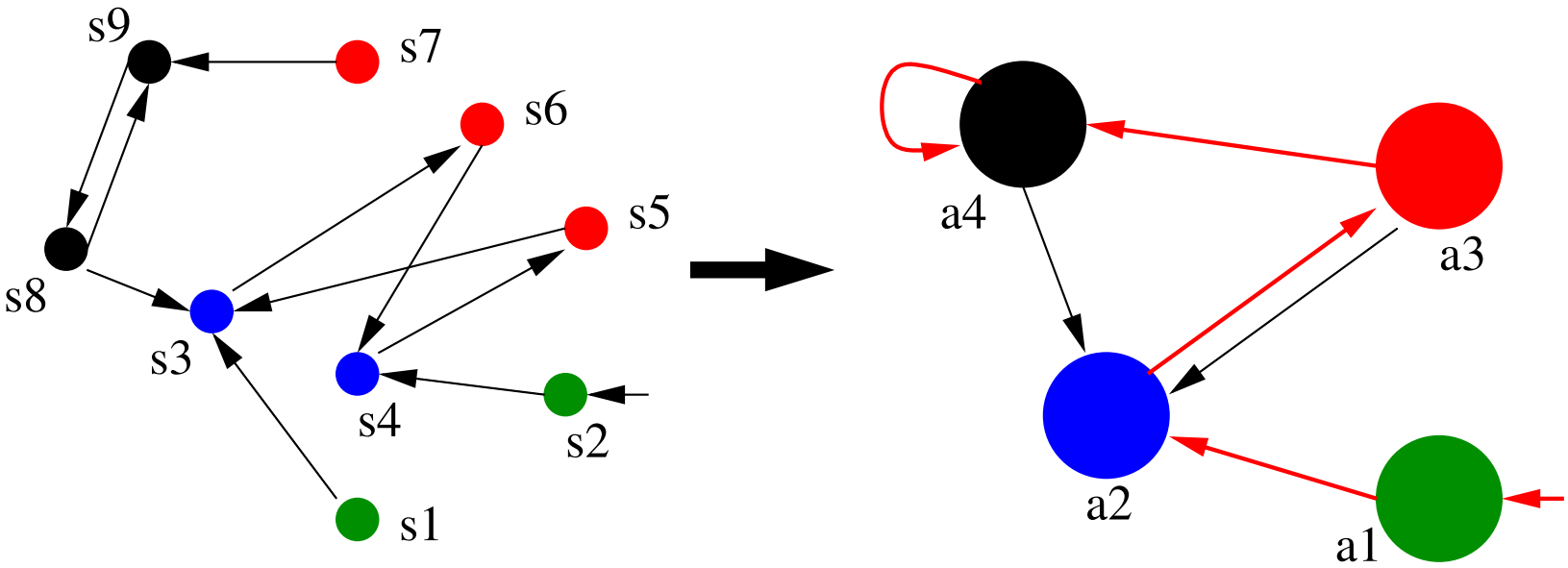
Problème: Étant donné un contre-exemple ρ, existe-il un calcul correspondant à ρ dans $\mathcal{K}$?

Solution: “Simuler” ρ sur $\mathcal{K}$.

Remarque: Tout contre-exemple ρ peut être partitionné en une partie initiale et une boucle: $\rho = w_S w_L^\omega$ pour quelques w_S, w_L.

Distinction de cas: La simulation peut échouer soit dans w_S soit dans w_L.

Exemple 1: $\mathrm{G}\,\neg$*black*



L'abstraction donne un contre-exemple avec $w_S = a_1 a_2 a_3 a_4$ et $w_S = a_4$.

Simuler w_S

Soit $w_S = b_0 \cdots b_k$.

On démarre avec $S_0 = \{r\}$ car $b_0 = [r]$.

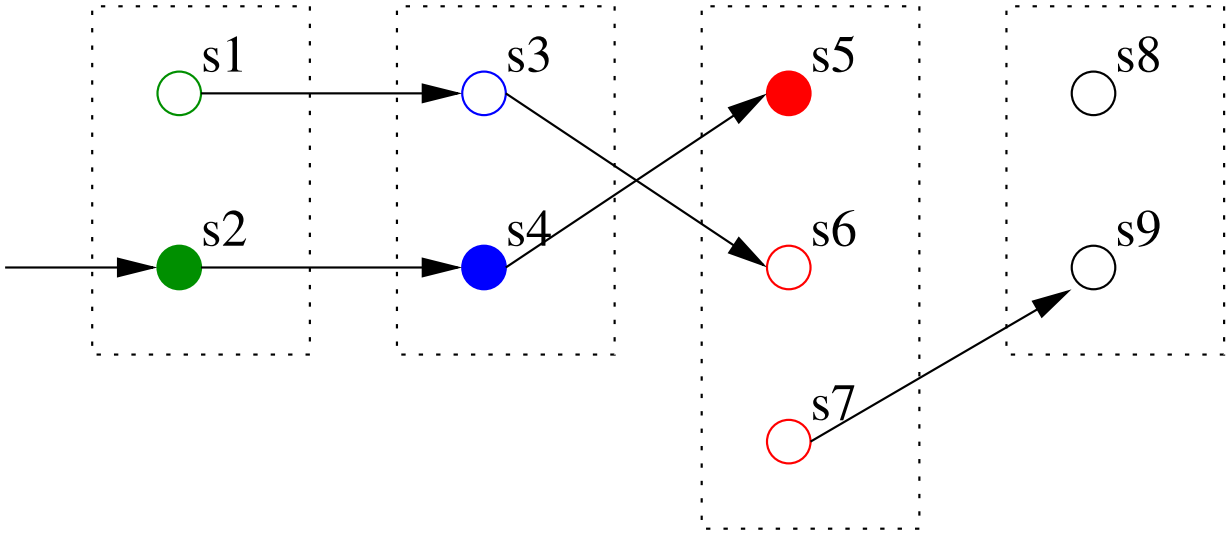
Pour $i = 1, \ldots, k$, calculer $S_i = \{\, t \mid t \in b_i \wedge \exists s \in S_{i-1} : s \to t \,\}$.

Si $S_k \neq \emptyset$, alors il existe un contre-exemple concret pour w_S.

Si $S_k = \emptyset$: Trouver l'index minimal ℓ avec $S_\ell = \emptyset$: Le raffinement devrait distinguer entre les états dans $S_{\ell-1}$ et dans ceux dans $b_{\ell-1}$ qui ont des prédécesseurs dans b_ℓ.

Exemple: $w_S = a_1 a_2 a_3 a_4$

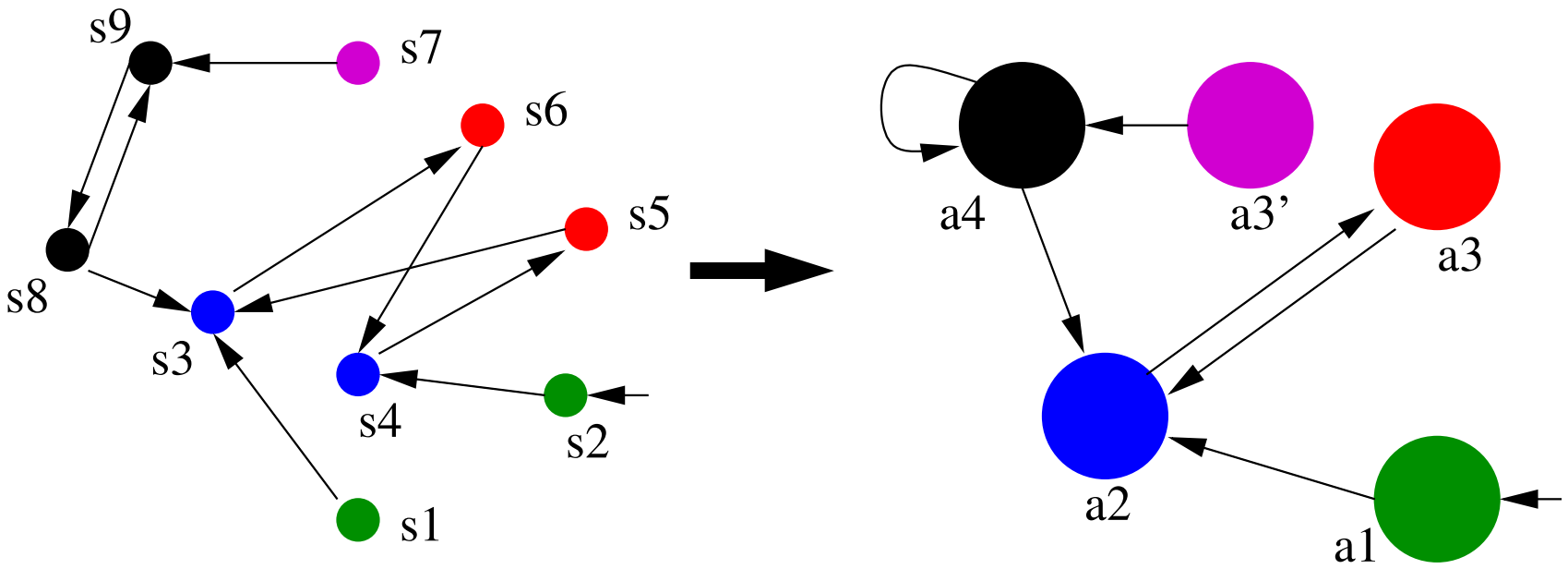
$S_0 = \{s_2\}$, $S_1 = \{s_4\}$, $S_2 = \{s_5\}$, $S_3 = \emptyset$.



Au prochain tour, on doit distinguer s_5 et s_7.

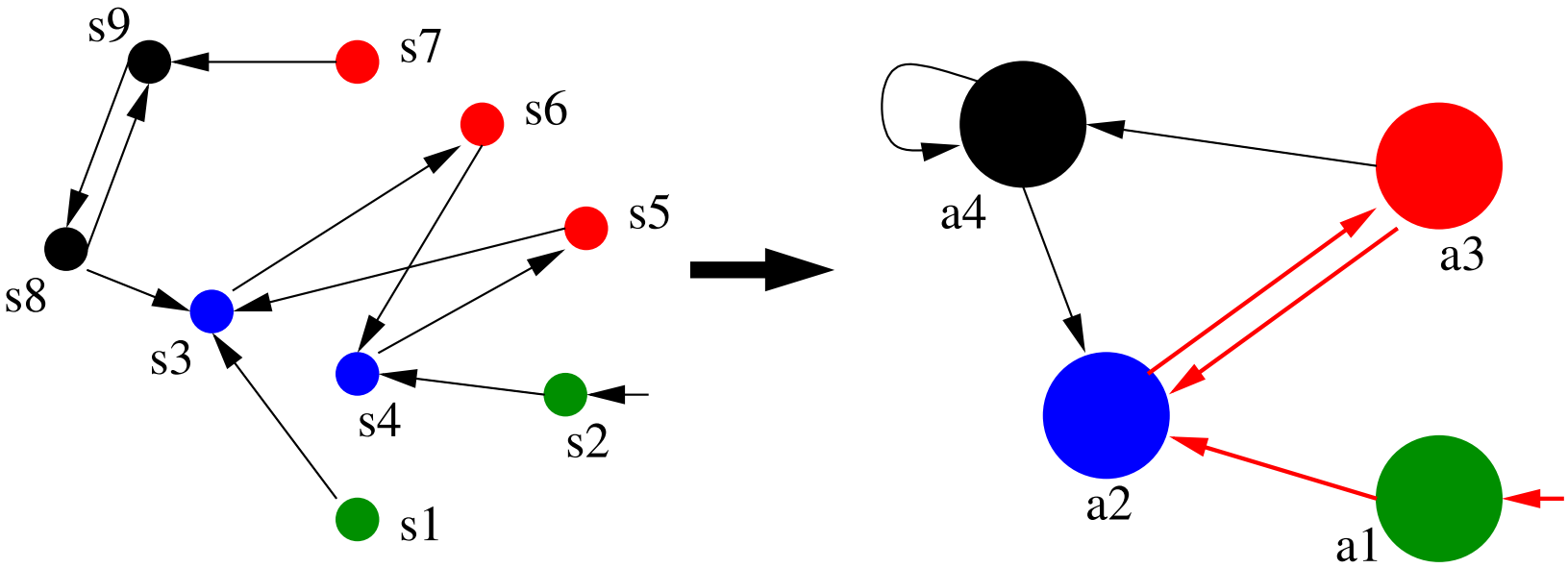
Solutions possibles $\{s_5, s_6\}, \{s_7\}$ ou $\{s_5\}, \{s_6, s_7\}$.

Nouvelle tentative: $\mathrm{G}\,\neg$*black* avec raffinement



Dans la nouvelle abstraction, il n'y a pas de contre-exemple. Du coup $\mathrm{G}\,\neg$*black* est vrai aussi dans la structure d'origine.

Exemple 2: F G *red*



L'abstraction donne un contre-exemple avec $w_S = a_1 a_2$ et $w_L = a_3 a_2$.

Simuler une boucle

Soient $w_S = b_0 \cdots b_k$, $\quad w_L = c_1 \cdots c_\ell$

On simule w_S comme avant, mais la simulation de w_L est répétée plusieurs fois.

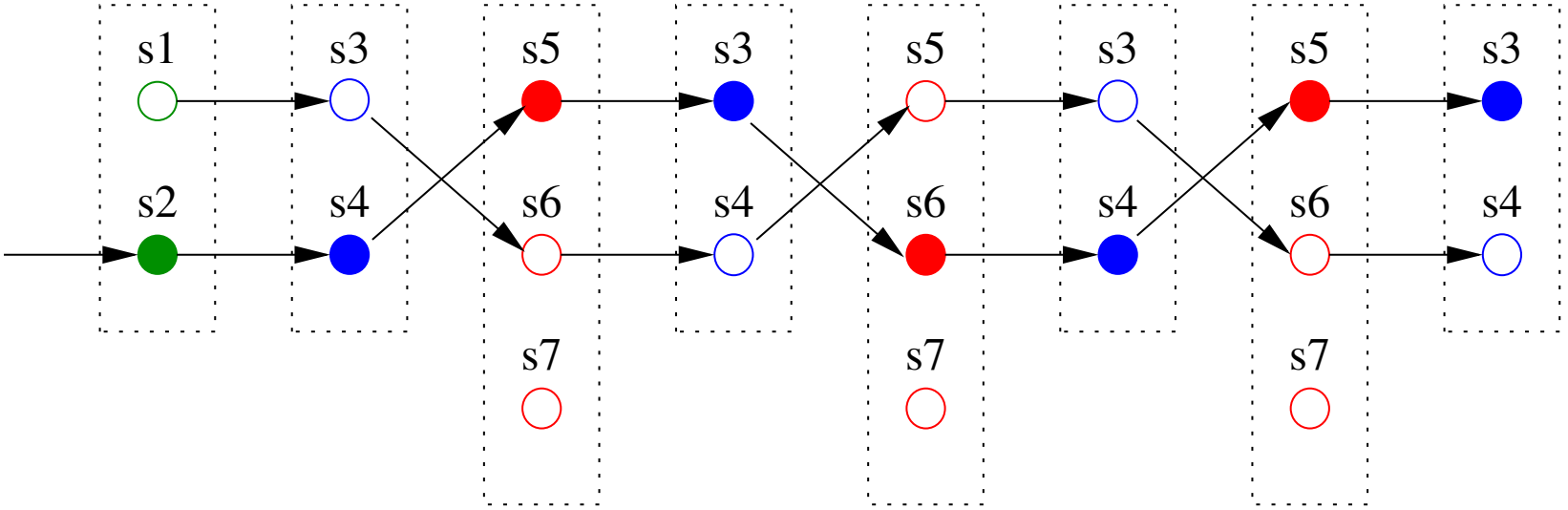
Soit m la taille minimale d'une classe d'équivalence dans w_L :

$$m = \min_{i=1,\ldots,\ell} |c_i|$$

Alors on simule $w_S w_L^{m+1}$; soit la simulation échoue, soit on obtient un contre-exemple.

Raffinement: comme avant

Exemple: $w_S = a_1 a_2$, $w_L = a_3 a_2$, $m = 2$



La simulation réussit car il y a une boucle autour de s_4.

Alors il existe un contre-exemple réel, du coup $\mathcal{K} \not\models \phi$.