Concepts et Model Checking

Stefan Schwoon

ENSIIE, Année 2025/2026

Organisation

Cours: lundi de 14h à 15h45

TD/TP: lundi de 16h à 17h45

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

Durée: 6 semaines (du 12 janvier au 2 mars)
pas de cours les 19 et 26 janvier

Évaluation: Examen (le 2 mars, au lieu du TD)

Connaissances préalables utiles :

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” ?

Notion de logique :

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

Ici : approche logique pour vérifier la correction d’un système.

Logique temporelle : extension de la logique du premier ordre

Rappels: Calcul/Logique propositionnel(le)

Formules avec prédicats :

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

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

et 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 ?

Évaluée en fonction d’une affectation des prédicats.

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

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 modèle 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.

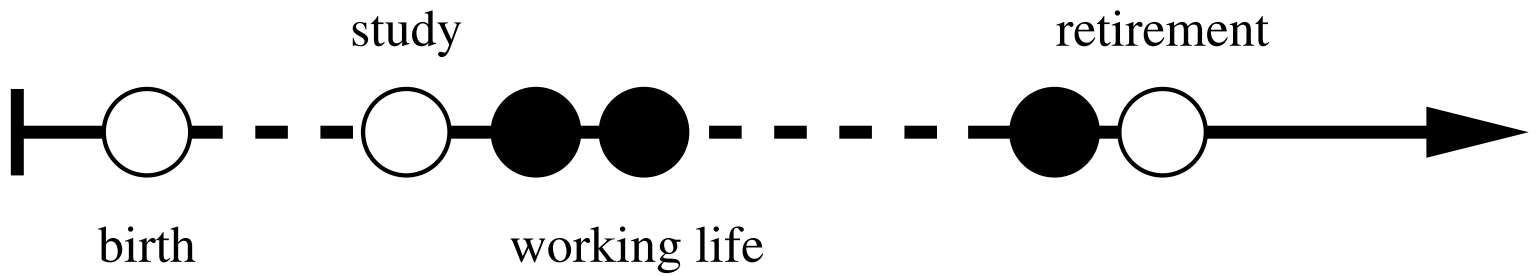
Exemples: Soient $\mathcal{B}_1(A) = 1$ et $\mathcal{B}_1(B) = 0$. Alors $\mathcal{B}_1 \not\models A \wedge B$ et $\mathcal{B}_1 \models \neg B$.

Soient $\mathcal{B}_2(A) = 1$ 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 considère des prédicats qui changent 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

évaluées par rapport aux *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 futurs potentiels.

Le rapport avec la vérification

Espace d'états d'un programme:

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

Prédicats, p.ex.

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

L'ensemble de ces prédicats peut être évalué dans un état du programme.

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 produisent un arbre d'affectations.

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

Question : cet arbre-là, satisfait-il la formule ?

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

Motivation

Les ordinateurs pénètrent de plus en plus nos vies quotidiennes :

- ordi personnel, smartphone, GPS, ...
- systèmes embarqués (voiture, avion, . . .)
- banque en ligne

Les erreurs informatiques ont un impact important économique, ou bien coûtent même des vies.

Coût estimé des systèmes informatiques fautifs aux États-Unis : 60 bn de dollars par an

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 : Cette erreur peut être trouvé par des tests 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 $2n$.

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

→ 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 d'erreurs.

$\rightarrow$ Pas du tout adapté aux effets non-déterministes (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 aspects de sécurité

La vérification considère toutes les exécutions d'un système

$\rightarrow$ on peut prouver l'absence d'erreurs
(mais 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 souhaitée.

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 souhaitée

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

Model checking

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

vérifient 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

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 coûteuses et nécessitent un effort.

$\Rightarrow$ utile dans les premières phases de conception

$\Rightarrow$ indispensable lorsque les erreurs sont très coûteuses ou même fatales (processeur, protocôles 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 qu'un système peut atteindre (ensemble fini ou infini)
$\rightarrow \subseteq S \times S$	$\widehat{=}$	transitions; décrivent les actions possibles
$r \in S$	$\widehat{=}$	état initial ("racine")

Exemple 1: Producteur/Consommateur

Pseudocode avec variables et concurrence:

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

P =

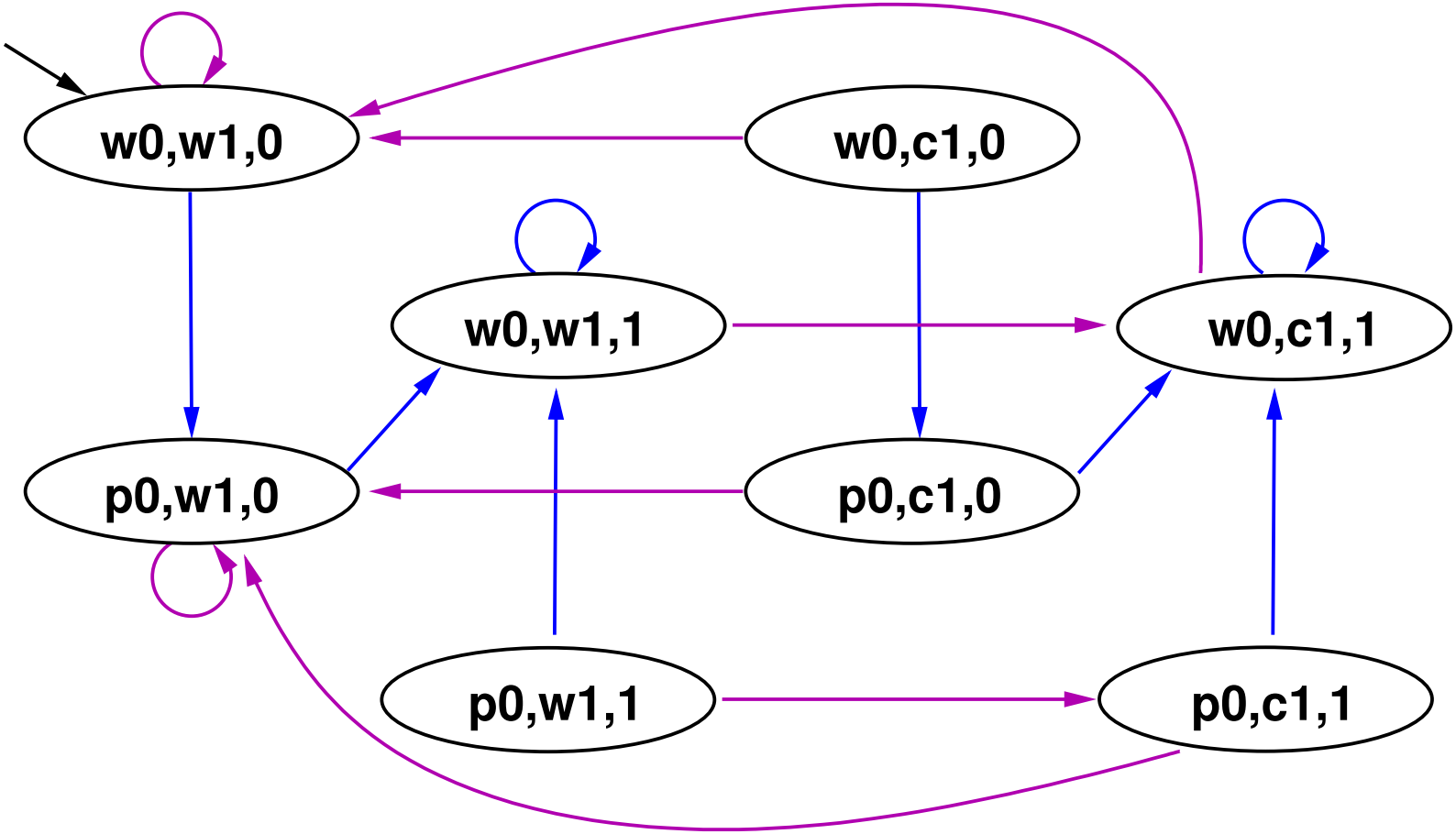
start;
while true do
  w_0: **wait** $(turn = 0)$;
  p_0: /* produce */
    $turn := 1$;
od;
end

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 $(w_0, w_1, 0)$



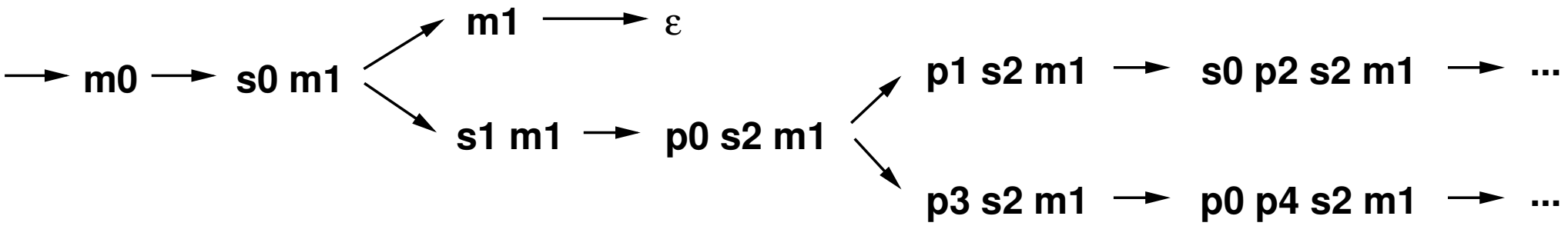
Exemple 2: Programme recursif

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

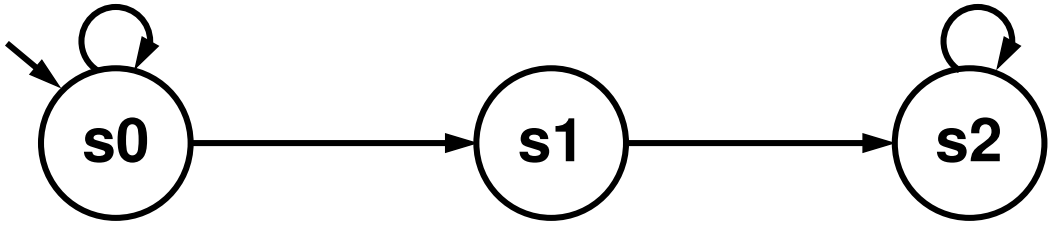
$\rho(i)$ dénote le i-ème élément de ρ et ρ^i le suffixe partant de $\rho(i)$.

$s \rightarrow^* t$ 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 rendent un ST potentiellement infini:

- Données: entiers, réels, listes, 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.

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:

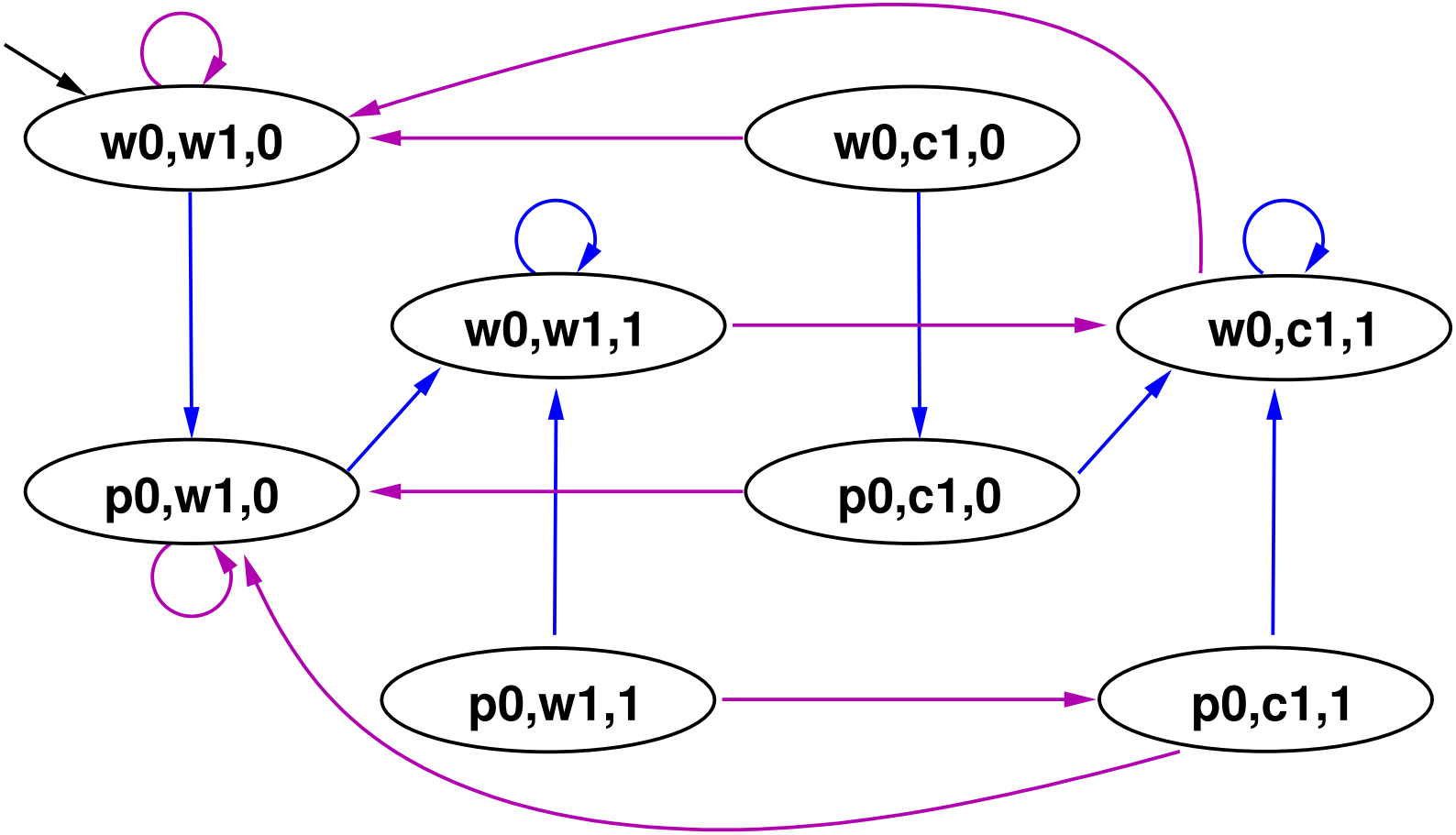
Soit $AP = \{prod, cons\}$;

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

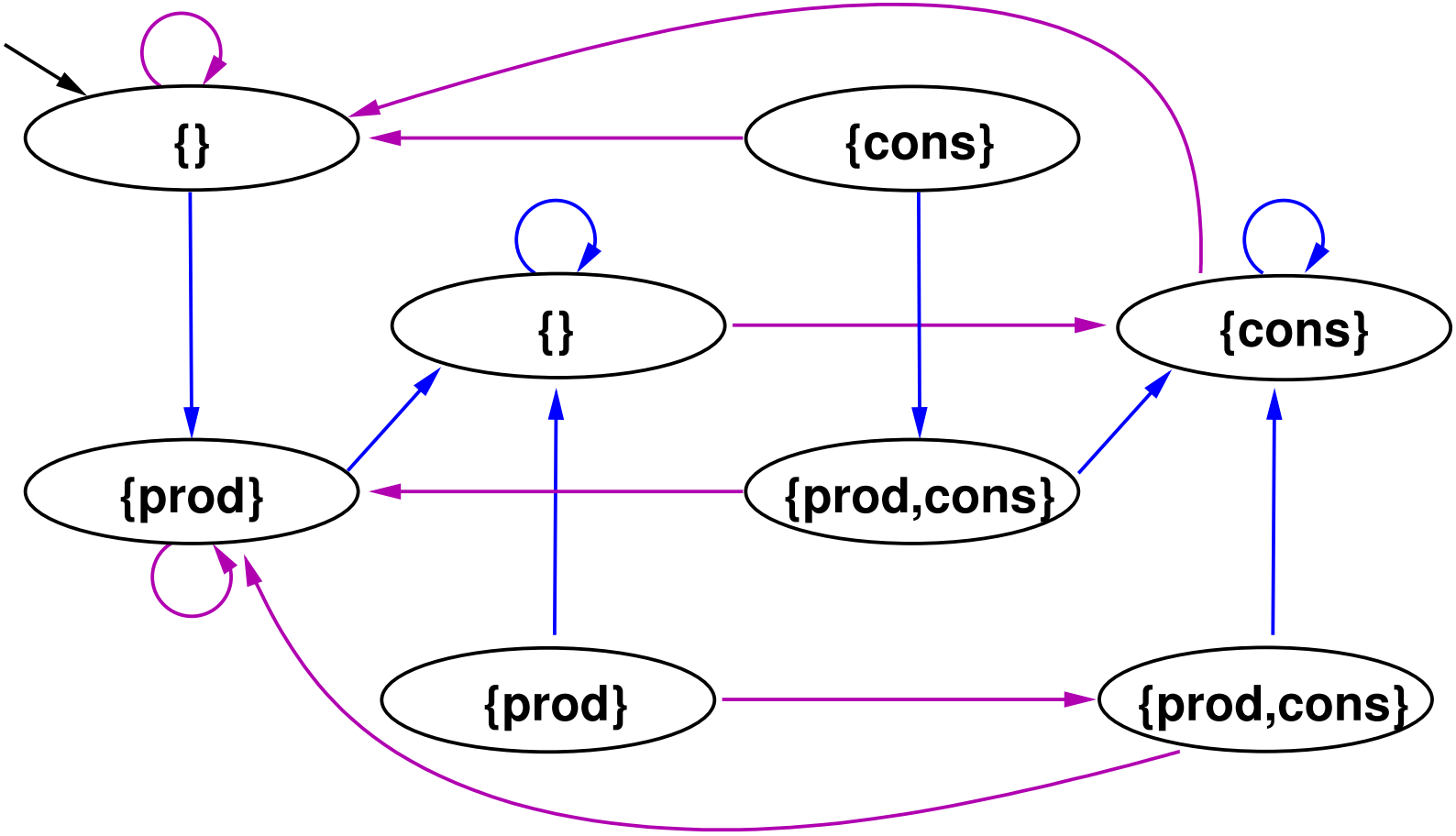
$\nu^{-1}(cons) = \{w_0, p_0\} \times \{c_1\} \times \{0, 1\}$.

Rappel: Exemple 1

Dans l'Exemple 1, . . .



. . . les affectations sont ainsi :

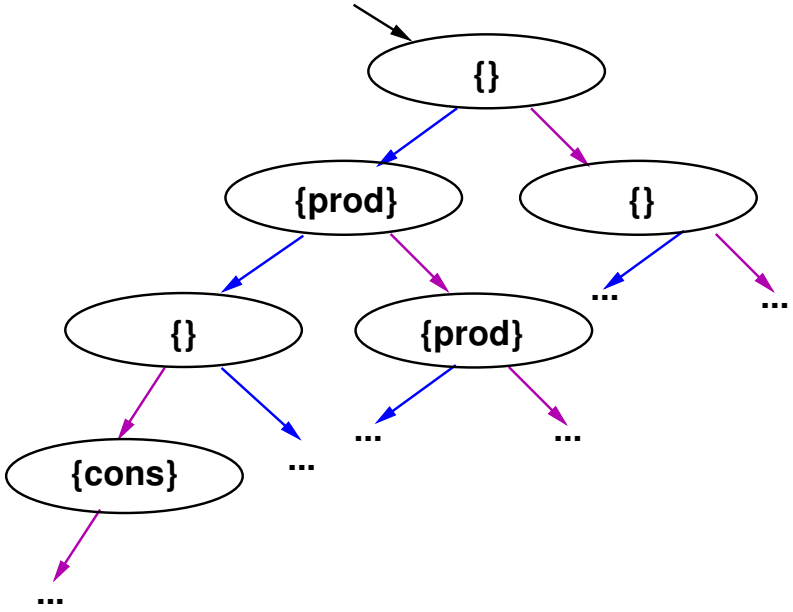


Séquences et arbres

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 mieux connu : 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 (voir suite).

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 $[\![\mathcal{K}]\!] \subseteq [\![\phi]\!]$.

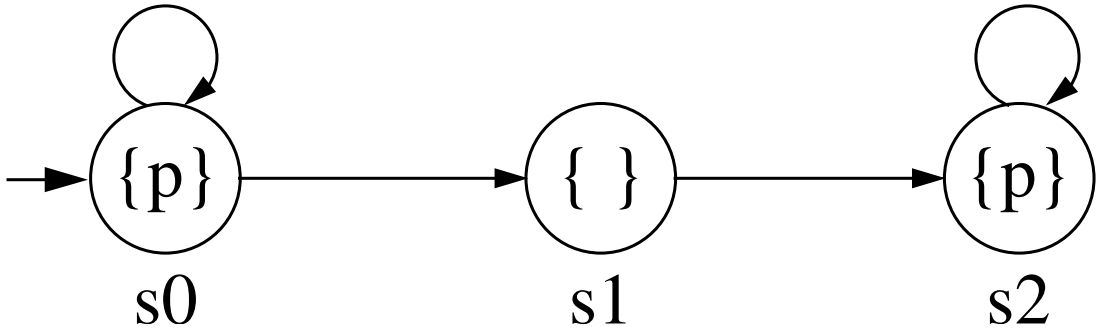
Définition: Si $[\![\mathcal{K}]\!] \subseteq [\![\phi]\!]$ alors on écrit $\mathcal{K} \models \phi$.

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:

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

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