Langages et Décidabilité
Modélisation par automates
Grammaires : Séance 4

Marie-Laure Potet

Grenoble INP-Ensimag

February 25, 2021

Summary

Classification des langages et algorithmes

Rappel :

- Langages réguliers
- Langages hors-contexte
- Langages sous-contexte
- Langages généraux
- Autres langages ($V^* - L(G), \forall G$)

Soit L un langage. Le problème du mot est :

$$w \in L?$$

Un programme répondant à ce problème est appelé un reconnaisseur.

Problème du mot

Résultats et paradigmes de reconnaisseurs :

Classe	Décidable	paradigme de reconnaisseur
Régulier	oui	automate d'état fini
Hors-contexte	oui	automate à pile
Sous-contexte	oui	automate borné linéairement
Général	non	Machine de Turing

On va voir : programmation des reconnaisseurs pour sous-contexte, hors-contexte et régulier.

Autres problèmes

Classe	Régulier	Hors-contexte	Sous-contexte	Général
$L(G) = \emptyset$	oui	oui	non	non
$L(G)$ infini	oui	oui	non	non
$L(G1) = L(G2)$	oui	non	non	non

oui : il existe un algorithme
non : il n'existe pas d'algorithme

Comment ?

Summary

Reconnaisseurs sous-contexte

Soit $G = (V_T, V_N, S, R)$ avec les règles de la forme $u \to v$ et $|u| \leq |v|$. Il n'y a donc pas d'ϵ-règle. On veut répondre à la question :

$$w \in L(G)?$$

Principe :

1. On énumère toutes les chaînes $x \in (V_T \cup V_N)^*$, accessibles à partir de S et telles que $|x| \leq |w|$.
2. si on a trouvé w la réponse est positive
3. si on n'a pas trouvé w la réponse est négative (ça ne sert à rien de continuer on ne pourra pas générer w)

$\Rightarrow$ On gère l'ensemble E des chaînes x en cours de traitement et un dictionnaire pour traiter au plus une fois chaque chaîne x.

Opérations utiles

- Application des règles :

$$next(d) = \{xvy : \exists\ u\ .\ d = xuy\ et\ u \to v \in R\}$$

$\Rightarrow$ applique à d toutes les règles possibles à toutes les occurrences. Exemple :

$aA \to aa$, $Ba \to bb$ et next(BaA)={Baa, bbA}.

- Opérations sur l'ensemble E : choix d'un élément (`choix(E)`) + opérations ensemblistes classiques ($-, \cup, \ldots$)

Algorithme

```
Procedure Reconnaitre(w : Mot) is
begin
   E := {S} ;
   dictio := {} ;
   while not(E={}) and not (w in E) loop
     d := choix(E) ;
     E := E -{d} ;
     dictio := dictio \/ {d} ;
     E :=  E \/ next(d) - {m : |m| >|w|} -dictio ;
   end loop ;
   if (w in E) then putline("le mot est dans L(G)") ;
   else putline ("le mot n'est pas dans L(G)") ;
end Reconnaitre ;
```

Exemple

1) Exemple :

$$S \to aSBc \mid abc \quad cB \to Bc \quad bB \to bb$$

On s'intéresse à la chaîne : *ababcc*

2) Arrêt du Programme : on traite au plus une fois chaque chaîne de longueur $|w|$. On a :

- dictio croît strictement
- $E \cap \ dictio = \emptyset$ (invariant)

$\Rightarrow$ Au pire on énumère toutes les chaînes x sur $(V_T \cup \ V_N)$ telles que $|x| \leq \ |w|$.

Summary

Reconnaisseur hors-contexte

On peut simplifier le programme précédent pour les hors-contexte en utilisant les dérivations canoniques (par exemple la plus à gauche).

- On réécrit uniquement le non-terminal le + à gauche. Rappel :

$$uAvBr \Longrightarrow^* u\alpha vBr \Longrightarrow^* u\alpha v\beta r$$

équivalent à

$$uAvBr \Longrightarrow^* uAv\beta r \Longrightarrow^* u\alpha v\beta r$$

- on peut vérifier/éliminer les symboles de V_T en début de chaîne

$$av \Longrightarrow^* w$$
avec $a \in V_T$
si et seulement si
w s'écrit aw'

On suppose pour simplifier pas de ϵ-règle ni de 1-règle.

Opérations utiles

- On manipule des couples (d, w) avec :
  - $d \in (V_T \cup V_N)^*$: mot de dérivation
  - $w \in V_T^*$: mot à reconnaître
  - Objectif : $d \Longrightarrow^* w$?
- configuration initiale : (S, w) et w le mot à reconnaitre
- Opération next:
  1. $next(xd, xw) = \{(d, w)\}$ ssi $x \in V_T$ (effacement)
  2. $next(xd, yw) = \emptyset$ ssi $x \in V_T$ et $x \neq y$
  3. $next(Ad, w) = \{(ud, w) \mid A \rightarrow u \in R\}$
- configuration gagnante : (ϵ, ϵ)

Algorithme

```
Procedure Reconnaitre(w : Mot) is
begin
    E := {(S, w)} ;
    while not(E={}) and not((epsilon, epsilon) in E)
    loop
     (d, r) := choix(E) ;
     E := E - {(d, r)}  ;
     E := E \/ {m |m in next(d, r) and |m|<= |w|};
    end loop ;
    if ((epsilon, epsilon) in E)
       then putline("le mot est dans L(G)") ;
       else putline ("le mot n'est pas dans L(G)") ;
end Reconnaitre ;
```

⇒ Automate à pile : `d` est la pile.

Exemple

(1) $S \to aSb$
(2) $S \to ab$

Et :

1. *aabb*
2. *aaba*

Arrêt ? Soit (m, w). On fait soit décroitre w soit $|w| - |m|$ (poids de 2 pour les non-terminaux et de 1 pour les terminaux).

$\Rightarrow$ on ne peut pas faire mieux . . . sauf pour des sous-classes de grammaires : au plus un successeur par la fonction next (TL2)

Summary

Reconnaisseur régulier : plusieurs solutions

Soit $A = (V, Q, q0, F, \delta)$ un automate déterministe avec q_erreur l'état puits. Exemple : a^+b^* et
$A = (\{a, b\}, \{q0, q1, q_erreur\}, q0, \{q1, q2\}, \delta)$avec δ :

	a	b
q0	q1	q_erreur
q1	q1	q2
q2	q_erreur	q2
q_erreur	q_erreur	q_erreur

$\Rightarrow$ Plusieurs implémentations possibles :

1. codage de l'automate par un tableau :
   - delta : array [Q, V] of Q ; une variable du programme initialisée par la fonction de transition δ
2. codage par flot de contrôle.
   - état=label, transition=goto.

Opérations utiles

- V'=V $\cup$ {$, erreur}
- lire_car : la fonction de lecture retournant :
  - l'élément courant dans V
  - ou $ si fin de chaîne
  - ou `erreur` si pas un élément de V.

Implémentation 1

```
Procedure Reconnaitre( ) is
begin
    q_cour := q0 ;
    c := lire_car() ;
    while (c in V) and not(q_cour=q_erreur) loop
      q_cour := delta(c, q_cour);
      c := lire_car() ;
    end loop ;
    if (q_cour in F) and (c=$)
       then putline("le mot est dans L(A)") ;
       else putline ("le mot n'est pas dans L(A)") ;
end Reconnaitre ;
```

Implémentation 2

```
Procedure Reconnaitre( ) is
begin
   q0  : c := lire_car() ;
         if c=a then goto q1 ; else goto q_erreur ;
   q1  : c := lire_car() ;
         if c=a then goto q1 ;
         if c=b then goto q2 ;
         if c=$ then goto fin ; else goto q_erreur ;
   q2  : c := lire_car() ;
         if c=b then goto q2 ;
         if c=$ then goto fin ; else goto q_erreur ;
   q_erreur : putline ("le mot n'est pas dans L(A)") ;
              goto reconnaitre ;
  fin : putline("le mot est dans L(A)") ;
  reconnaitre :
end Reconnaitre ;
```

Implémentation 1 (non déterministe)

Ici delta(q, c) fournit un ensemble d'états (potentiellement vide)

```
begin
    Q_cour := {q0} ;
    c := lire_car() ;
    while (c in V) and not(Q_cour={}) loop
      State := Q_cour; Q_cour :={} ;
      wile not (State={}) loop
         q := choix(State) ; State := State -{q} ;
         Q_cour  := Q_cour \/ delta(c, q);
      end loop ;
      c := lire_car() ;
    end loop ;
    if not (Q_cour /\ F ={}) and (c=$)
       then putline("le mot est dans L(A)") ;
       else putline ("le mot n'est pas dans L(A)") ;
end  ;
```

Summary

Programmes à états finis

Un automate permet de modéliser (si nombre fini de valeurs):

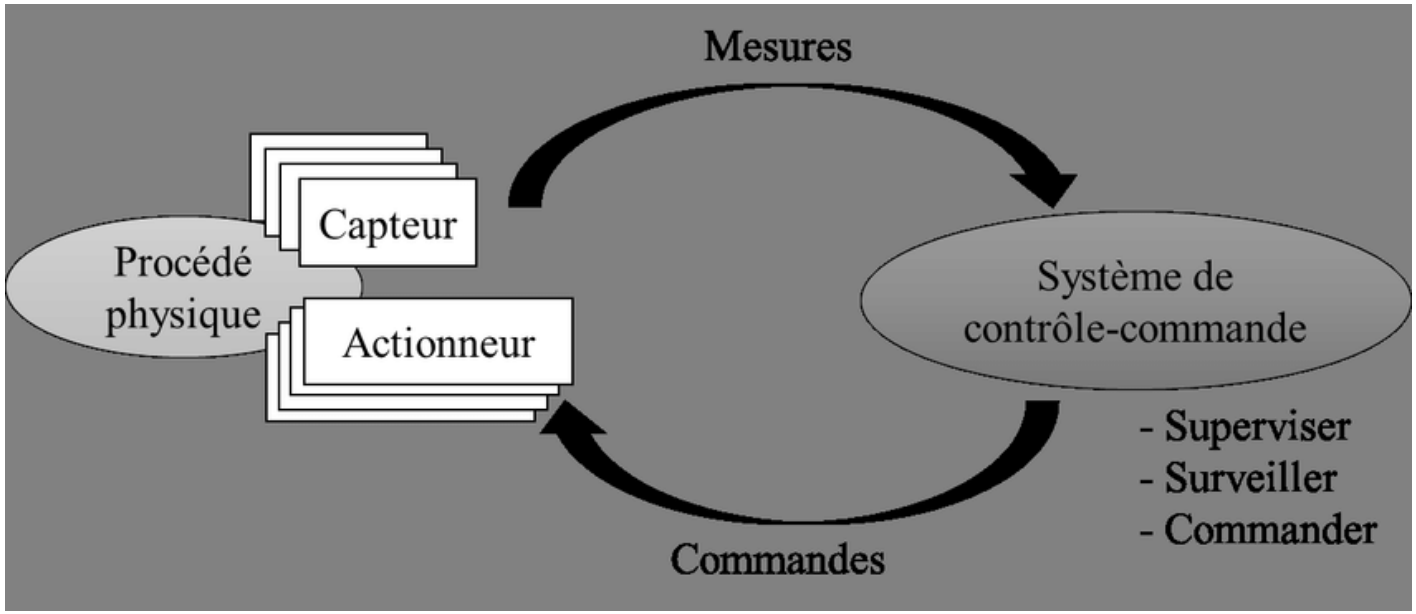
- les entrées admises par un programme
- les états du programme sous la forme des valeurs des variables
- les comportements du programme : la suite des états et des transitions qui peuvent s'enchaîner

Domaines d'applications : par exemple les programmes de contrôle/commande (commande d'un pool d'ascenseurs, gestion de réservoirs de fluide, . . .)

$\Rightarrow$ On peut alors vérifier des propriétés sur ce modèle :

- états dangereux non atteignables (le niveau d'un réservoir doit rester entre 2 valeurs min/max)
- comportements interdits (on ne peut pas ouvrir 2 vannes en même temps)
- . . .

Domaine d'application



Automate industriel, programme de contrôle/commande

Cycle : 1) acquisition des données 2) traitement 3) résultats

Nombre fini d'états : état interne des capteurs/actionneurs
Exemple : gestion de réservoirs de fluide. Propriété : le niveau d'une cuve doit rester entre 2 valeurs min/max.

Illustration

1. Produire un automate à partir d'un programme
2. Combiner cet automate avec son contexte d'utilisation
3. Vérifier des propriétés

$\Rightarrow$ utilise les opérations sur les automates :

- atteignabilité d'état
- intersection d'automate = ensemble des chemins ayant des comportements de A_1 et A_2.

$\Rightarrow$ Démarche automatisable

Exemple

Soit P le programme suivant :

```
x:=0 ; y := 1 ;
while (true) loop
    z := lire() ;
    if z=a then  x:=x+2 ; y := y+1 ;
    elsif z=b then  x:=x+1 ; y := y+2 ;
    elsif z=c then  x:=x+1 ; y := y+1 ;
    if x > 2 then x:=x-2;
    if y > 3 then y:=y-3;
end loop ;
```

- un nombre fini d'entrées : a, b ou c
- On peut énumérer l'ensemble des valeurs prises par les couples (x, y) avec :

$$x \in 0..2 \wedge y \in 1..3$$

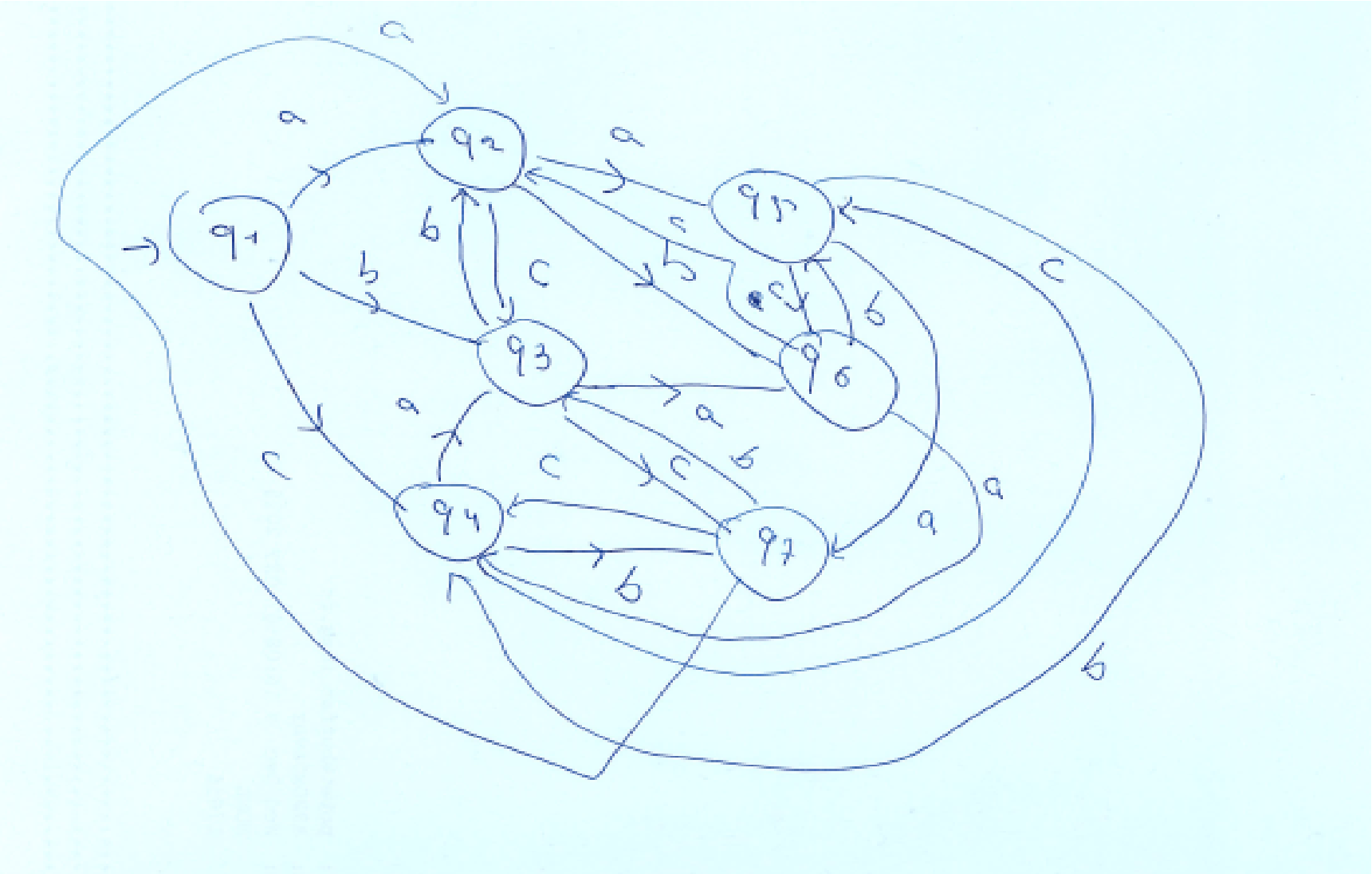
Automate P

	a	b	c
q1 : x=0 & y=1	q2 : x=2 & y=2	q3 : x=1 & y=3	q4 : x=1 & y=2
q2 : x=2 & y=2	q5 : x=2 & y=3	q6 : x=1 & y=1	q3 : x=1 & y=3
q3 : x=1 & y=3	q6 : x=1 & y=1	q2 : x=2 & y=2	q7 : x=2 & y=1
q4 : x=1 & y=2	q3 : x=1 & y=3	q7 : x=2 & y=1	q5 : x=2 & y=3
q5 : x=2 & y=3	q7 : x=2 & y=1	q4 : x=1 & y=2	q6 : x=1 & y=1
q6 : x=1 & y=1	q4 : x=1 & y=2	q5 : x=2 & y=3	q2 : x=2 & y=2
q7 : x=2 & y=1	q2 : x=2 & y=2	q3 : x=1 & y=3	q4 : x=1 & y=2

Tout état peut être vu comme final : tout préfixe d'une trace d'exécution est une trace d'exécution.

$\Rightarrow$ P peut être produit automatiquement (simulation)

Automate P



Etape 2

On suppose maintenant que l'environnement E suit le comportement suivant :

$$(a+b)(c(ac+b))^*$$

$\Rightarrow$ On veut construire l'automate S décrivant le programme interagissant avec cet environnement.

- Revient à calculer : $S = P \cap E$

Automate de E :

	a	b	c
p1	p2	p2	-
p2	-	-	p3
p3	p4	p2	-
p4	-	-	p2

Produit d'automates

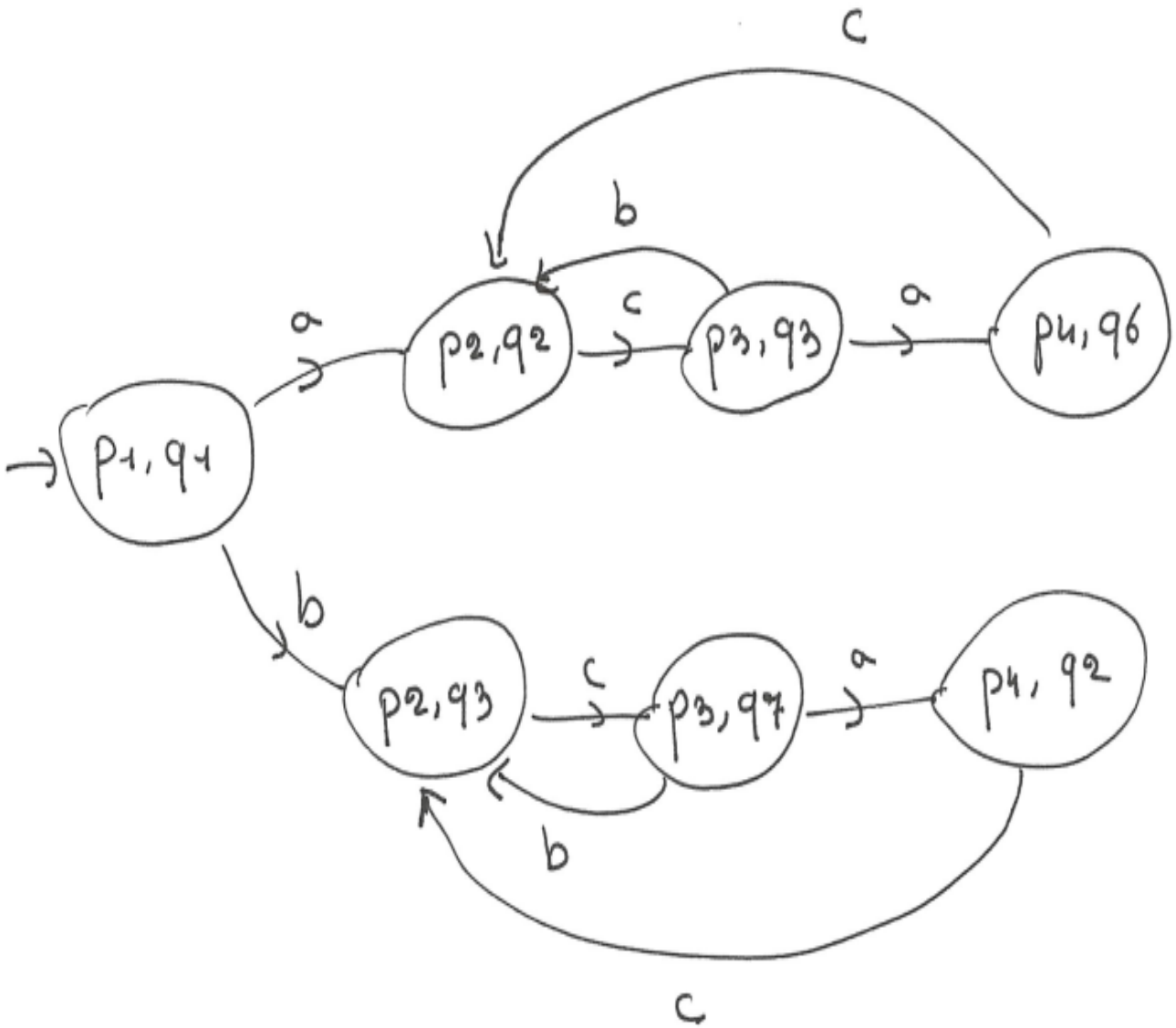
$\Rightarrow$ Intersection des 2 automates E et P :

$$((p1, q1), x, (p2, q2)) \in \delta_S$$

si et seulement si :

1. $(p1, x, p2) \in \delta_E$
2. $(q1, x, q2) \in \delta_P$

Automate produit S



Etape 3

On veut maintenant vérifier des propriétés sur l'automate S :

- P_1 : On n'atteint jamais un état tel que les valeurs de x et y vérifient le prédicat $y = x + 1 \wedge x \neq 0$
  - Calcul des états atteignables dans S
  - Si aucun ne vérifie le prédicat ci-dessus la propriété est vraie
- P_2 : pour tout chemin si on atteint $q2$ on atteindra $q3$ avant de revenir en $q2$
  - Intersection avec l'automate $\neg P_2$
  - Si l'automate obtenu reconnait le langage vide la propriété P_2 est vérifiée

Propriété 1

On n'atteint jamais un état tels que les valeurs de x et y vérifient le prédicat $y = x + 1 \wedge x \neq 0$?

	valeurs	$y = x + 1 \wedge x \neq 0$
q1	x=0 & y=1	✗
q2	x=2 & y=2	✗
q3	x=1 & y=3	✗
q4	x=1 & y=2	✓
q5	x=2 & y=3	✓
q6	x=1 & y=1	✗
q7	x=2 & y=1	✗

✓ : vérifie le prédicat. ✗ : ne vérifie pas le prédicat.

$\Rightarrow$ Effectivement aucun état atteignable de S n'est de la forme $(p_i, q4)$ et $(p_i, q5)$ $\forall$ i .

Propriété 2

$\Rightarrow$ si on atteint $q2$ on atteindra $q3$ avant de revenir en $q2$ (pour tout chemin)

Méthode :

- On construit l'automate O (non déterministe) qui reconnait la propriété $\neg P_2$: il existe un chemin qui passe 2 fois par $q2$ sans atteindre $q3$ (état final = propriété vérifiée)

- On construit un produit (une intersection avec condition sur les états) entre O et S : si le résultat est vide aucun chemin de S vérifie $\neg P_2$ sinon on obtient des contre-exemples

Automate *O*

Automate non déterministe :

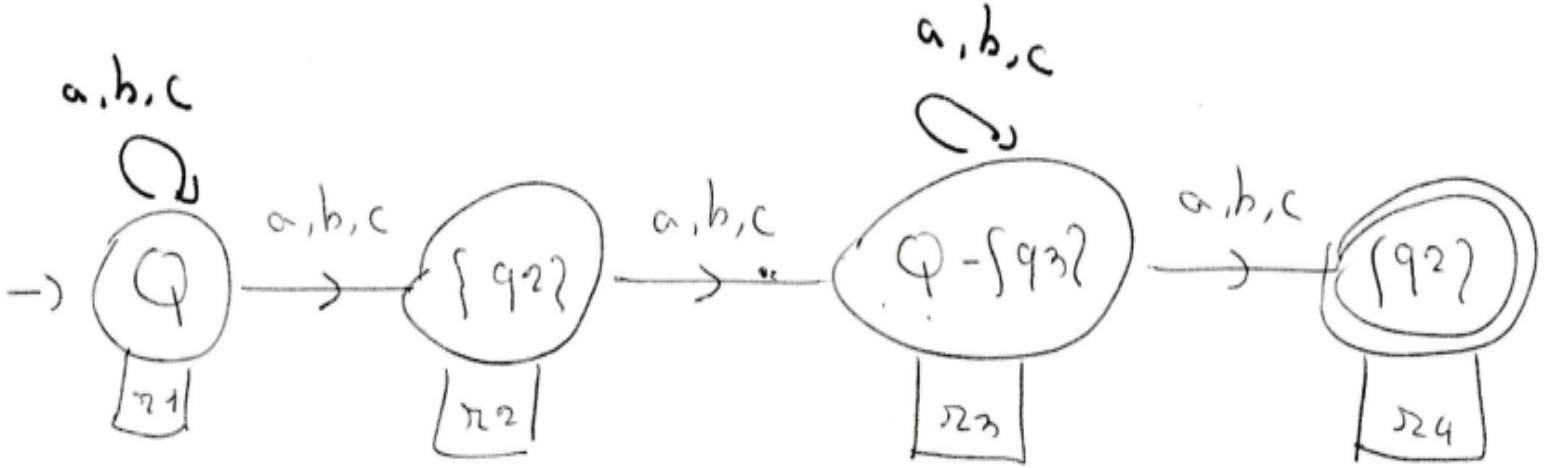
	a	b	c
r1	r1, r2	r1, r2	r1, r2
r2	r3, r4	r3, r4	r3, r4
r3	r3, r4	r3, r4	r3, r4

On associe aux états *ri* de *O* les états de *S* qui ont été atteints :

r1 : on a atteint n'importe quel état de P
r2 : on a atteint q2
r3 : on a atteint n'importe quel état de *P* sauf q3
r4 : on a atteint q2

*r*4 : état final

Automate O : encode la propriété non P2



Produit d'automates

$\Rightarrow$ On fait le produit entre l'automate S et l'automate O de la manière suivante :

- intersection classique
- un état (ri, pj, qk) est valide si et seulement si qk fait partie des états possibles pour ri.
  - (r1, p2, q2) valide (r1 n'importe quel état atteint)
  - (r3, p3, q3) non valide (r3 n'importe quel état atteint sauf q3)
- un état non valide n'est pas productif (état puits)
- les chemins qui mène à $r4$ viole la propriété $P2$

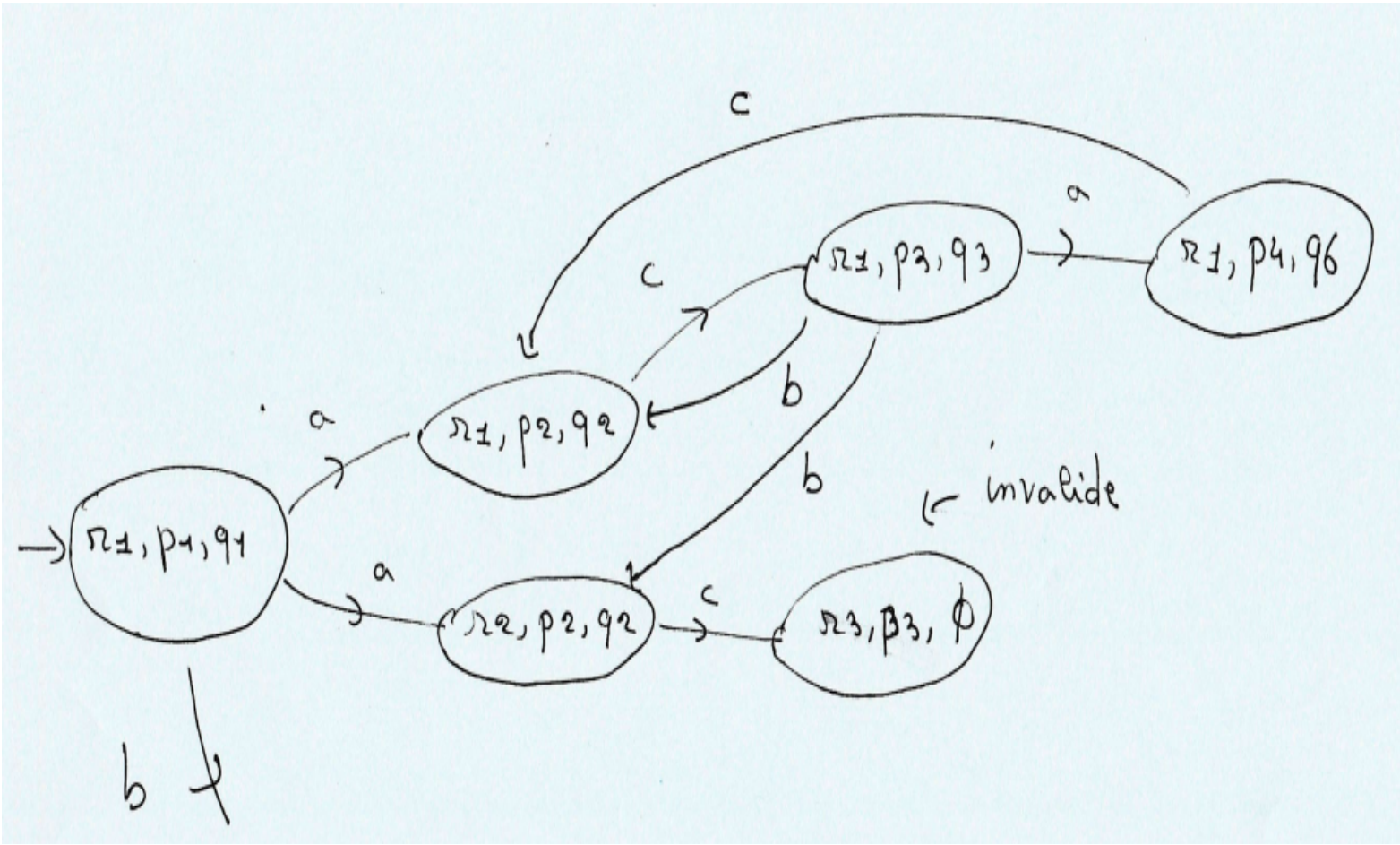
Résultat

On ne traite que le sous-automate de S partant avec *a* (même traitement et résultat pour la branche avec *b*)

	a	b	c
(r1, p1, q1)	(r1, p2, q2), (r2, p2, q2)	non traité	-
(r1, p2, q2)	-	-	(r1, p3, q3)
(r2, p2, q2)	-	-	(r3, p3, q3) non valide
(r1, p3, q3)	(r1, p4, q6)	(r1, p2, q2) (r2, p2, q2)	-
(r1, p4, q6)	-	-	(r1, p2, q2)

Les états *ri* désignent les états de *P* qui ont été atteints :
On n'atteint jamais un état final (contenant r4) : on reconnait donc le langage vide : aucun chemin ne vérifie $\neg P2$ (donc tous vérifient *P2*).

Automate produit S et O



On n'atteint jamais un état avec $r4$.

Modèles par Automates et Vérification

Il existe des outils qui mettent en place de telles démarches :

- Production d'automates à partir d'une formalisation haut niveau (plus expressif que ce qu'on a vu, exemple communication)
- Langages pour énoncer des propriétés (logique temporelle) qui peut être traduit en vérification sur les automates (atteignabilité, existence de chemin . . .)
- Vérification ou production de contre-exemples
- démarche basée sur des automates et des opérations sur les automates

Exemple l'outil UPPAAL