

Un corrigé : X-ENS INFO C 2026

Vérification de systèmes concurrents

Pour toutes les remarques ou corrections, vous pouvez m'envoyer un mail à
galatee.hemery@gmail.com

1 Partie I. Systèmes de transitions

Question 1. Montrons l'équivalence par double implication.

(a) $\implies$ (b) :

On suppose I invariant.

Soit $s \in I$. Et soit π une exécution infinie à partir de s s'il en existe un.

On a pour une suite (α_i) d'actions :

$$\pi : s = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_{k-1}} s_k \xrightarrow{\alpha_k} \dots$$

et par récurrence sur i , on a $s_i \in I$ grâce au caractère invariant de I .

Donc $\pi \in P$.

Ceci est vrai pour tout π donc $s \models P$.

Ceci est bien vrai pour tout choix de $s \in I$.

(b) $\implies$ (a) :

On suppose que $\forall s \in I, s \models P$.

Le fait qu'il n'y ait aucun état terminal assure l'existence de calcul infini depuis tout état : en effet, on peut poursuivre n'importe quel exécution finie en choisissant une action disponible depuis le dernière état et cela sans s'arrêter pour construire une exécution infinie.

Soit $s \in I$ et soit α une action disponible depuis s , on a alors l'existence d'une exécution infinie depuis $\delta(s, \alpha)$ et en ajoutant s devant, on a une exécution infinie depuis s .

Comme $s \in I$, on a $s \models P$, la trace de cette exécution est dans I^ω donc en particulier $\delta(s, \alpha) \in I$.

Ceci est vrai pour tout choix de s et de α donc I est un invariant.

Remarque 1.1. Un Σ au lieu d'un S en haut de la page 6 dans le sujet original.

Question 2.

1. P_1 est une propriété de sûreté.

En effet, pour $w \notin P_1$, on a $|w|_s \geq 1$ pour tout $s \in S$ donc on peut considérer $i_0 = \max_{s \in S}(\min(\{i \in \mathbb{N} \mid w_i = s\}))$, S étant fini.

On prend le préfixe $\hat{w} = w_0 \dots w_{i_0}$ contenant au moins une fois chaque lettre de S .

On obtient bien que pour tout $w' \in S^\omega$, $\hat{w}w' \notin P_1$ car il contient au moins une fois chaque lettre de S .

2. P_2 n'est pas une propriété de sûreté dès lors que S contient au moins deux éléments, sinon $P_2 = S^\omega$ est une propriété de sûreté car $S^\omega \setminus P_2 = \emptyset$.

On note $S = \{s_1, \dots, s_p\}$ avec $p = |S| \geq 2$.

Soit $w = s_1 s_1 \dots \notin P_2$. Pour tout préfixe $\hat{w} = s_1 \dots s_1$, et $w' = s_1 s_2 \dots s_p s_1 s_2 \dots s_p \dots$, on a $\hat{w}w' \in P_2$.

3. P_3 n'est pas une propriété de sûreté dès lors que S contient au moins deux éléments, sinon $P_3 = \emptyset$ est une propriété de sûreté car $\hat{w}w' \notin P_3$ peu importe les mots choisis.

On note $S = \{s_1, \dots, s_p\}$ avec $p = |S| \geq 2$.

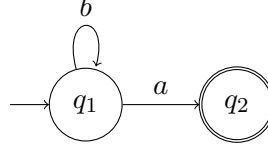
Soit $w = s_1 s_2 \dots s_p s_1 s_2 \dots s_p \dots \notin P_3$. Pour tout préfixe $\hat{w}$, et $w' = s_1 s_1 \dots$, on a $\hat{w}w' \in P_3$ car on a s_2 dans $\hat{w}$ fini uniquement.

Question 3. Tous les Σ devraient être des S dans le sujet original.

1. Les préfixes interdits pour P_1 sont tous les mots contenant au moins un a . En effet, dans ce cas, peu importe la suite, on a des préfixes qui contiennent un a . Pour un préfixe sans a , une suite sans a permet d'être dans P_1 .

Les préfixes minimaux sont les mots contenant exactement un a en dernière position.

L'automate suivant reconnaît ce langage :

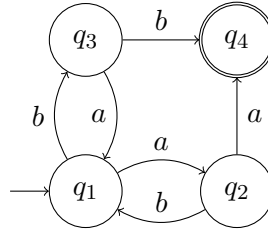


P_1 est donc régulière.

2. Les préfixes interdits pour P_2 sont tous les mots u tels que $||u|_a - |u|_b| \geq 2$. En effet, dans ce cas, peu importe la suite, on a des préfixes qui ne vérifient pas la propriété. Pour un préfixe qui ne vérifie pas cela, une suite contenant une alternance de a et de b en commençant par la lettre qui apparaît le moins dans le préfixe permet d'être dans P_2 .

Les préfixes minimaux sont les mots u tels que $|u|_a = |u|_b + 2$ ou $|u|_a + 2 = |u|_b$ et tels que tout préfixe strict x de u est tel que $||x|_a - |x|_b| \leq 1$.

L'automate suivant reconnaît ce langage :



P_2 est donc régulière.

3. Les préfixes interdits pour P_3 sont tous les mots u tels que $|u|_a > |u|_b$. En effet, dans ce cas, peu importe la suite, on a des préfixes qui ne vérifient pas la propriété. Pour un préfixe qui ne vérifie pas cela, une suite contenant uniquement des b permet d'être dans P_3 .

Les préfixes minimaux sont les mots u tels que $|u|_a = |u|_b + 1$ et tels que tout préfixe strict x de u est tel que $|x|_a \leq |x|_b$.

Le langage $\hat{P}_3$ n'est pas régulier.

Supposons par l'absurde qu'il le soit et considérons N l'entier donné par le lemme de l'étoile.

Soit $u = b^N a^{N+1}$ de longueur $2N + 1 > N$.

D'après le lemme de l'étoile, il existe des mots x, y, z tels que :

- $u = xyz$
- $|xy| \leq N$
- $y \neq \varepsilon$
- $\forall k \geq 0, xy^kz \in \hat{P}_3$

Or ces conditions imposent $y = b^m$ avec $1 \leq m \leq N$.

Pour $k = 0$, on a $xz = b^{N-m} a^{N+1} \notin \hat{P}_3$. Contradiction.

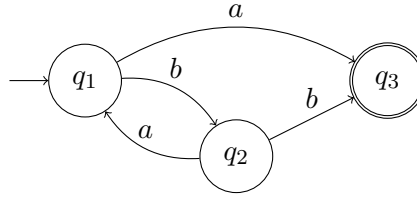
Ainsi, P_3 n'est pas régulière.

4. Les préfixes interdits pour P_4 sont tous les mots u tels que $|u|_b < |u|_a$ ou $|u|_b > |u|_a + 1$. En effet, dans ce cas, peu importe la suite, on a des préfixes qui ne vérifient pas la propriété. Pour un préfixe qui ne vérifie pas cela, une suite contenant une alternance de a et de b en commençant par la lettre qui apparaît le moins dans le préfixe permet d'être dans P_2 .

Les préfixes minimaux sont les mots u tels que $|u|_a = |u|_b + 1$ ou $|u|_a + 2 = |u|_b$ et tels que

tout préfixe strict x de u est tel que $|u|_a \leq |u|_b \leq |u|_a + 1$.

L'automate suivant reconnaît ce langage :



P_4 est donc régulière.

Question 4. Montrons par double inclusion que l'ensemble des préfixes interdits est exactement $\hat{P} \cdot S^*$ où $\cdot$ désigne la concaténation de langage.

Soit u un préfixe interdit de P :

- s'il est minimal, il est dans $\hat{P} \cdot S^*$ en prenant ε dans S^* ;
- sinon, il admet un préfixe strict x qui est un préfixe interdit, on peut prendre le plus petit, il est alors minimal et $u = xy$ avec $x \in \hat{P}, y \in S^*$.

Soit réciproquement $u \in \hat{P} \cdot S^*$, alors $u = xy$ avec $x \in \hat{P}$ et $y \in S^*$.

Pour tout $v \in S^\omega$, $xv \notin P$ donc en particulier, pour tout $v' \in S^\omega$, $xyv' \notin P$ donc xy est un préfixe interdit.

On en déduit une partie du résultat attendu : si $\hat{P}$ est régulier, alors $\hat{P} \cdot S^*$ est régulier car les langages réguliers sont stables par concaténation.

Réciproquement, si $\hat{P} \cdot S^*$ est régulier, alors il existe un automate qui reconnaît ce langage. Quitte à le déterminer, on peut supposer cet automate déterministe.

Il suffit de supprimer les transitions sortant des états finaux pour obtenir un automate qui reconnaît $\hat{P}$. Une fois que l'on a un préfixe interdit, si on continue le chemin dans l'automate, on aura un préfixe qui n'est plus minimal. On conserve bien tous les préfixes minimaux car on conserve tous les chemins acceptants aboutissant à un état final sans passer avant par un état final.

Le caractère déterministe permet de n'avoir qu'un chemin par mot et donc pas de risque qu'un début de chemin corresponde à un préfixe interdit sans atteindre un état final.

Question 5.

```

1 let reachable t s0 target =
2   let vu = Array.make t.size false in
3   let rec visiter s =
4     if target s then true
5     else begin
6       vu.(s) <- true ;
7       let rec suivants l = match l with
8         | [] -> false
9         | (a,s1)::q when vu.(s1) -> suivants q
10        | (a,s1)::q -> visiter s1 || suivants q in
11       suivants (t.next s)
12     end in
13   visiter s0

```

Il s'agit d'un parcours en profondeur. Le tableau `vu` permet de ne visiter les sommets qu'une seule fois. La complexité est linéaire en la somme du nombre d'états et d'actions de `t` car chaque

appel à **visiter** a une contribution linéaire en son degré et qu'il y a un appel à **visiter** au plus par état. Les fonctions **target** et **t.next** sont supposées en temps constant et sont appelées au plus une fois par état.

Question 6. L'idée est de simuler le calcul dans l'automate du mot construit par la trace d'exécution et de se souvenir à la fois de l'état courant du système de transition et de l'état dans l'automate lors d'une exécution.

Lorsqu'on atteint dans le nouveau système un état qui est associé à un état final de l'automate, alors on a lu un préfixe interdit dans la trace et on sort de P la propriété de sûreté. Réciproquement, si ce n'est jamais atteint, on construit un mot dont aucun préfixe n'est dans $\hat{P}$ donc par la propriété admise, la trace est dans P .

Comme le système de transition est sans état terminal, il existe une exécution infinie depuis tout état donc on a $s \not\models P$ si et seulement si on peut atteindre un état associé à un état final de l'automate depuis s associé à l'état initial.

La construction formelle ressemble à celle de l'automate produit :

- $S' = S \times Q$ où Q l'ensemble des états de $\mathcal{A}$;
- $\forall s \in S, q \in Q, \mathcal{D}'((s, q)) = \mathcal{D}(s)$;
- $\delta' : S' \times A \rightarrow S'$ fonction partielle telle que $\delta'((s, q), a) = (\delta(s, a), t(q, s))$ où t est la fonction de transition de $\mathcal{A}$.

Ainsi, $\iota : S \rightarrow S', s \mapsto (s, q_0)$ où q_0 est l'état initial de $\mathcal{A}$ et $C = \{(s, q) \mid s \in S, q \in F\}$ avec F l'ensemble des états finaux de $\mathcal{A}$.

2 Partie II : Ultime invariance

Question 7. Pour avoir $s \models \Phi^\omega$, il est nécessaire d'avoir $s \in \Phi$ car c'est le premier état de toute trace.

On a $s_2 \not\models \Phi^\omega$ car $s_2 s_3 s_5 s_5 \dots$ est une trace d'un calcul à partir de s_2 et $s_3 \notin \Phi$.

Pour $s \in \{s_4, s_5, s_6\}$, on a $s \models \Phi^\omega$ car le sous ensemble est invariant. Ce sont les seuls états pour lesquels c'est vrai.

On a $s_0 \not\models \mathbf{EA}\Phi$ car $s_0 s_1 s_1 s_1 \dots$ est une trace d'un calcul à partir de s_0 et $s_1 \notin \Phi$.

On a $s_1 \not\models \mathbf{EA}\Phi$ car $s_1 s_1 s_1 \dots$ est une trace d'un calcul à partir de s_1 et $s_1 \notin \Phi$.

Pour $s \in \{s_2, s_3, s_4, s_5, s_6\}$, on a $s \models \mathbf{EA}\Phi$. En effet, depuis ses états, le seul accessible hors de Φ est s_3 et après une action, on est de nouveau dans Φ et on y reste par invariance de l'ensemble au dessus. Ce sont les seuls états pour lesquels c'est vrai.

Question 8. On suppose que $s \not\models \mathbf{EA}\Phi$, on a alors l'existence d'un calcul π de trace w tel que $\forall i \in \mathbb{N}, \exists j \geq i, w_j \notin \Phi$. En particulier, au moins un élément s' hors de Φ apparaît une infinité de fois dans w .

En considérant deux occurrences distinctes de s' dans la trace de π , on obtient la trace d'une exécution finie de longueur non nulle de s' à s' . Donc il existe bien s' accessible depuis s tel que $s' \notin \Phi$ et s' fait un cycle sur lui-même.

Réciproquement, s'il existe bien s' accessible depuis s tel que $s' \notin \Phi$ et s' fait un cycle sur lui-même. On peut construire une exécution π partant de s , allant à s' et empruntant une infinité de fois le cycle sur s' de sorte que s' apparaît arbitrairement loin dans la trace de π donc $s \not\models \mathbf{EA}\Phi$.

Question 9. Soit s_1 et s_2 deux états tels que :

- s_1 apparaît avant s_2 dans l'énumération;

— il existe une exécution de s_1 à s_2 dans T .

On considère s_3 le plus proche ancêtre commun de s_1 et s_2 dans l'arbre considéré (la racine du plus petit sous-arbre contenant s_1 et s_2).

Si $s_2 = s_3$, alors il existe une exécution de s_2 vers s_1 correspondant à la branche de l'arbre et en concaténant cette exécution et celle que l'on a par hypothèse, on obtient un cycle de s_1 vers s_1 . $s_1 = s_3$ est impossible car l'ordre est postfixe donc s_1 apparaîtrait après s_2 .

Sinon, s_3 est un sommet différent de s_1 et s_2 et s_1 et s_2 apparaissent dans des sous-arbres différents. Montrons que ce cas est également impossible.

s_1 apparaît avant donc le sous-arbre contenant s_1 doit également faire apparaître tout ce qui est accessible depuis s_1 et qui n'a pas été visité avant par propriété du parcours en profondeur postfixe : en particulier, s_2 devrait apparaître car il existe une exécution de s_1 vers s_2 . Ce n'est pas le cas, car s_2 apparaît dans un autre sous-arbre. Ce dernier cas est donc impossible.

Question 10.

```
1 let outer_iter t s0 f =
2   let marked = Array.make t.size false in
3   let rec parcours_postfixe s =
4     marked.(s) <- true;
5     let rec voisins l = match l with
6       | [] -> ()
7       | (_,s')::q when marked.(s') -> voisins q
8       | (_,s')::q -> parcours_postfixe s' ;
9                               voisins q in
10    voisins (t.next s) ;
11    f s in
12    parcours_postfixe s0
```

Question 11. La fonction `cycle_check` réalise un parcours en profondeur depuis un sommet et vérifie s'il existe une transition revenant à l'état `start` de départ depuis un des sommet exploré donc accessible depuis `start`.

L'exception `Found` est levée dès qu'un retour à `start` est obtenu. Sinon, `cycle_check` termine son parcours une fois tous les sommets accessibles non marqués dans des appels précédents traités exactement une fois sans rien renvoyer (type de retour `unit`).

La signature de la fonction est `state -> unit`.

Initialement `marked` contient uniquement des `false` avant tous les appels de `cycle_check`.

Avant un appel `inner_visit s`, on a `marked.(s')` qui vaut `true` pour exactement tous les sommets s' accessibles traités ou en cours de traitement et en fin d'appel : soit l'exception `Found` a été levée dans le cas où un sommet accessible non déjà traité précédemment permet un retour à `start`, soit on a traité tous les sommets accessibles non déjà traités précédemment depuis `s` et dans ce cas `marked.(s')` a été mis à jour à `true` pour tous les nouveaux sommets accessibles.

La fonction `verify` parcourt l'ensemble des sommets accessibles depuis `s0` dans l'ordre imposé par la question 9 grâce à la fonction `outer_iter`. Pour chacun de ces sommets, lorsqu'il n'appartient pas à Φ , on appelle `cycle_check` qui lève `Found` dans le cas où un cycle sur lui-même existe. Si aucun cycle n'est trouvé de cette façon, la fonction renvoie `true`.

Enfin, supposons par l'absurde qu'il existe un état $s_2 \notin \Phi$ accessible depuis s_0 et admettant un cycle sur lui-même mais que la fonction `verify` ne lève jamais l'exception `Found` dans son exécution. On considère s_2 le sommet qui est le premier vérifiant cela dans l'ordre de l'énumération de la question 9.

Cela arrive si lors de l'appel `cycle_check s2` un des sommets du cycle s_3 a été marqué dans

un appel précédent sur un sommet s_1 qui n'est pas dans Φ et accessible depuis s_0 .

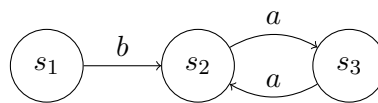
Dans ce cas, s_1 apparaît avant s_2 dans l'énumération et il existe une exécution de s_1 à s_2 que l'on peut construire avec une exécution de s_1 à s_3 (qui existe car s_3 marqué depuis s_1) et une exécution de s_3 à s_2 (qui existe car s_2 admet un cycle sur lui-même passant par s_3). D'après la question 9, s_0 admet un cycle sur lui-même : CONTRADICTION avec le fait que s_2 est choisi minimal pour cette propriété.

Comme l'ordre est celui spécifié dans la question 9, la fonction `verify` renvoie donc `true` si et seulement si il existe un sommet s' accessible depuis s_0 et hors de Φ admettant un cycle sur lui-même.

D'après la question 8, `verify` renvoie `true` si et seulement si $s_0 \not\models \mathbf{EA}\Phi$.

Question 12. Cela ne fonctionne plus car la propriété de la question 9 n'est plus vraie.

Pour le système suivant :



Avec $\Phi = \emptyset$.

On a $s_1 \not\models \mathbf{EA}\Phi$ et pourtant, si `outer_iter` traite les sommets dans l'ordre préfixe, on va d'abord tester s'il existe un cycle qui revient en s_1 : il n'y en a pas et lors de cette vérification tous les sommets sont marqués car accessibles et `Found` n'est pas levée alors que l'on a un cycle sur les deux autres sommets qui seront déjà marqués lors de la vérification qui ne lèvera pas `Found`.

3 Partie III : Programmation multi-threads en C

Question 13.

- Pour les deux threads, on s'intéresse à l'ordre des instructions des lignes 6 et 7 qui utilise la variable partagée, le reste étant des calculs indépendants sur le tableau partagé en lecture uniquement et des variables non partagées.

thread1 (m2 = 20)	thread2 (m2 = 40)	m = -1
ligne 6 (m3 = 20)		
ligne 7		m = 20
	ligne 6 (m3 = 40)	
	ligne 7	m = 40
Le programme affiche "m == 40".		

thread1 (m2 = 20)	thread2 (m2 = 40)	m = -1
ligne 6 (m3 = 20)		
	ligne 6 (m3 = 40)	
ligne 7		m = 20
	ligne 7	m = 40
Le programme affiche "m == 40".		

thread1 (m2 = 20)	thread2 (m2 = 40)	m = -1
ligne 6 (m3 = 20)		
	ligne 6 (m3 = 40)	
ligne 7		m = 40
	ligne 7	m = 20
Le programme affiche "m == 20".		

thread1 (m2 = 20)	thread2 (m2 = 40)	m = -1
	ligne 6 (m3 = 40)	
	ligne 7	m = 40
ligne 6 (m3 = 40)		
ligne 7		m = 40
Le programme affiche "m == 40".		

thread1 (m2 = 20)	thread2 (m2 = 40)	m = -1
	ligne 6 (m3 = 40)	
ligne 6 (m3 = 20)		
	ligne 7	m = 40
ligne 7		m = 20
Le programme affiche "m == 20".		

thread1 (m2 = 20)	thread2 (m2 = 40)	m = -1
	ligne 6 (m3 = 40)	
ligne 6 (m3 = 20)		
ligne 7		m = 20
	ligne 7	m = 40
Le programme affiche "m == 40".		

- b. Il faut protéger la variable partagée `m` par un mutex que l'on verrouille avant d'entrer dans la section critique des lignes 6 et 7 et que l'on déverrouille ensuite de sorte à ce que les threads soient en exclusion mutuelle.

Question 14.

```

1  int main() {
2      int* array = malloc(nb_threads*items_per_thread*sizeof(int));
3      assert(array != NULL);
4      for (int i = 0; i < nb_threads*items_per_thread; i++) {
5          array[i] = rand()%max_item;
6      }
7      /* attributs par défaut comme sur la figure 5 */
8      pthread_attr_t attr;
9      pthread_attr_init(&attr);
10     /* création */
11     pthread_t* t = malloc(nb_threads*sizeof(pthread_t));
12     assert(t != NULL);
13     for (int j = 0; j < nb_threads; j++) {
14         pthread_create(&(t[j]),&attr,local_max,&(array[j*items_per_thread]));
15     }
16     /* attente */
17     for (int j = 0; j < nb_threads; j++) {
18         pthread_join(t[j], NULL);
19     }
20     free(t);
21     printf("m == %d\n", m);
22     free(array); // non demandé ?
23     return 0;
24 }
```

4 Partie IV : Systèmes de transitions concurrents

Question 15. On a 4 transitions pour chaque thread qui doivent être dans l'ordre. Donc choisir parmi les 8 transitions lesquelles concernent le thread 1 déterminer entièrement l'exécution et cela donne des exécutions toutes différentes. Il y a donc $\binom{8}{4} = \frac{8 \times 7 \times 6 \times 5}{4 \times 3 \times 2} = 7 \times 2 \times 5 = 70$ exécutions différentes de longueur 8.

Question 16. En s_0 , les actions disponibles sont t_1^1 et t_2^1 , on a :

$$\delta(s_0, t_1^1) = (\mathbf{m}=-1, t_1 @ 1(\mathbf{m1}=10), t_2 @ 0)$$

$$\delta(s_0, t_2^1) = (\mathbf{m}=-1, t_1 @ 0, t_2 @ 1(\mathbf{m1}=30))$$

Puis :

$$\delta(\delta(s_0, t_2^1), t_1^1) = \delta(\delta(s_0, t_1^1), t_2^1) = (\mathbf{m}=-1, t_1 @ 1(\mathbf{m1}=10), t_2 @ 1(\mathbf{m1}=30))$$

Les deux actions sont indépendantes car s_0 est l'unique état où elles sont toutes les deux disponibles.

En s' , les actions disponibles sont t_1^1 et t_2^2 , on a :

$$\delta(s', t_1^1) = (\mathbf{m}=-1, t_1 @ 1(\mathbf{m1}=10), t_2 @ 1(\mathbf{m1}=30))$$

$$\delta(s', t_2^2) = (\mathbf{m}=-1, t_1 @ 0, t_2 @ 2(\mathbf{m1}=30, \mathbf{m2}=40))$$

Puis :

$$\delta(\delta(s', t_2^2), t_1^1) = \delta(\delta(s', t_1^1), t_2^2) = (\mathbf{m}=-1, t_1 @ 1(\mathbf{m1}=10), t_2 @ 2(\mathbf{m1}=30, \mathbf{m2}=40))$$

Les deux actions sont indépendantes car s' est l'unique état où elles sont toutes les deux disponibles.

En s_1 , les actions disponibles sont t_1^3 et t_2^3 , on a :

$$\delta(s_1, t_1^3) = (\mathbf{m}=-1, t_1 @ 3(\mathbf{m1}=10, \mathbf{m2}=20, \mathbf{m3}=20), t_2 @ 2(\mathbf{m1}=30, \mathbf{m2}=40))$$

$$\delta(s_1, t_2^3) = (\mathbf{m}=-1, t_1 @ 2(\mathbf{m1}=10, \mathbf{m2}=20), t_2 @ 3(\mathbf{m1}=30, \mathbf{m2}=40, \mathbf{m3}=40))$$

Puis :

$$\delta(\delta(s_1, t_2^3), t_1^3) = \delta(\delta(s_1, t_1^3), t_2^3) = (\mathbf{m}=-1, t_1 @ 3(\mathbf{m1}=10, \mathbf{m2}=20, \mathbf{m3}=20), t_2 @ 3(\mathbf{m1}=30, \mathbf{m2}=40, \mathbf{m3}=40))$$

Les deux actions sont indépendantes car s_1 est l'unique état où elles sont toutes les deux disponibles.

En s'' , les actions disponibles sont t_1^4 et t_2^3 , on a :

$$\delta(s'', t_1^4) = (\mathbf{m}=20, t_1 @ 4(\mathbf{m1}=10, \mathbf{m2}=20, \mathbf{m3}=20), t_2 @ 2(\mathbf{m1}=30, \mathbf{m2}=40))$$

$$\delta(s'', t_2^3) = (\mathbf{m}=-1, t_1 @ 3(\mathbf{m1}=10, \mathbf{m2}=20, \mathbf{m3}=20), t_2 @ 3(\mathbf{m1}=30, \mathbf{m2}=40, \mathbf{m3}=40))$$

Puis :

$$\delta(\delta(s'', t_2^3), t_1^4) = \delta(\delta(s'', t_1^4), t_2^3) = (\mathbf{m}=20, t_1 @ 4(\mathbf{m1}=10, \mathbf{m2}=20, \mathbf{m3}=20), t_2 @ 3(\mathbf{m1}=30, \mathbf{m2}=40, \mathbf{m3}=40))$$

Les deux actions sont indépendantes car s'' est l'unique état où elles sont toutes les deux disponibles.

Remarque : ce ne serait pas le cas avec les actions disponibles en $(\mathbf{m}=-1, t_1 @ 3(\mathbf{m1}=10, \mathbf{m2}=20, \mathbf{m3}=20), t_2 @ 3(\mathbf{m1}=30, \mathbf{m2}=40, \mathbf{m3}=40))$, les actions t_1^4 et t_2^4 ne sont pas indépendantes.

Question 17. Montrons que $\alpha \in \mathcal{D}(s_i)$ par récurrence sur $0 \leq i \leq n+1$.

Initialisation : $\alpha \in \mathcal{D}(s_0)$ car X ensemble persistant pour s_0 donc $\alpha \in X \subseteq \mathcal{D}(s_0)$.

Hérédité : D'après la propriété (A2), pour une telle exécution, on a $\alpha_i \leftrightarrow X$ donc en particulier $\alpha_i \leftrightarrow \alpha$.

Par hypothèse de récurrence, on a $\alpha \in \mathcal{D}(s_i)$, on veut montrer que $\alpha \in \mathcal{D}(s_{i+1})$.

Cela découle du premier point de l'indépendance : exécuter α_i ne change pas la disponibilité de α .

Question 18. En s_0 , $\{t_1^1\}$ et $\{t_2^1\}$ sont des ensembles persistants minimaux. On ne peut pas prendre $\emptyset$ car $\mathcal{D}(s_0) = \{t_1^1, t_2^1\}$ et on peut montrer $t_1^1 \leftrightarrow t_2^1$ pour tout i , ce qui garanti la deuxième condition et de même en inversant les rôles.

En s_1 , on a $\mathcal{D}(s_1) = \{t_1^3, t_2^3\}$. On ne peut pas prendre $\emptyset$. On ne peut pas prendre $\{t_1^3\}$ car $t_1^3 \not\leftrightarrow t_2^4$. On peut prendre $\{t_2^3\}$ en revanche car $t_2^3 \leftrightarrow t_1^4$ et $t_2^3 \leftrightarrow t_2^3$.

Question 19. On considère une exécution quelconque de s_0 à s_t :

$$\pi' : s'_0 = s_0 \xrightarrow{\beta_0} s'_1 \xrightarrow{\beta_1} \dots s'_{n-1} \xrightarrow{\beta_{n-1}} s'_n = s_t$$

On sait que $D(s_t) = \emptyset$.

Montrons par récurrence sur $0 \leq i < n$ que l'on peut construire une exécution π_i de longueur n tel que les i premières actions sont dans le $X(s)$ de l'état d'où elles sont appliquées.

Initialisation : pour $i = 0$, si $\beta_0 \in X(s_0)$, π' convient.

D'autre part, par (A1), on a $X(s_0) \neq \emptyset$ car $\beta_0 \in \mathcal{D}(s_0) \neq \emptyset$.

Supposons par l'absurde : $\forall 0 \leq j < n, \beta_j \notin X(s_0)$, dans ce cas par la question précédente, on obtient que tout $\alpha \in X(s_0)$ est tel que $\alpha \in \mathcal{D}(s_t) = \emptyset$. CONTRADICTION.

Soit $j_0 > 0$ tel que $\beta_{j_0} \in X(s_0)$. On a alors par (A2), $\beta_{j_0} \leftrightarrow \beta_j$ pour $j < j_0$.

Ainsi, on peut échanger β_{j_0} et β_{j_0-1} sans changer l'état s'_{j_0+1} et l'état s'_{j_0-1} et itérer jusqu'à obtenir une exécution avec β_{j_0} en position 0 avec le même s_0 de départ et le même s_t d'arrivée de longueur n .

On pose π_0 cette exécution et $\beta_{j_0} = \alpha_0$, s_1 le deuxième état de ce calcul.

Hérédité : A partir du calcul π_i obtenu par hypothèse de récurrence, on obtient π_{i+1} en appliquant le raisonnement de l'initialisation sur le calcul de s_{i+1} à s_t .

Cette récurrence est finie et on a π_n le calcul π attendu.

Question 20.

- a. Pour $u = \alpha\beta\gamma$, $\beta \not\leftrightarrow \alpha$ impose $\alpha <_u \beta$ et $\beta \not\leftrightarrow \gamma$ impose $\beta <_u \gamma$, puis la transitivité de $<_u$ donne $\alpha <_u \gamma$: on a un ordre total.
Pour $w = \gamma\alpha\beta$ et $v = \alpha\gamma\beta$, $\beta \not\leftrightarrow \alpha$ impose $\alpha <_v \beta$ et $\alpha <_w \beta$ et $\beta \not\leftrightarrow \gamma$ impose $\gamma <_v \beta$ et $\gamma <_w \beta$, les ordres partiels ainsi définis sont bien transitifs : aucune relation entre α et γ qui sont incomparables pour ces ordres.
- b. Pour $u = \alpha\alpha'\beta\beta'$ et $v = \alpha\beta\alpha'\beta'$, $\alpha \not\leftrightarrow \beta$ impose $\alpha <_u \beta$ et $\alpha <_v \beta$ et $\alpha' \not\leftrightarrow \beta'$ impose $\alpha' <_u \beta'$ et $\alpha' <_v \beta'$, les ordres partiels ainsi définis sont bien transitifs : aucune relation entre les autres éléments qui sont incomparables pour ces ordres.
Pour $v = \beta\alpha\alpha'\beta'$, $\alpha \not\leftrightarrow \beta$ impose $\beta <_w \alpha$ et $\alpha' \not\leftrightarrow \beta'$ impose $\alpha' <_w \beta'$, l'ordre partiel ainsi défini est bien transitifs : aucune relation entre les autres éléments qui sont incomparables pour cet ordre.

Question 21. On suppose que $<_u$ et $<_v$ coïncident.

Cela signifie que pour tout couple α, β de lettres de A tel que $\alpha \not\leftrightarrow \beta$, α et β apparaissent dans le même ordre dans u et v .

Si $u = v$, on a directement $u \sim v$.

Sinon, on peut poser i_0 le plus petit indice tel que $u_{i_0} \neq v_{i_0}$ et j_0 tel que $v_{j_0} = u_{i_0}$.

Par coïncidence de l'ordre, on a $u_{i_0} \leftrightarrow v_j$ pour $i_0 \leq j < j_0$ car ces lettres apparaissent avant la lettre u_{i_0} dans v et après dans u .

On a donc $v_1 \dots v_{i_0-1} v_{j_0} v_{i_0} \dots v_{j_0-1} v_{j_0+1} \dots v_{n-1} \sim v$ par échange de proche en proche de lettres adjacentes indépendantes.

Le nouveau mot a un préfixe commun strictement plus long avec u et l'ordre est conservé.

On peut itérer jusqu'à retomber sur u (c'est une récurrence).

On obtient $u \sim v$ car $\sim$ est transitive.

Pour s'intéresser à la réciproque, on revient à la définition de $\sim$.

Soit α, β tels que $\alpha \leftrightarrow \beta$ et x, y des mots. On a $x\alpha\beta y \sim x\beta\alpha y$.

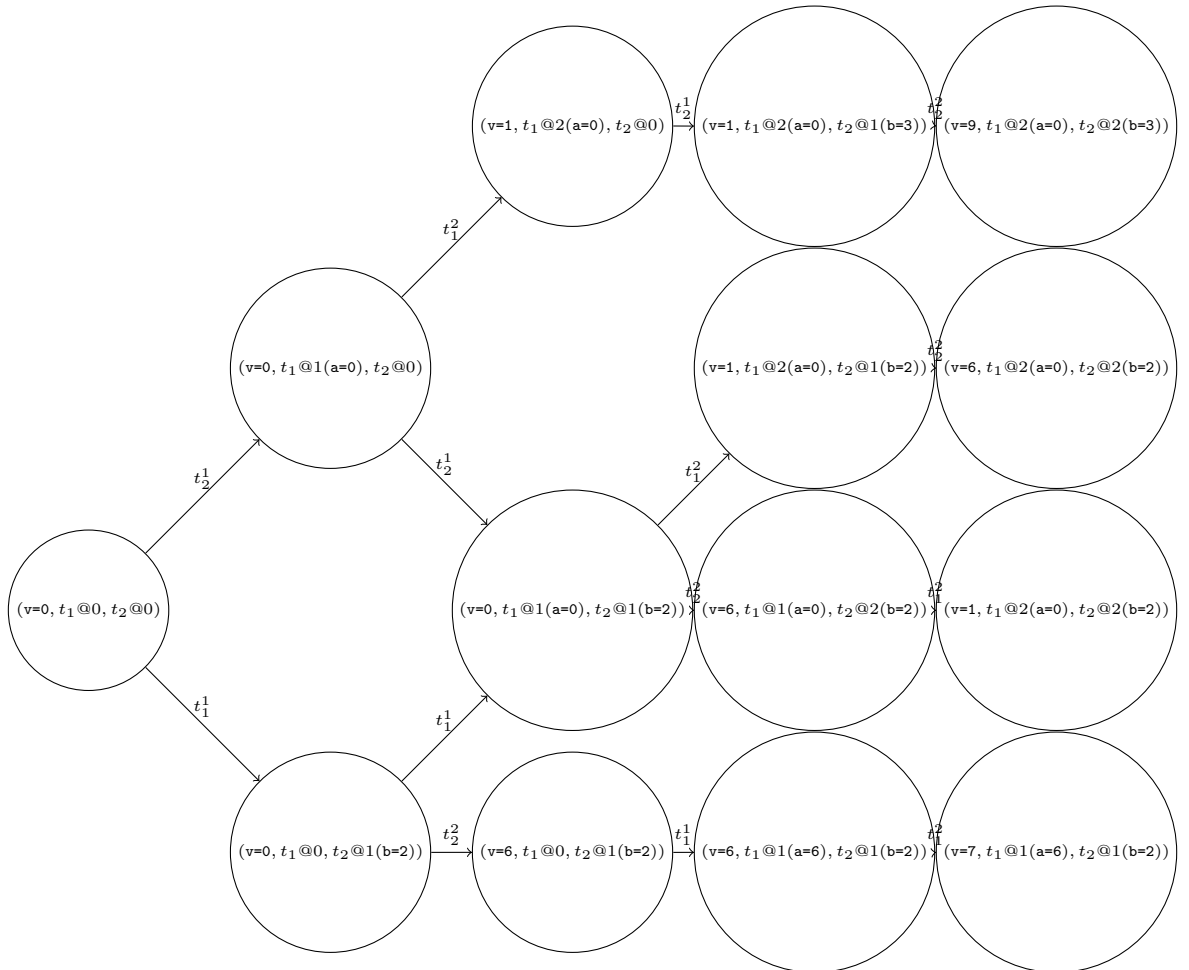
Les couples de lettres non indépendantes dans $x\alpha\beta y$ et $x\beta\alpha y$ sont dans le même ordre car seul l'ordre des lettres α et β est modifié. Ainsi, l'ordre induit est le même.

La relation $\sim$ étant la plus petite relation d'équivalence vérifiant que pour tout α, β tels que $\alpha \leftrightarrow \beta$ et x, y des mots, on a $x\alpha\beta y \sim x\beta\alpha y$, $u \sim v$ vient d'une suite finie d'échanges sur des lettres adjacentes indépendantes.

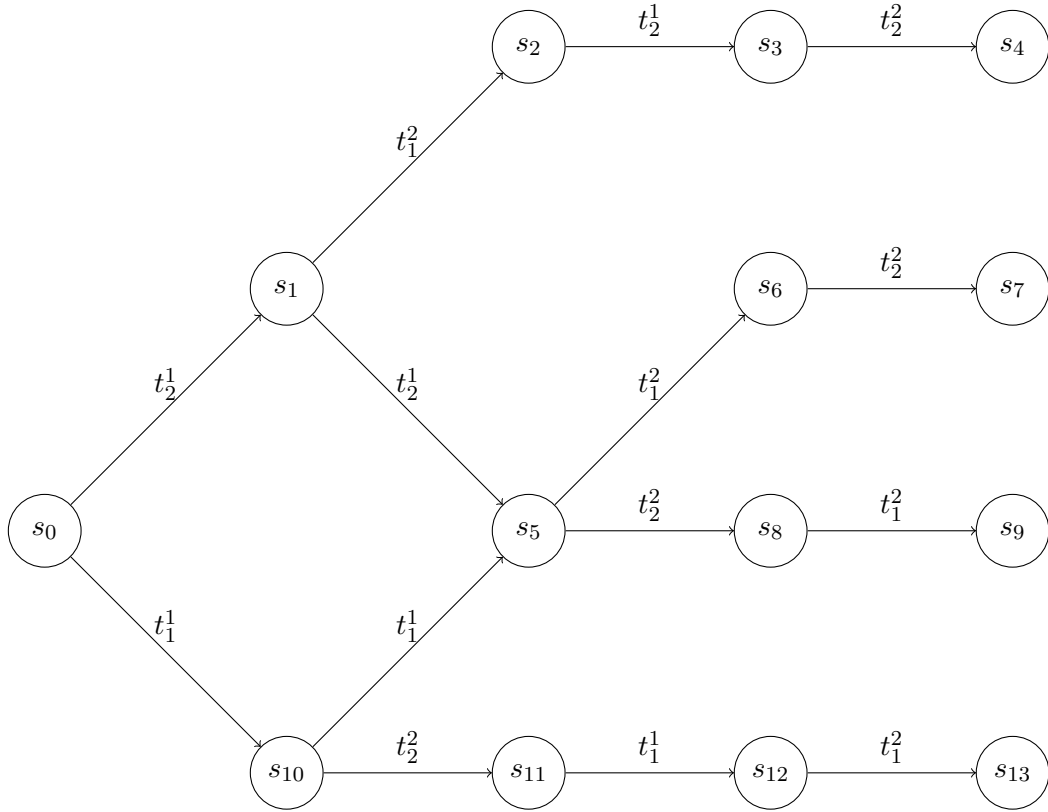
Par transitivité de l'égalité des ordres, on obtient que $<_u$ et $<_v$ sont égaux si $u \sim v$ en appliquant la propriété élémentaire justifiée ci-dessus pour chaque échange.

Question 22.

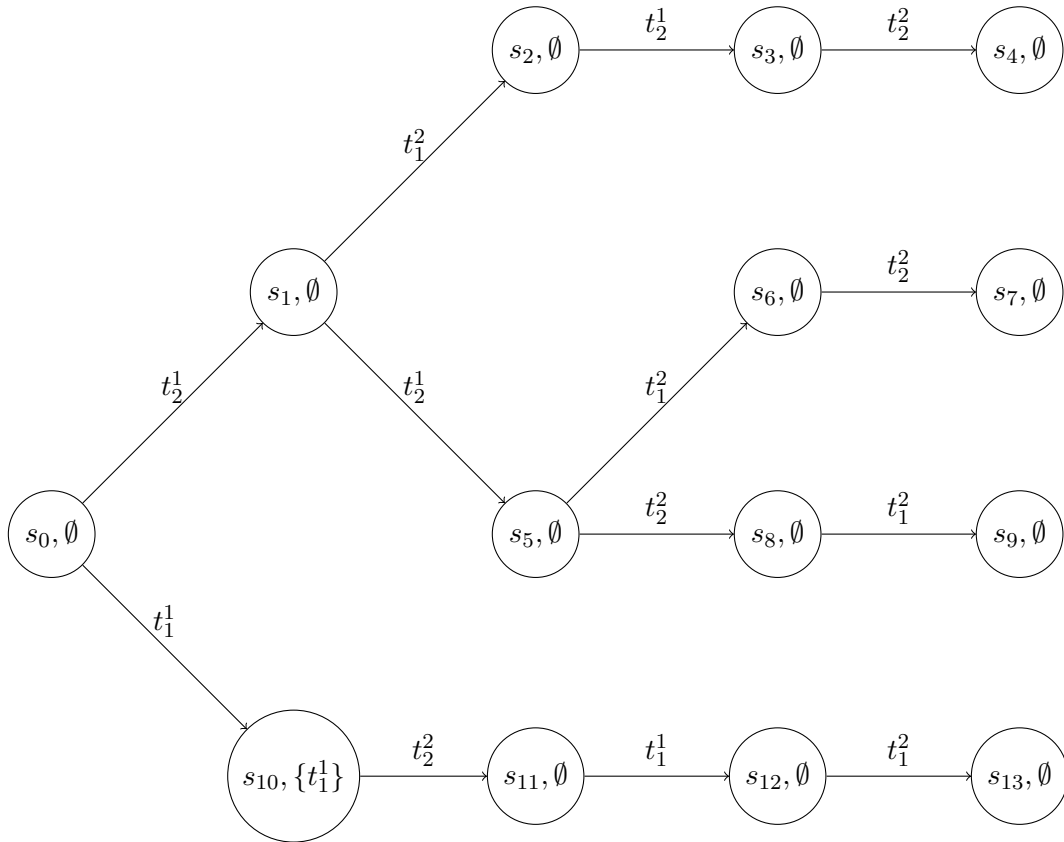
a.



Et sans les contenus pour plus de lisibilité et car ils ne sont pas attendus :



b. On a $t_1^1 \leftrightarrow t_2^1$ mais aucun autre couple d'actions indépendantes et disponibles simultanément donc on obtient :



Question 23. Supposons $u = u_1\beta u_2\alpha u_3$ avec u_1, u_2, u_3 dans $A^\neq$ et $\alpha < \beta$ et $\alpha \leftrightarrow \beta u_2$. Par échange de lettre adjacentes indépendantes successives, on obtient $u \sim u_1\alpha\beta u_2 u_3$ car $\sim$ transitives.

Or $u_1\alpha\beta u_2u_3 < u$ car $\alpha < \beta$ donc $u \neq u^<$.

Par contraposée, si $u = u^<$, alors u ne se décompose pas de cette façon.

Supposons $u \neq u^<$.

Il existe donc v tel que $u \sim v$ et $v < u$.

Soit i le premier rang tel que $u_i \neq v_i$.

$<_u$ et $<_v$ qui coïncident d'après 21.

On pose α la lettre en position i de v et β la lettre en position i de u , on a $u = u_1\beta u_2\alpha u_3$ avec u_1, u_2, u_3 dans $A^\neq$ et $\alpha < \beta$ et $v = u_1\alpha v_2\beta v_3$ avec v_2, v_3 dans $A^\neq$.

Le fait que $<_u$ et $<_v$ coïncident assure que $\alpha \leftrightarrow u_2\beta$ car les lettres que $u_2\beta$ sont après α dans v et avant α dans u .

Ainsi, on obtient la décomposition attendue.

On déduit la réciproque de la propriété demandée par contraposée.

Question 24.

- a. Soit Z, Z' des sous-ensembles de A tels que $(s, Z) \xrightarrow{u} (s', Z)$.

On note $u = u_1 \dots u_n$ avec $n = |u|$ et l'exécution :

$$(s, Z) = (s_0, Z_0) \xrightarrow{u_1} (s_1, Z_1) \xrightarrow{u_2} \dots \xrightarrow{u_n} (s_n, Z_n) = (s', Z')$$

Pour $1 \leq i \leq n$, on a $u_i \in \mathcal{D}(s_{i-1}) \setminus Z_{i-1}$ et

$$Z_i = \{\beta \in Z_{i-1} \mid u_i \leftrightarrow \beta\} \cup \{\beta \in \mathcal{D}(s_{i-1}) \mid u_i \leftrightarrow \beta, \beta < u_i\}.$$

Supposons par l'absurde qu'il existe i, j tels que $j < i$, $u_i < u_j$ et $u_i \leftrightarrow u_j \dots u_{i-1}$ (c'est la décomposition envisagée dans la question précédente).

Montrons par récurrence que $u_i \notin Z_{i-k}$ et $u_i \in \mathcal{D}(s_{i-k})$ pour $1 \leq k \leq i - j$.

Initialisation : $u_i \notin Z_{i-1}$ par définition de $\mathcal{D}^<$ et $u_i \in \mathcal{D}(s_{i-k})$.

Hérédité : On suppose $u_i \notin Z_{i-k}$ et $u_i \in \mathcal{D}(s_{i-k})$, en particulier $u_i \notin \{\beta \in Z_{i-k-1} \mid u_{i-k} \leftrightarrow \beta\}$, or $u_{i-k} \leftrightarrow u_i$ car $j \leq i - k \leq i - 1$ donc $u_i \notin Z_{i-k-1}$.

De plus $\delta(s_{i-k-1}, u_{i-k}) = s_{i-k}$ et $u_{i-k} \leftrightarrow u_i$ donc $u_i \in \mathcal{D}(s_{i-k-1})$.

Enfin, on a $u_i \notin Z_j$ et $u_i \in \mathcal{D}(s_{j-1})$ en l'appliquant une dernière fois.

Or $Z_j = \{\beta \in Z_{j-1} \mid u_j \leftrightarrow \beta\} \cup \{\beta \in \mathcal{D}(s_{j-1}) \mid u_j \leftrightarrow \beta, \beta < u_j\}$ et $u_i \in \mathcal{D}(s_{j-1}), u_j \leftrightarrow u_i, u_i < u_j$.

On obtient donc $u_i \in Z_j$ et $u_i \notin Z_j$: CONTRADICTION.

D'après la question précédente, comme u n'admet pas de "mauvaise" décomposition, on a $u = u^<$.

- b. On remarque que pour w un mot de $A^\neq$ se décomposant en $w = x\alpha\beta y$ avec $\alpha \leftrightarrow \beta$, on a $s \xrightarrow{x\alpha\beta y} s'$ si et seulement si $s \xrightarrow{x\beta\alpha y} s'$ en vertu de l'indépendance.

Grâce à cette petite propriété et le fait que $u \sim u^<$ et $\sim$ plus petite relation d'équivalence stable par échange de lettres indépendantes adjacentes, on obtient $s \xrightarrow{u} s'$ si et seulement si $s \xrightarrow{u^<} s'$.

$$s = s_0 \xrightarrow{u_0} s_1 \dots \xrightarrow{u_{n-1}} s' = s_n \text{ avec } n = |u|, u^< = u_0 \dots u_{n-1}$$

Montrons que cette dernière exécution est alors conservée dans $T^<$ depuis $(s, \emptyset)$.

Initialement, depuis $(s, \emptyset = Z_0)$, on peut appliquer u_0 et on atteint (s_1, Z_1) avec $Z_1 = \{\beta \in \mathcal{D}(s_0) \mid u_0 \leftrightarrow \beta, \beta < \alpha\}$.

On poursuit ainsi le calcul.

Supposons par l'absurde qu'on atteigne (s_i, Z_i) tel que $u_i \in Z_i$, on aurait alors $u_i \leftrightarrow u_{i-1}$ donc $u_{i-1} < u_i$ par minimalité de $u^<$ pour l'ordre lexicographique donc $u_i \in Z_{i-1}$ nécessairement. Puis, en itérant $u_i \in Z_0$ sinon on obtiendrait une décomposition comme

dans la question précédente.
CONTRADICTION car $Z_0 = \emptyset$.
Donc l'exécution est bien conservée.