ECOLE POLYTECHNIQUE
ECOLES NORMALES SUPERIEURES

CONCOURS D'ADMISSION 2026

JEUDI 16 AVRIL 2026
08h00 - 12h00

FILIERE MPI - Epreuve n° 7

INFORMATIQUE C

Durée : 4 heures

L'utilisation des calculatrices n'est pas autorisée pour cette épreuve

Vérification de systèmes concurrents

Le sujet comporte 17 pages et 24 questions.

Vue d'ensemble du sujet.

Les systèmes de transitions constituent un cadre classique pour la modélisation et l'analyse des systèmes en informatique. On peut les voir comme une variante des automates finis. Cependant, les automates sont utilisés pour lire des mots finis (et les accepter ou non) tandis que les systèmes de transitions sont utilisés pour produire des exécutions infinies, qu'on va ensuite analyser.

Dans les parties I et II de ce sujet, on s'intéresse à la vérification algorithmique de certaines propriétés des exécutions infinies d'un système de transitions. En particulier, on verra en partie II l'algorithme des parcours en profondeur imbriqués pour la vérification des propriétés d'ultime invariance. Ces deux parties comportent des questions de programmation OCaml.

Dans les parties III et IV, on s'intéresse plus particulièrement aux systèmes concurrents. La partie III porte sur la programmation concurrente en langage C, rappelant l'utilisation de la bibliothèque `pthread`, avec des questions d'analyse et d'écriture de programmes. On voit en partie IV comment de tels programmes peuvent être modélisés par des systèmes de transitions, et l'on introduit des notions classiques de théorie de la concurrence permettant d'analyser ces systèmes en évitant autant que possible l'explosion combinatoire induite par les entrelacements d'instructions concurrentes.

Les définitions de la partie I sont utiles dans tout le sujet. La partie II est indépendante du reste du sujet. La partie III fournit des rappels utiles et un exemple de programme utilisé en partie IV.

Rappels

Si E est un ensemble, on note 2^E l'ensemble de ses **parties**.

Si E et F sont deux ensembles, on rappelle qu'une **relation** $\mathcal{R}$ entre E et F est une partie de $E \times F$, que l'on appelle parfois le *graphe* de la relation. Deux éléments $x \in E$ et $y \in F$ sont en relation pour $\mathcal{R}$ quand $(x, y) \in \mathcal{R}$; on note alors $x \mathcal{R} y$. Deux relations entre les mêmes ensembles *coïncident* lorsque leurs graphes sont égaux. On parlera parfois de *plus petite relation* entre E et F vérifiant certaines propriétés. Cette formulation est à comprendre au sens de l'inclusion des graphes : une relation $\mathcal{R}$ est plus petite que $\mathcal{R}'$ quand le graphe de la seconde contient celui de la première, *i.e.* $\mathcal{R} \subseteq \mathcal{R}'$.

On rappelle que Σ^* est l'ensemble des **mots finis** sur Σ. La longueur d'un mot $u \in \Sigma^*$ sera notée $|u|$: c'est le nombre de lettres dans u. On note ϵ l'unique mot de longueur nulle. Pour $w \in \Sigma^*$ et $i < |w|$, on note w_i la lettre d'indice i dans w, en numérotant à partir de 0 : on a donc $w = w_0 \dots w_{|w|-1}$.

On note Σ^ω l'ensemble des **mots infinis** sur Σ, défini comme l'ensemble des suites infinies $(a_i)_{i\in\mathbb{N}}$ avec $a_i \in \Sigma$ pour tout $i \in \mathbb{N}$. Si w est un mot infini et $i \in \mathbb{N}$, on note w_i la lettre d'indice i dans w, c'est-à-dire a_i si $w = (a_i)_{i\in\mathbb{N}}$. La concaténation d'un mot fini $u \in \Sigma^*$ et d'un mot infini $v \in \Sigma^\omega$ sera simplement notée uv, et définie précisément comme la suite $(a_i)_{i\in\mathbb{N}}$ telle que $a_i = u_i$ pour $i < |u|$ et $a_{j+|u|} = v_j$ pour $j \in \mathbb{N}$. On écrira généralement un mot infini $w \in \Sigma^\omega$ comme $w_0 w_1 \dots$

On dit que $u \in \Sigma^*$ est un **préfixe** du mot v (fini ou infini) quand v s'écrit uw pour un certain mot w. Le préfixe u est dit **strict** quand $u \neq v$.

Pour un mot fini $u \in \Sigma^*$ de longueur $n \in \mathbb{N}$ et une lettre $a \in \Sigma$, on note

$$|u|_a \stackrel{\text{def}}{=} |\{i < n \mid u_i = a\}|$$

le nombre de positions dans u où la lettre a apparaît. On étend naturellement cette notion aux mots infinis, et l'on aura $|u|_a = +\infty$ si le mot contient la lettre a à une infinité de positions.

Dans ce sujet, quand on parle d'un mot sans plus de précision, il peut être fini ou infini. À l'inverse, on précisera toujours quand un mot est supposé fini (resp. infini).

Langage C

Dans ce sujet, on utilisera le mot anglais *thread* pour parler de *fil d'exécution* dans le cadre de la programmation concurrente, aussi appelée programmation *multi-threads*.

On considèrera toujours que tous les en-têtes nécessaires à un programme C sont implicitement inclus. Cela concerne notamment les en-têtes usuels tels que `<assert.h>`, `<stdio.h>` ou `<stdlib.h>`, mais aussi l'en-tête `<pthread.h>`. Cette convention vaut pour les programmes présentés dans le sujet, mais s'applique aussi aux programmes que le candidat devra écrire : il n'est pas demandé d'inclure explicitement les en-têtes.

On peut définir et initialiser un tableau *statique* de n entiers grâce à l'instruction `int tableau[`n`] = {`$\mathtt{e}_0$`,`$\mathtt{e}_1$`,...,`$\mathtt{e}_{n-1}$`};` où n est une constante littérale entière et les $\mathtt{e}_i$ pour $i \in \{0, \dots, n-1\}$ sont des expressions de type `int`.

Langage OCaml

Dans le reste du sujet, on pourra utiliser les fonctions suivantes de la bibliothèque standard OCaml pour les listes. Les hypothèses de complexité données, légèrement simplifiées par rapport à la réalité, pourront être utilisées pour des listes sur des types pour lesquels l'égalité est en temps constant — on supposera que c'est le cas pour les états, actions, et paires de ces objets, manipulés dans le sujet.

- Si `lst1` et `lst2` sont deux listes, `lst1 @ lst2` est leur concaténation. Elle se calcule en temps linéaire en la longueur de `lst1`. L'ajout `e::lst` d'un élément `e` en tête de la liste `lst` se fait en temps constant.
- `List.mem x lst` renvoie `true` s'il existe un élément `e` de la liste `lst` tel que `x = e`, et `false` sinon. Elle s'exécute en temps linéaire en la longueur de `lst`.
- `List.assoc k` $[(k_1, v_1); \ldots; (k_n, v_n)]$ renvoie le premier v_i tel que `k =` k_i s'il existe, et lève l'exception `Not_found` sinon. Elle s'exécute en temps linéaire en la longueur de la liste.
- `List.iter f` $[e_1; \ldots; e_n]$ exécute `f` e_i pour i = 1, 2, ..., n, et renvoie `()`. Son temps d'exécution est la somme des temps d'exécution des `f` e_i.
- `List.map f` $[e_1; \ldots; e_n]$ renvoie $[f\ e_1; \ldots; f\ e_n]$. Son temps d'exécution est la somme des temps d'exécution des `f` e_i.
- `List.filter f lst` renvoie la liste des éléments `e` de `lst` tels que `f e = true`, dans l'ordre. Son temps d'exécution est la somme des temps d'exécution des `f e`.
- `List.fold_left f` a_0 $[e_1; \ldots; e_n]$ renvoie a_n où a_{k+1} `= f` a_k e_{k+1} pour tout $0 \leq k < n$. Son temps d'exécution est la somme des temps d'exécution des `f` a_k e_{k+1}.

Pour les tableaux, on pourra utiliser les fonctions suivantes :

- `Array.init n f` renvoie le tableau `[|f 0; ...; f (n-1)|]` pour `n` positif. Son temps d'exécution est la somme des temps des `f i` pour `i` $= 0, 1, \ldots, n-1$.
- `Array.make n v` renvoie le tableau `[|v; ...; v|]` de taille `n` dont chaque case contient la même valeur `v`, pour `n` positif. Son temps d'exécution est linéaire en `n`.

Partie I : Systèmes de transitions

Un **système de transitions** $T = (S, A, \mathcal{D}, \delta)$ est composé de :
- un ensemble fini d'*états* S ;
- un ensemble fini d'*actions* A ;
- une application $\mathcal{D} : S \to 2^A$ indiquant pour chaque état s l'ensemble $\mathcal{D}(s)$ des actions *disponibles* en s ;
- une fonction partielle $\delta : S \times A \to S$ de domaine $\{(s, \alpha) \in S \times A \mid \alpha \in \mathcal{D}(s)\}$ indiquant pour chaque état s et $\alpha \in \mathcal{D}(s)$ le résultat $\delta(s, \alpha)$ de la transition α depuis s.

On écrira $s \xrightarrow{\alpha} s'$ pour $\delta(s, \alpha) = s'$. Pour $u = u_0 \dots u_n \in A^*$, on écrira $s \xrightarrow{u} s'$ pour signifier qu'il existe des états s_i tels que $s \xrightarrow{u_0} s_0 \xrightarrow{u_1} \dots \xrightarrow{u_n} s'$. Un état s est dit **terminal** si $\mathcal{D}(s) = \emptyset$.

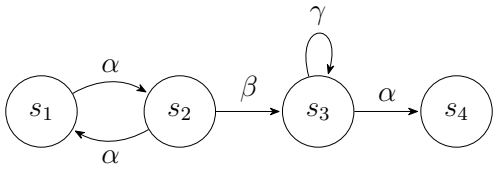


FIGURE 1 – Représentation graphique d'un système de transitions simple.

On pourra représenter graphiquement un système de transitions comme on le fait pour les automates finis. Ainsi, la figure 1 représente le système $T = (S, A, \mathcal{D}, \delta)$ avec $S = \{s_1, s_2, s_3, s_4\}$ et $A = \{\alpha, \beta, \gamma\}$, où l'on a

$$\mathcal{D}(s_1) = \{\alpha\} \qquad \mathcal{D}(s_2) = \{\alpha, \beta\} \qquad \mathcal{D}(s_3) = \{\alpha, \gamma\} \qquad \mathcal{D}(s_4) = \emptyset$$

et δ est donnée par les arcs entre états, avec notamment $\delta(s_1, \alpha) = s_2$ et $\delta(s_3, \gamma) = s_3$. Dans ce système, s_4 est le seul état terminal, *i.e.* le seul état s tel que $\mathcal{D}(s) = \emptyset$.

Une **exécution infinie** dans T est donnée par un état de départ $s_0 \in S$ et une suite d'actions $(\alpha_i)_{i \in \mathbb{N}}$ telles que l'état successeur $s_{i+1} = \delta(s_i, \alpha_i)$ est défini pour tout $i \in \mathbb{N}$. Quand $\pi = (s_0, (\alpha_i)_{i \in \mathbb{N}})$ définit une exécution infinie, on notera simplement :

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

La **trace** d'une telle exécution infinie est la suite des états de l'exécution :

$$\text{trace}(\pi) = (s_i)_{i \in \mathbb{N}}$$

Sur l'exemple de la figure 1, on a des exécutions infinies à partir de s_1 avec pour trace $s_1 s_2 s_1 s_2 \dots$ mais aussi $s_1 s_2 s_1 s_2 s_3 s_3 \dots$ Il n'y a qu'une exécution infinie à partir de s_3, dont la trace est $s_3 s_3 s_3 \dots$

On définit de façon analogue les **exécutions finies**, données par un état de départ $s_0 \in S$, une longueur $n \in \mathbb{N}$, et une suite d'actions $(\alpha_i)_{i<n}$ telle que $s_{i+1} = \delta(s_i, \alpha_i)$ est défini pour tout $i < n$. Pour indiquer que $\pi = (s_0, (\alpha_i)_{i<n})$ est une exécution finie, on notera :

$$\pi : s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_{n-1}} s_n$$

Dans ce cas-là, on dira qu'il existe une exécution de s_0 à s_n, ou que s_n est accessible à partir de s_0. Plus généralement, on dira qu'un ensemble d'états E est accessible à partir de s_0 s'il existe un état $s \in E$ accessible à partir de s_0.

Attention ! *Un système de transitions sur un ensemble d'actions A ressemble beaucoup à un automate fini sur l'alphabet A, mais il ne faut pas confondre les deux notions. On remarquera notamment qu'un système de transitions n'a pas d'état initial distingué : n'importe quel état du système peut servir de point de départ pour des exécutions. De plus, la notion d'état terminal dans un système de transitions ne doit pas être confondue avec la notion d'état final dans un automate : elle ne témoigne pas de l'acceptation d'un mot, mais simplement de l'impossibilité d'effectuer toute transition.*

Dans ce sujet, on utilise à la fois les systèmes de transitions et les automates finis, mais dans des rôles différents. Pour bien les distinguer, on réserve l'usage de la lettre s pour les états des systèmes de transitions, et la lettre q pour les états des automates.

Une **propriété de trace** pour un système de transitions $T = (S, A, \mathcal{D}, \delta)$ est un ensemble $P \subseteq S^\omega$ de mots infinis sur S. On dit qu'un état s **satisfait** la propriété P quand, pour toute exécution infinie π à partir de s, on a $\text{trace}(\pi) \in P$. On écrit alors $s \models P$.

On remarquera qu'un état s à partir duquel aucune exécution infinie n'est possible satisfait trivialement toute propriété P. Sous cette condition, on a même $s \models P$ et $s \models S^\omega \setminus P$, ce qui ne peut arriver dès lors que s admet au moins une exécution infinie.

On illustre la notion de satisfaction sur l'exemple de la figure 1 avec P l'ensemble des mots $w \in S^\omega$ tels que $w_i = s_2$ pour au moins une position $i \in \mathbb{N}$. On a alors $s_1 \models P$ et $s_2 \models P$ car toutes les exécutions infinies à partir de ces états ont s_2 dans leur trace, mais $s_3 \not\models P$. On a trivialement $s_4 \models P$ car il n'existe pas d'exécution infinie à partir de s_4.

Un **invariant** de T est un ensemble d'états $I \subseteq S$ tel que, pour tous $s \in I$ et $\alpha \in \mathcal{D}(s)$, on a $\delta(s, \alpha) \in I$. On pourra remarquer que l'ensemble vide et l'ensemble de tous les états sont des invariants triviaux, pour tout T. Sur l'exemple de la figure 1, les seuls invariants non-triviaux sont $\{s_3, s_4\}$ et $\{s_4\}$.

Question 1. Soit $I \subseteq S$. On considère la propriété $P = I^\omega$, *i.e.* l'ensemble des mots infinis sur I. Montrer que les conditions suivantes sont équivalentes quand T ne contient aucun état terminal :

(a) L'ensemble I est un invariant.

(b) Pour tout $s \in I$, on a $s \models P$.

Une propriété P est une **propriété de sûreté** quand, pour tout mot $w \in S^\omega \setminus P$, il existe un préfixe $\hat{w} \in \Sigma^*$ de w tel que pour tout $w' \in S^\omega$ on a encore $\hat{w}w' \notin P$.

Par exemple, si l'on fixe $s \in S$, alors $P_1 = \{w \in S^\omega \mid |w|_s = 0\}$ est une propriété de sûreté : tout $w \notin P_1$ contient s à une certaine position i, et si l'on prend pour $\hat{w}$ le préfixe de w de longueur $i+1$, on a bien $\hat{w}w' \notin P_1$ pour tout $w' \in S^\omega$. En revanche, $P_2 = \{w \in S^\omega \mid |w|_s = +\infty\}$ n'est pas une propriété de sûreté dès lors qu'il existe un état $s' \neq s$: en effet, le mot $w = s's's' \ldots$ n'est pas dans P_2, mais pour tout préfixe $\hat{w}$ de w, on a $\hat{w}sss \ldots \in P_2$.

Question 2. On suppose dans cette question que S contient au moins deux états distincts. Parmi les propriétés suivantes, lesquelles sont des propriétés de sûreté ?

1. $P_1 = \{w \in S^\omega \mid |w|_s = 0$ pour au moins un $s \in S\}$
2. $P_2 = \{w \in S^\omega \mid |w|_s = +\infty$ pour tout $s \in S\}$
3. $P_3 = \{w \in S^\omega \mid |w|_s < +\infty$ pour au moins un $s \in S\}$

Soit P une propriété de sûreté. On dit qu'un mot fini $u \in S^*$ est un **préfixe interdit** pour P quand, pour tout $v \in S^\omega$, on a $uv \notin P$. Le préfixe interdit u est **minimal** si aucun de ses préfixes stricts n'est lui-même préfixe interdit. On définit $\hat{P}$ comme l'ensemble des préfixes interdits minimaux pour P. On dit que la propriété P est **régulière** quand $\hat{P}$ est un langage régulier, au sens usuel des langages réguliers de mots finis.

Question 3. On suppose pour cette question que le système T n'a que deux états, et l'on pose $S = \{a, b\}$. Parmi les propriétés de sûreté suivantes, lesquelles sont régulières ? Justifier vos réponses, en donnant notamment un automate reconnaissant $\hat{P}_i$ quand P_i est régulière.

1. $P_1 = \{w \in S^\omega \mid |u|_a = 0$ pour tout $u \in \Sigma^*$ préfixe de $w\}$
2. $P_2 = \{w \in S^\omega \mid ||u|_a - |u|_b| \leq 1$ pour tout $u \in \Sigma^*$ préfixe de $w\}$
3. $P_3 = \{w \in S^\omega \mid |u|_a \leq |u|_b$ pour tout $u \in \Sigma^*$ préfixe de $w\}$
4. $P_4 = \{w \in S^\omega \mid |u|_a \leq |u|_b \leq |u|_a + 1$ pour tout $u \in \Sigma^*$ préfixe de $w\}$

Question 4. Soit P une propriété de sûreté. Montrer que l'ensemble de ses préfixes interdits minimaux est régulier si et seulement si l'ensemble de ses préfixes interdits est régulier.

Afin de vérifier algorithmiquement des propriétés de systèmes de transitions, on introduit quelques structures de données en OCaml. Par simplicité, on représentera les états des systèmes de transitions par des entiers dans un intervalle fini : on pose `type state = int`. On suppose disposer d'un type `action` pour représenter les actions ; il est inutile de le préciser.

Un système de transitions sera décrit par un enregistrement du type suivant :

```
type trans_sys = {
  size : int;                              (* états = { 0,1,...,size-1 } *)
  next : state -> (action * state) list;              (* transitions *)
}
```

Plus précisément, une valeur `t` de ce type représente le système de transitions dont :

- les états sont les entiers positifs (ou nuls) strictement inférieurs à `t.size` ;
- les transitions à partir d'un état `s` sont données par `t.next s`, qui est une liste d'associations sans répétition d'une même action : on a $\texttt{s} \xrightarrow{\texttt{a}} \texttt{s'}$ pour tout `(a,s')` dans `t.next s`.

Les actions disponibles $\mathcal{D}(\texttt{s})$ sont données implicitement via `t.next s`. On supposera que la fonction `t.next` s'exécute en temps constant.

Question 5. Étant donné un système de transitions T, un état s_0 et un ensemble d'états cible C, on souhaite vérifier algorithmiquement l'accessibilité de C à partir de s_0. Implémenter en OCaml une fonction

```
val reachable : trans_sys -> state -> (state -> bool) -> bool
```

telle que `reachable t s0 target = true` quand il existe un état `s` de `t` accessible à partir de `s0` et tel que `target s = true`. Votre fonction devra terminer sur toute entrée, en supposant seulement que `s0` est un état valide de `t`. Justifier brièvement sa correction, et évaluer sa complexité en temps en fonction des nombres d'états et d'actions de `t`.

On démontrerait aisément que les mots d'une propriété de sûreté P sont exactement ceux dont aucun préfixe n'est dans $\hat{P}$, et l'on admet ce résultat dans la suite :

$$P = \{w \in S^\omega \mid u \notin \hat{P} \text{ pour tout préfixe } u \text{ de } w\}$$

Question 6. Soit $T = (S, A, \mathcal{D}, \delta)$ un système de transitions *sans état terminal*, et $P \subseteq S^\omega$ une propriété de sûreté dont les préfixes interdits minimaux sont reconnus par un automate fini $\mathcal{A}$ déterministe et complet. On souhaite ramener le problème de la satisfaction de P à un problème d'inaccessibilité dans un système de transitions construit à partir de T et $\mathcal{A}$. Pour cela, construire un système de transitions $T_\mathcal{A} = (S', A, \mathcal{D}', \delta')$, une application $\iota : S \to S'$ et un sous-ensemble $C \subseteq S'$ tels que, pour tout $s \in S$, on a $s \not\models P$ si et seulement si C est accessible à partir de $\iota(s)$ dans $T_\mathcal{A}$. On justifiera cette équivalence.

L'objectif de cette question est de donner une construction qui mènerait immédiatemment à un algorithme pour la satisfaction, mais on ne demande ici qu'une description mathématique de cette construction.

Partie II : Ultime invariance

Soit $T = (S, A, \mathcal{D}, \delta)$ un système de transitions, et $\Phi \subseteq S$. On définit la propriété de trace $\mathsf{EA}\Phi$ comme suit :

$$\mathsf{EA}\Phi \stackrel{\text{def}}{=} \{\ w \in S^\omega \ \mid\ \exists i \in \mathbb{N}, \forall j \geq i,\ w_j \in \Phi\ \}$$

La notation $\mathsf{EA}\Phi$ vient de l'anglais *eventually always* Φ, qu'on peut traduire littéralement comme *au bout d'un moment, on est tout le temps dans* Φ. De façon plus concise, on appellera $\mathsf{EA}\Phi$ la propriété d'*ultime invariance* de Φ.

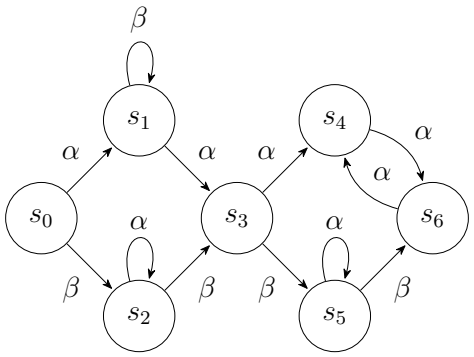


FIGURE 2 – Exemple de système sur $A = \{\alpha, \beta\}$.

Question 7. On suppose *pour cette question seulement* que T est le système décrit en figure 2. On pose $\Phi = \{s_2, s_4, s_5, s_6\}$. Pour quels états s a-t-on $s \models \Phi^\omega$? Même question pour $s \models \mathsf{EA}\Phi$.

Question 8. On considère désormais un système T quelconque. Soit s un état de T. Montrer que $s \not\models \mathsf{EA}\Phi$ si et seulement s'il existe un état s' accessible à partir de s tel que $s' \notin \Phi$ et s' fait un cycle sur lui-même, *i.e.* il existe une exécution de longueur non-nulle de s' à s'.

Vérifier $s_0 \models \mathsf{EA}\Phi$ se ramène donc à vérifier qu'il n'existe pas d'état accessible à partir de s_0 qui soit hors de Φ et dans un cycle. Cela peut se faire par plusieurs parcours du système de transitions. On peut d'abord lancer un parcours (appelé *externe*) à partir de s_0 pour déterminer tous les états s' accessibles à partir de s_0. Lors de ce parcours externe, pour chaque état s' visité tel que $s' \notin \Phi$, on lance un nouveau parcours (appelé *interne*) à partir de s' pour déterminer s'il est dans un cycle.

Cette approche naïve peut cependant être améliorée en évitant des parcours internes redondants : c'est l'algorithme des parcours imbriqués donné en figure 3. Ici, le parcours externe a vocation à être encapsulé dans la fonction `outer_iter`, qui doit itérer sur les états accessibles à partir de s_0 — sa spécification précise et son implémentation sont l'objet des prochaines questions. Les parcours internes sont réalisés par la fonction `cycle_check`. Il faut bien noter que les multiples parcours internes partagent le tableau `marked` : ainsi, un état ne sera traité par `inner_visit` qu'au plus une fois lors de *tous* les appels à `cycle_check` au sein d'un même appel à `verify`.

```
(** Parcours externe: [outer_iter t s0 f] appelle la fonction [f] sur
    tous les états [s] accessibles à partir de [s0]. *)
let outer_iter (t:trans_sys) (s0:state) (f:state->unit) : unit =
  ... (* Implémentation à réaliser en question 10. *)

(** Exception utilisée pour interrompre la recherche. *)
exception Found

(** [verify t phi s0] vérifie que [s0] satisfait [EA phi].
    Ici [phi] n'est pas un ensemble mais une fonction,
    représentant l'ensemble des états [s] tels que [phi s = true]. *)
let verify (t:trans_sys) (phi:state->bool) (s0:state) : bool =

  (* Parcours internes: détection de cycle *)
  let marked = Array.make t.size false in
  let cycle_check start =
    let rec inner_visit s =
      if not marked.(s) then begin
        marked.(s) <- true;
        List.iter
          (fun (_,s') ->
             if s' = start then raise Found;
             inner_visit s')
          (t.next s)
      end
    in
    inner_visit start
  in

  (* Parcours externe: états accessibles *)
  try
    outer_iter t s0
      (fun s' -> if not (phi s') then cycle_check s');
    true
  with Found -> false
```

FIGURE 3 – Algorithme des parcours imbriqués.

Question 9. Soit s_0 un état quelconque de T. Un parcours en profondeur à partir de s_0 induit un arbre couvrant l'ensemble des états accessibles dans T à partir de s_0. On considère une énumération des états aux nœuds de cet arbre satisfaisant les deux conditions suivantes :

(a) L'énumération suit un ordre postfixe.

(b) Si deux nœuds s et s' ont le même parent dans l'arbre, et que s a été visité avant s' dans le parcours en profondeur, alors s (et tous ses descendants dans l'arbre) apparaît avant s' (et tous ses descendants) dans l'énumération.

Montrer que, si un état s_1 apparaît avant s_2 dans cette énumération, et qu'il existe une exécution de s_1 à s_2 dans T, alors s_1 fait un cycle sur lui-même dans T, *i.e.* il existe une exécution de longueur non-nulle de s_1 à s_1 dans T.

Question 10. Donner une implémentation de `outer_iter` telle que `outer_iter t s f` itère la fonction `f` sur tous les états accessibles à partir de `s` dans `t`, selon un ordre satisfaisant les conditions (a) et (b) de la question précédente.

Question 11. On suppose que `outer_iter` implémente correctement la spécification donnée dans la question précédente. Proposer une spécification pour la fonction `cycle_check`, montrer la correction de la fonction par rapport à cette spécification, et s'appuyer sur ce résultat pour démontrer la correction de `verify`. On ne demande pas de démontrer la terminaison. On attend en revanche des pré- et postconditions sur l'état du tableau `marked` ainsi qu'une spécification des cas où l'exception `Found` est levée.

Question 12. La fonction `verify` reste-t-elle correcte si l'on modifie `outer_iter` pour parcourir les états accessibles selon un ordre préfixe au lieu de postfixe ? Justifier votre réponse, en expliquant comment l'argument de la question précédente s'adapte, ou en donnant un contre-exemple à moins de quatre états.

Partie III : Programmation multi-threads en C

On rappelle les éléments de programmation concurrente en C au programme, via trois fonctions déclarées dans l'en-tête <pthread.h> et dont les prototypes sont donnés en figure 4 :

- La fonction `pthread_create` lance un nouveau thread (en français, fil d'exécution). Elle prend en premier argument un pointeur `pthread` vers un descripteur de thread dans lequel les informations concernant le nouveau thread seront remplies, pour permettre de s'y référer ensuite. En second argument, elle prend un pointeur `attr` vers des attributs de threads ; il n'est pas utile d'en connaître la teneur exacte, car on se contentera de donner ici une valeur par défaut décrite plus loin. En troisième argument, elle prend une fonction `start_routine` qui attend un argument de type `void*` et renvoie une valeur du même type — on ignorera cette valeur de retour dans le sujet. En dernier argument est donnée la valeur `arg` qui sera passée à la fonction `start_routine` pour démarrer le calcul dans le nouveau thread.
- La fonction `pthread_join` permet d'attendre la fin de l'exécution du thread dont le descripteur est donné en premier argument. Dans ce sujet, on lui passera toujours `NULL` en second argument, et on ignorera l'entier renvoyé.
- La fonction `pthread_attr_init` prend en argument un pointeur vers des attributs de thread, et initialise ces attributs avec des valeurs par défaut. On ignorera l'entier renvoyé.

On donne en figure 5 un exemple de programme illustrant l'usage de ces fonctions. Cet extrait de programme suppose la donnée préalable d'une fonction `max` calculant le maximum de deux valeurs de type `int`. On y exploite par ailleurs la représentation des tableaux en C : un tableau est un pointeur, dont la valeur est l'adresse du premier élément du tableau. Plus généralement, l'adresse de l'élément d'index k d'un tableau de n éléments représente le tableau de $n - k$ éléments commençant à l'index k. Ainsi, en figure 5, l'expression `&(array[2])` représente la deuxième moitié du tableau `array` et, quand le deuxième thread accède à `((int*)pair)[0]`, il lit en fait la valeur `array[2]`.

Question 13.

a. Quels sont les affichages possibles à la fin de l'exécution du programme de la figure 5 ? Expliquer brièvement comment chaque valeur peut être obtenue.

b. L'objectif de ce programme est manifestement d'obtenir dans `m` le maximum des quatre entiers contenus dans `array`. Quelle est la façon canonique de corriger notre programme pour éviter les exécutions menant à un résultat incorrect ? Proposer une correction du programme par l'insertion de quelques lignes [1]. On ne demande pas de produire du code C valide : une description haut niveau des instructions à insérer suffit.

1. Bien sûr, la correction devra fonctionner quelles que soient les quatre valeurs initialement contenues dans `array`, dès lors qu'elles sont positives.

```c
/* Les types pthread_t (descripteur de thread)
   et pthread_attr_t (attributs de thread)
   sont déclarés dans <pthread.h>.
   Il est inutile de préciser leurs définitions. */

int pthread_attr_init(pthread_attr_t *attr);
int pthread_create(pthread_t *pthread,
                   pthread_attr_t *attr,
                   void *(*start_routine)(void *),
                   void *arg);
int pthread_join(pthread_t pthread, void **retval);
```

FIGURE 4 – Fonctions de la bibliothèque `pthread`.

```c
int m = -1;

void* max2(void* pair) {
  int m1 = ((int*)pair)[0];
  int m2 = max(m1, ((int*)pair)[1]);
  int m3 = max(m2, m);
  m = m3;
  return NULL;
}

int main() {
  int array[4] = {10, 20, 30, 40};
  /* Attributs par défaut des threads */
  pthread_attr_t attr;
  pthread_attr_init(&attr);
  /* Création des threads */
  pthread_t thread1, thread2;
  pthread_create(&thread1, &attr, max2, array);
  pthread_create(&thread2, &attr, max2, &(array[2]));
  /* Attente de fin et affichage du résultat */
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
  printf("m == %d\n", m);
}
```

FIGURE 5 – Exemple de programme C multi-threads.

On souhaite généraliser le programme de la figure 5 pour calculer le maximum d'un tableau en répartissant le travail entre plusieurs threads. On suppose déclarées trois variables globales :

— `const int nb_threads` indiquant le nombre de threads à utiliser ;
— `const int items_per_thread` indiquant combien de valeurs chaque thread devra traiter ;
— `int m`, initialisée à `-1`, destinée à contenir le résultat final du calcul comme dans l'exemple de la figure 5.

On suppose enfin qu'on dispose d'une fonction `void* local_max(void* a)` analogue à `max2` mais pour un tableau de `items_per_thread` éléments plutôt que deux. Plus précisément :

— La fonction doit être appelée (précondition) avec en argument `a` l'adresse d'un tableau contenant (au moins) `items_per_thread` éléments de type `int`, tous positifs.
— À titre indicatif, la fonction calcule alors le maximum `v` de ces `items_per_thread` éléments, puis effectue l'affectation `m = v` si la variable globale `m` est inférieure à `v`. Le fonctionnement précis de la fonction n'a cependant pas d'importance dans la suite du sujet.

Question 14. Écrire une fonction `main` qui :

a. Alloue sur le tas un tableau `array` de `nb_threads*items_per_thread` valeurs de type `int`.
b. Initialise chaque case du tableau avec un entier positif pseudo-aléatoire. Celui-ci sera simplement obtenu via l'expression `rand()%max_item` qui renvoie un entier entre 0 (inclus) et `max_item` (exclu), cette dernière variable étant supposée déclarée globalement.
c. Lance `nb_threads` exécutant tous la fonction `local_max`, appelée dans chaque thread sur l'adresse d'un sous-tableau de `array` commençant à un index de la forme `k*items_per_thread`, de sorte que chaque valeur du tableau sera prise en compte par exactement un thread.
d. Affiche la valeur de `m` une fois que tous les threads ont terminé leur exécution.

Cette fonction ne devra occasionner aucune fuite de mémoire. On ne demande pas dans cette question de régler d'éventuels problèmes d'accès concurrents comme en question 13.

Partie IV : Systèmes de transitions concurrents

Les systèmes de transitions sont souvent utilisés pour modéliser des systèmes concurrents, par exemple des programmes multi-threads C. Dans ce cas, la notion d'indépendance entre actions concurrentes induit un quotient sur l'espace des traces, appelant à la vision d'une trace comme un ordre partiel. Ces notions peuvent être exploitées pour réduire le nombre d'états et/ou de traces à explorer lors de la vérification d'un système. On introduit certaines de ces techniques dans cette partie, en s'appuyant sur des exemples correspondant à des programmes multi-threads C pour les illustrer.

On s'intéresse tout d'abord à la modélisation de programmes multi-threads par des systèmes de transitions, que nous illustrons sur l'exemple de la figure 5, dans sa version originale *sans prendre en compte la correction demandée en question 13*. Ce programme comporte deux threads dont chacun peut effecter quatre instructions, aux lignes 4 à 7 ; on ignorera ici l'instruction `return`. On le modélise comme un système de transitions dont les états sont les états du programme, qui sont donnés par la valeur des variables locales et globales, et la position de chaque thread dans son exécution. Ces positions seront prises dans $\{0, 1, 2, 3, 4\}$, la position j indiquant que le thread a exécuté ses j premières instructions. On utilisera des actions de la forme t_i^j avec $i \in \{1, 2\}$ et $j \in \{1, 2, 3, 4\}$: l'action t_i^j représente l'exécution de la $j^{\text{ème}}$ instruction du thread i.

On définit quelques états, en utilisant une notation expliquée juste après :

$$
\begin{aligned}
s_0 &\stackrel{\text{def}}{=} (\texttt{m=-1}, t_1@0, t_2@0)\\
s' &\stackrel{\text{def}}{=} (\texttt{m=-1}, t_1@0, t_2@1(\texttt{m1=30}))\\
s_1 &\stackrel{\text{def}}{=} (\texttt{m=-1}, t_1@2(\texttt{m1=10,m2=20}), t_2@2(\texttt{m1=30,m2=40}))\\
s'' &\stackrel{\text{def}}{=} (\texttt{m=-1}, t_1@3(\texttt{m1=10,m2=20,m3=20}), t_2@2(\texttt{m1=30,m2=40}))
\end{aligned}
$$

L'état s_0 correspond à l'état initial de notre programme : la variable globale `m` vaut `-1` ; chaque thread i est à la position 0, ce que l'on note $t_i@0$; aucune variable locale n'est spécifiée car aucune n'est encore définie. Dans cet état, deux actions sont disponibles : t_1^1 et t_2^1. La transition t_2^1 mène à l'état s', où le second thread est passé à la position 1 et a défini sa variable locale `m1` avec pour valeur 30. Ensuite, l'état s_1 correspond à la situation où chaque thread a défini ses deux premières variables locales et s'apprête à définir la dernière, en fonction de la valeur de la variable globale `m`. Enfin, on a $s'' = \delta(s_1, t_1^3)$.

On notera que cette modélisation considère chaque instruction du programme comme étant atomique. C'est une hypothèse raisonnable sur notre exemple, où une instruction ne contient pas plus d'une lecture ou écriture sur une variable globale.

Question 15. À partir de l'état s_0, notre système de transitions permet des exécutions de longueur au plus huit. Combien d'exécutions différentes de longueur huit sont-elles possibles ?

Attention ! *Dans la fin du sujet, les systèmes modélisant des programmes C sont utilisés uniquement pour illustrer des notions générales liées à la concurrence dans les systèmes de transitions. On prendra garde à ne pas interpréter ces notions dans le contexte restrictif des programmes C, et à traiter en toute généralité les questions qui ne font pas explicitement référence à un exemple.*

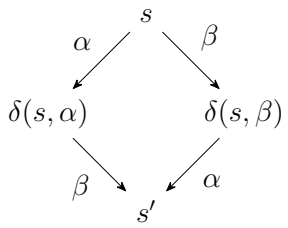
Dans un système de transitions $T = (S, A, \mathcal{D}, \delta)$, on dit que deux actions $\alpha, \beta \in A$ sont **indépendantes** quand $\alpha \neq \beta$ et les conditions suivantes sont satisfaites :

— Exécuter α ne change pas la disponibilité de β :
pour tout $s \in S$ tel que $\alpha \in \mathcal{D}(s)$, on a $\beta \in \mathcal{D}(s) \Leftrightarrow \beta \in \mathcal{D}(\delta(s, \alpha))$.

— Exécuter β ne change pas la disponibilité de α :
pour tout $s \in S$ tel que $\beta \in \mathcal{D}(s)$, on a $\alpha \in \mathcal{D}(s) \Leftrightarrow \alpha \in \mathcal{D}(\delta(s, \beta))$.

— L'ordre d'exécution de α et β n'a pas d'importance :
pour tout $s \in S$ tel que $\alpha \in \mathcal{D}(s)$ et $\beta \in \mathcal{D}(s)$, on a $\delta(\delta(s, \alpha), \beta) = \delta(\delta(s, \beta), \alpha)$.

L'indépendance de α et β est notée $\alpha \leftrightarrow \beta$. Elle peut être résumée de façon moins précise mais visuelle comme suit, pour tout état s, avec $s' = \delta(\delta(s, \alpha), \beta) = \delta(\delta(s, \beta), \alpha)$:



Pour un mot $u \in A^*$, on écrira $\alpha \leftrightarrow u$ quand $\alpha \leftrightarrow u_i$ pour tout $i < |u|$. Pour un ensemble d'actions $E \subseteq A$, on écrira $\alpha \leftrightarrow E$ quand $\alpha \leftrightarrow \beta$ pour tout $\beta \in E$.

Question 16. Dans le système correspondant à la figure 5, a-t-on indépendance des deux actions disponibles en s_0 ? Même question pour les états s', s_1 et s''.

Pour $s_0 \in S$ un état, on dit qu'un ensemble $X \subseteq \mathcal{D}(s_0)$ est un **ensemble persistant pour** s_0 quand les deux conditions suivantes sont satisfaites :

(**A1**) $X \neq \emptyset$ si $\mathcal{D}(s_0) \neq \emptyset$.

(**A2**) Pour toute exécution $s_0 \xrightarrow{\alpha_0} \ldots \xrightarrow{\alpha_n} s_{n+1}$ à partir de s_0 et telle que $\alpha_i \notin X$ pour tout $i \leq n$, on a $\alpha_i \leftrightarrow X$ pour tout $i \leq n$.

La seconde condition impose que, si l'on exécute à partir de s_0 une séquence d'actions qui ne sont pas dans X, ces actions seront nécessairement indépendantes de toutes les actions de X. On notera que $\mathcal{D}(s_0)$ est trivialement persistant pour s_0.

Question 17. Soit s_0 un état, X un ensemble persistant pour s_0, et $\alpha \in X$. On considère une exécution comme dans la condition (**A2**), *i.e.* de la forme $s_0 \xrightarrow{\alpha_0} \ldots \xrightarrow{\alpha_n} s_{n+1}$ avec $\alpha_i \notin X$ pour tout $i \leq n$. Montrer que $\alpha \in \mathcal{D}(s_i)$ pour tout $i \leq n + 1$.

Question 18. Sur notre exemple issu de la figure 5, donner un ensemble persistant minimal en s_0, *i.e.* un ensemble persistant dont aucun sous-ensemble strict n'est persistant. Même question en s_1.

La question suivante énonce la propriété fondamentale des ensembles persistants : les états terminaux accessibles le restent si l'on se restreint aux exécutions dont toutes les transitions sont choisies dans des ensembles persistants.

Question 19. On se donne une application $X : S \to 2^A$ telle que, pour tout $s \in S$, l'ensemble d'actions $X(s)$ est un ensemble persistant pour s. Soit s_0 un état quelconque, et s_t un état terminal accessible à partir de s_0 dans T : on a donc une exécution de s_0 à s_t, mais aucune exécution possible à partir de s_t. Montrer qu'il existe une exécution

$$\pi : s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \dots s_{n-1} \xrightarrow{\alpha_{n-1}} s_t$$

de longueur $n \geq 0$ telle que $\alpha_i \in X(s_i)$ pour tout $i < n$.

Pour un système de transitions donné T, on définit la relation $\sim$ d'**équivalence par permutation** sur les mots de A^* comme la plus petite relation d'équivalence sur A^* telle que, pour toutes les actions $\alpha, \beta \in A$ satisfaisant $\alpha \leftrightarrow \beta$, on a $u\alpha\beta v \sim u\beta\alpha v$ pour tous $u, v \in A^*$. On remarque immédiatemment que si $u, v \in A^*$ sont tels que $u \sim v$, alors pour tous $s, s' \in S$ tels que $s \xrightarrow{u} s'$, on a aussi $s \xrightarrow{v} s'$.

On définit $A^{\neq}$ comme l'ensemble des mots sans répétition sur A, *i.e.* l'ensemble des mots $u \in A^*$ tels que $u_i \neq u_j$ pour tous $0 \leq i < j < |u|$. Travailler avec $A^{\neq}$ plutôt que A^* simplifie la suite du sujet.

Étant donné un mot $u \in A^{\neq}$ de longueur $n \in \mathbb{N}$, on définit l'**ordre induit** $<_u$ sur l'ensemble $\{u_0, \dots, u_{n-1}\}$ des lettres de u comme la plus petite relation transitive satisfaisant la condition suivante : pour tous $i, j \in \{0, \dots, n-1\}$ tels que $i < j$ et $u_i \not\leftrightarrow u_j$, on a $u_i <_u u_j$.

Question 20. On illustre la notion d'ordre induit sur quelques exemples simples.

a. On suppose que T est construit sur $A = \{\alpha, \beta, \gamma\}$ de sorte que $\beta \not\leftrightarrow \alpha$ et $\beta \not\leftrightarrow \gamma$ mais $\alpha \leftrightarrow \gamma$. Donner les ordres induits pour $u = \alpha\beta\gamma$, pour $v = \alpha\gamma\beta$ et pour $w = \gamma\alpha\beta$.

b. On suppose que T est construit sur $A = \{\alpha, \alpha', \beta, \beta'\}$ de sorte que $\alpha \not\leftrightarrow \beta$ et $\alpha' \not\leftrightarrow \beta'$, mais on a indépendance pour toutes les autres paires d'actions distinctes. Donner les ordres induits pour $u = \alpha\alpha'\beta\beta'$, pour $v = \alpha\beta\alpha'\beta'$ et pour $w = \beta\alpha\alpha'\beta'$.

Question 21. Soient $u, v \in A^{\neq}$ de même longueur $n \in \mathbb{N}$, et contenant les mêmes lettres, *i.e.* $\{u_0, \dots, u_{n-1}\} = \{v_0, \dots, v_{n-1}\}$. Montrer qu'on a $u \sim v$ si et seulement si $<_u$ et $<_v$ coïncident.

On définit enfin les exécutions avec **mise en veille**. Étant donnés un système de transitions $T = (S, A, \mathcal{D}, \delta)$ et un ordre total $<$ sur A, le système $T^< = (S^<, A, \mathcal{D}^<, \delta^<)$ est défini comme suit :

— $S^< = S \times 2^A$, *i.e.* les états de $S^<$ sont constitués d'un état de S auquel on adjoint un ensemble d'actions ;

— pour tout état $(s, Z) \in S^<$, on a $\mathcal{D}^<((s, Z)) = \mathcal{D}(s) \setminus Z$;

— pour tous $(s, Z) \in S^<$ et $\alpha \in \mathcal{D}^<(s)$, on a $\delta^<((s, Z), \alpha) = (\delta(s, \alpha), Z')$ avec

$$Z' = \{\beta \in Z \mid \alpha \leftrightarrow \beta\} \cup \{\beta \in \mathcal{D}(s) \mid \alpha \leftrightarrow \beta, \beta < \alpha\}.$$

```
1  void* thread1(void* _) {
2    int a = v;
3    v = a+1;
4    return NULL;
5  }
```

```
1  void* thread2(void* _) {
2    int b = v+2;
3    v = 3*b;
4    return NULL;
5  }
```

FIGURE 6 – Deux threads pour illustrer la mise en veille.

Question 22. On considère un programme comportant deux threads exécutant respectivement les deux fonctions de la figure 6, manipulant une variable globale `v` de type `int` initialisée à 0. Soit T le système de transitions modélisant ce programme, dans le même style que précédemment, sur $A = \{t_1^1, t_1^2, t_2^1, t_2^2\}$ correspondant aux instructions des lignes 2 et 3 dans `thread1` et `thread2` respectivement. Soit s_0 l'état correspondant à l'état initial du programme.

a. Dessiner le système de transitions T, en indiquant l'action associée à chaque transition et la valeur des variables locales et globale définies en chaque état. Il est inutile de décrire complètement les états comme on l'a fait en début de partie pour certains états du programme de la figure 5.

b. On considère l'ordre sur les actions donné par $t_1^1 < t_1^2 < t_2^1 < t_2^2$. Dessiner les états accessibles à partir de $(s_0, \emptyset)$ dans $T^<$ et les transitions entre ceux-ci. On pourra se contenter d'indiquer (idéalement, en couleur) les modifications sur le dessin précédent : quels ensembles Z sont associés à chaque état, et quelles transitions ne sont plus possibles dans $T^<$.

On munit $A^{\neq}$ de l'ordre lexicographique induit par l'ordre $<$ sur les actions. L'ordre lexicographique sera simplement noté $<$. Cela permet de définir, pour tout mot $u \in A^{\neq}$, $u^<$ comme le **minimum lexicographique** de la classe d'équivalence de u pour $\sim$. (Ce minimum est bien défini car la classe d'équivalence ne contient que des mots de même longueur que u.)

Question 23. Soit $u \in A^{\neq}$. Montrer que $u = u^<$ si et seulement si u ne peut pas s'écrire $u_1 \beta u_2 \alpha u_3$ avec $u_1, u_2, u_3 \in A^{\neq}$ et $\alpha, \beta \in A$ tels que $\alpha < \beta$ et $\alpha \leftrightarrow \beta u_2$.

Question 24. En déduire que la mise en veille sélectionne les minima lexicographiques. Plus précisement, démontrer les deux résultats suivants pour tous $u \in A^{\neq}$ et $s, s' \in S$:

a. Pour tous $Z, Z' \subseteq A$, si on a $(s, Z) \xrightarrow{u} (s', Z')$ dans $T^<$, alors $u = u^<$.

b. Si $s \xrightarrow{u} s'$ dans T, alors il existe Z tel que $(s, \emptyset) \xrightarrow{u^<} (s', Z)$ dans $T^<$.

Fin du sujet.