

Exercice 1.1 Exprimer les propriétés suivantes par des formules de LTL.

- La propriété p arrive un jour.
- La propriété p est toujours vraie.
- La propriété p est vraie à l’instant 1.
- La propriété p est stable (si elle arrive, elle demeure).
- Exclusion mutuelle : p et q n’arrivent jamais en même temps.
- Tout p est suivi immédiatement après par un p ou un q .
- Vivacité : la propriété p est répétée infiniment souvent.
- Toute demande de ressource est acquittée plus tard.
- Toute demande de ressource est acquittée plus tard et aucune demande supplémentaire n’arrive avant l’acquittement.
- Équité 1 : toute demande continuellement répétée finit par être acquittée.
- Équité 2 : toute demande infiniment répétée finit par être acquittée.
- Équité 3 : toute demande infiniment répétée est acquittée infiniment souvent.
- L’alarme reste active (propriété a) tant que le bouton stop n’est pas appuyé (la propriété s exprime que stop est appuyé).

Exercice 1.2 Trouver une formule LTL et un automate de Büchi décrivant chacun des langages suivants. L’ensemble des propositions est $AP = \{q, r\}$.

- ensemble des mots commençant par une lettre vérifiant q .
- ensemble des mots ayant un nombre infini de positions satisfaisant q .
- ensemble des mots ayant un nombre infini de positions satisfaisant q et un nombre infini de positions satisfaisant r .
- ensemble des mots contenant un unique facteur ab tel que $q \in a$ et $r \in b$.
- ensemble des mots ayant un nombre *fini* de positions satisfaisant q .

Exercice 1.3 L’opérateur *weak until* W est défini par $\alpha W \beta \stackrel{\text{def}}{=} (\alpha U \beta) \vee G\alpha$. Trouver les équivalences qui sont des tautologies parmi les suivantes.

1. $\alpha U \beta \iff \beta \vee (\alpha \wedge X(\alpha U \beta))$,
2. $\neg(\alpha U \beta) \iff (\neg\beta) W (\neg\alpha \wedge \neg\beta) \iff (\neg\alpha) R (\neg\beta)$,
3. $\neg X\alpha \iff X\neg\alpha$,
4. $G(\alpha \wedge \beta) \iff G\alpha \wedge G\beta$,
5. $F(\alpha \wedge \beta) \iff F\alpha \wedge F\beta$,
6. $FG(\alpha \wedge \beta) \iff FG\alpha \wedge FG\beta$,
7. $\alpha U (\beta \vee \gamma) \iff (\alpha U \beta) \vee (\alpha U \gamma)$,
8. $\alpha U (\beta \wedge \gamma) \iff (\alpha U \beta) \wedge (\alpha U \gamma)$,
9. $(\alpha \wedge \beta) U \gamma \iff (\alpha U \gamma) \wedge (\beta U \gamma)$.

Exercice 1.4 On rappelle que la logique LTL(X, U) utilise deux opérateurs temporels $next$ et *Until*. On considère la logique LTL(XU) ayant un unique opérateur temporel XU , dont la sémantique est donnée par :

$$x, k \models \alpha XU \beta \iff \exists \ell [(\ell > k) \wedge (x, \ell \models \beta) \wedge \forall i (k < i < \ell \Rightarrow x, i \models \alpha)].$$

Écrire une formule équivalente à $\alpha XU \beta$ en LTL(X, U). Inversement, écrire des formules équivalentes à $X\alpha$ et $\alpha U \beta$ en LTL(XU).

Exercice 1.5 Montrer que le langage des mots ayant un nombre *fini* de b , sur $A = \{a, b\}$, n'est pas accepté par un automate de Büchi *déterministe*.

Exercice 1.6 À partir de deux automates de Büchi reconnaissant des langages K et L , construire un automate de Büchi reconnaissant $K \cap L$.

Exercice 1.7 Un *automate de Büchi généralisé* $\mathcal{A} = (S, A, T, I, F_1, \dots, F_k)$ sur l'alphabet A est donné par un ensemble d'états S , un ensemble de transitions $T \subseteq S \times A \times S$, un ensemble d'états initiaux $I \subseteq S$, et un sous-ensemble d'états F_i . Un run est accepté s'il rencontre infiniment souvent *chaque* ensemble F_k . Montrer qu'à partir d'un automate de Büchi généralisé, on peut construire un automate de Büchi usuel reconnaissant le même langage.

Exercice 1.8 Pour chacune des formules LTL suivantes, construire un automate de Büchi reconnaissant les modèles de la formule.

- $\alpha = (a \cup b) \cup c$,
- $\beta = a \cup (b \cup c)$,
- $\gamma = F((p \wedge \text{XX}\neg p) \vee (\neg p \wedge \text{XX}p))$,
- $\delta = \neg\gamma$,
- $\varphi = G(r \Rightarrow X(\neg r \cup g))$,
- $\psi = GF r \Rightarrow GF g$.

Exercice 1.9 Un *automate de Büchi sur les transitions* est donné par $\mathcal{A} = (S, A, T, I, T_1)$ où A est l'alphabet, S est l'ensemble des états, $T \subseteq S \times A \times S$ est l'ensemble des transitions, $I \subseteq S$ est l'ensemble des états initiaux, et $T_1 \subseteq T$ est un sous-ensemble de l'ensemble des transitions. Un run est accepté s'il passe infiniment souvent par une transition de T_1 . Montrer qu'à partir d'un automate de Büchi sur les transitions, on peut construire un automate de Büchi classique qui reconnaît le même langage.

Montrer réciproquement que tout automate de Büchi classique peut être transformé en automate de Büchi sur les transitions.

Quelle est la taille de l'automate classique obtenu, en fonction de celle de l'automate sur les transitions ?

Exercice 1.10 Un automate de Büchi généralisé sur les transitions $\mathcal{A} = (Q, T, I, T_1, \dots, T_k)$ est un tuple dont les premières composantes sont comme dans l'exercice 1.9, et pour $1 \leq i \leq k$: $T_i \subseteq T$ (noter que si k est nul, il n'y a aucun ensemble T_i). Un run est acceptant si pour tout i , il utilise infiniment souvent une transition de T_i . Montrer qu'un tel automate se transforme en automate de Büchi sur les transitions avec une seule condition d'acceptation.

Exercice 1.11 Que peut-on dire d'un automate comme dans l'exercice 1.10 si $k = 0$ (c'est-à-dire, aucun ensemble T_i n'est donné) ?

Exercice 1.12 Étant donnés des automates \mathcal{A}_α et \mathcal{A}_β reconnaissant les modèles de formules LTL α et β , construire les automates pour les formules $X\alpha$ et $\alpha \cup \beta$. En déduire un algorithme pour construire un automate \mathcal{A}_φ à partir d'une formule φ donnée, reconnaissant les modèles de la formule φ . Évaluer la taille de l'automate construit, dans le cas le pire, en fonction de la taille de φ .

Exercice 1.13 Utiliser la construction de l'exercice 1.12 pour trouver des automates pour les premières formules de l'exercice 1.8.

On rappelle les deux constructions d'un automate de Büchi reconnaissant les mots du langage $L(\varphi) = \{x \in \Sigma^\omega \mid x \models \varphi\}$ décrit par une formule LTL φ .

Algorithme 1 On considère l'ensemble $\text{cl}(\varphi)$ composé

- des sous-formules de φ ,
- des formules $\mathbf{X}\beta$ où $\beta = \gamma \cup \delta$ est une sous-formule de φ ,
- des négations des formules précédentes.

Un état q de \mathcal{A}_φ est un sous ensemble de $\text{cl}(\varphi)$ « consistant », c'est-à-dire vérifiant les propriétés C_1, C_2, C_3 suivantes.

$$(C_1) \quad \forall \beta \in \text{cl}(\varphi), \beta \in q \Leftrightarrow \neg\beta \notin q.$$

$$(C_2) \quad \forall \beta \vee \gamma \in \text{cl}(\varphi), \beta \vee \gamma \in q \Leftrightarrow \beta \in q \text{ ou } \gamma \in q.$$

$$(C_3) \quad \forall \beta \cup \gamma \in \text{cl}(\varphi), \beta \cup \gamma \in q \Leftrightarrow \gamma \in q \text{ ou } (\beta \in q \text{ et } \mathbf{X}(\beta \cup \gamma) \in q).$$

L'automate \mathcal{A}_φ est alors défini par :

- États initiaux = $\{q \mid \varphi \in q\}$.
- Transitions : $q \xrightarrow{P} q'$ si $P = q \cap \text{AP}$ et, pour tout $\mathbf{X}\beta \in \text{cl}(\varphi)$, $\mathbf{X}\beta \in q \Leftrightarrow \beta \in q'$. Note : P peut être vide.
- Soient $\gamma \cup \delta$ les formules Until apparaissant dans $\text{cl}(\varphi)$. On ajoute une condition de Büchi généralisée pour chaque formule de ce type :

$$F_{\gamma \cup \delta} = \{q \mid \neg(\gamma_i \cup \delta_i) \in q \text{ ou } \delta_i \in q\}$$

Algorithme 2 (plus efficace) (P. Gastin, MOVEP'04). On commence par mettre la formule φ en forme normale négative, en faisant descendre les négations au niveau des propositions atomiques. Une formule est en forme normale négative si elle est construite à partir de **false**, p , $\neg p$ (où $p \in \text{AP}$) en utilisant les opérateurs $\mathbf{X}, \cup, \mathbf{R}, \vee, \wedge$. Dans la suite, on suppose que φ est déjà mise en forme normale négative.

Un état est un sous-ensemble de sous-formules de φ . L'état initial est $\{\varphi\}$. Un sous-ensemble Z de formules en forme normale négative est *réduit* si

- les formules de Z sont de la forme $p, \neg q$, ou $\mathbf{X}\alpha$.
- **false** $\notin Z$, et $\{p, \neg p\} \not\subseteq Z$ pour tout $p \in \text{AP}$.

Pour définir les transitions à partir d'un état Y , on forme le graphe à partir de Y de la façon suivante. Soit $Y = Z \cup \{\alpha\}$ où α n'est pas réduite et est maximale dans Y (c'est-à-dire que α n'est pas sous-formule d'une telle autre formule de Y). Les arêtes à partir de Y sont les suivantes.

- Si $\alpha = \alpha_1 \vee \alpha_2$, $Y \rightarrow Z \cup \{\alpha_1\}$ et $Y \rightarrow Z \cup \{\alpha_2\}$.
- Si $\alpha = \alpha_1 \wedge \alpha_2$, $Y \rightarrow Z \cup \{\alpha_1, \alpha_2\}$,
- Si $\alpha = \alpha_1 \mathbf{R} \alpha_2$, $Y \rightarrow Z \cup \{\alpha_1, \alpha_2\}$ et $Y \rightarrow Z \cup \{\mathbf{X}\alpha, \alpha_2\}$
- Si $\alpha = \alpha_1 \cup \alpha_2$, $Y \rightarrow Z \cup \{\alpha_2\}$ et $Y \xrightarrow{\alpha} Z \cup \{\mathbf{X}\alpha, \alpha_1\}$.

La dernière arête est marquée par α . Soient

$$\text{Red}(Y) = \{Z \text{ réduit} \mid Y \xrightarrow{*} Z\}$$

$$\text{Red}_\alpha(Y) = \{Z \text{ réduit} \mid Y \xrightarrow{*} Z \text{ sans utiliser une arête marquée par } \alpha\}$$

et soit pour Z réduit

$$\text{next}(Z) = \{\alpha \mid \mathbf{X}\alpha \in Z\}$$

$$\Sigma_Z = \bigcap_{p \in Z} \Sigma_p \cap \bigcap_{\neg p \in Z} \Sigma_{\neg p}$$

Les transitions depuis un état Y sont alors : $\{Y \xrightarrow{\Sigma_Z} \text{next}(Z) \mid Z \in \text{Red}(Y)\}$ On ajoute une condition d'acceptation sur les transitions pour chaque sous-formule α qui est un Until : $F_\alpha = \{Y \xrightarrow{\Sigma_Z} \text{next}(Z) \mid Y \in Q \text{ et } Z \in \text{Red}_\alpha(Y)\}$.

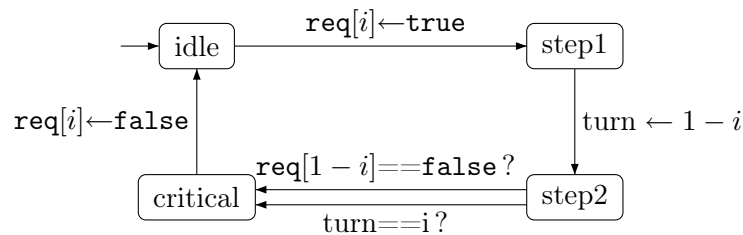
Exercice 2.1 On considère le code suivant pour 2 processus $i = 0, 1$.

```
bool flags[2]; // Tableau partagé, initialisé à false.
Thread(i) {
  while (flags[1-i] == true)
    ;
  flags[i] = true;
  section_critique();
  flags[i] = false;
  section_non_critique();
}
```

Modéliser le programme par structure de Kripke avec variables, et donner une trace d'exécution montrant que l'exclusion mutuelle n'est pas assurée.

Exercice 2.2 Modéliser en SPIN l'algorithme de Peterson pour 2 processus, dont la structure de Kripke pour le processus i est rappelée ci-dessous, et vérifier si l'exclusion mutuelle et l'absence de blocage sont vérifiées :

- en utilisant des assertions;
- en utilisant une formule LTL.



Exercice 2.3 L'algorithme distribué de Dolev, Klawe et Rodeh (1982) permet l'élection de leader dans un anneau unidirectionnel. Chaque processus a initialement une identité, un entier qui l'identifie sur l'anneau. L'objectif est qu'un et un seul processus se déclare leader. On demande de plus que chaque processus exécute le même algorithme. L'algorithme DKR est le suivant.

- Chaque processus peut fonctionner suivant deux modes : tant qu'il pense pouvoir être élu leader, un processus est dans un mode actif. Dès qu'il détecte qu'il ne sera pas leader, il passe en mode passif, dans lequel il se contente de relayer les messages qu'il reçoit.
- Tous les processus sont initialement actifs.
- Chaque processus maintient une variable **leader** valant initialement son identité.

- (a) Un processus commence par transmettre la valeur **val** de sa variable **leader** à son voisin suivant dans l'anneau, et reçoit celle de son prédécesseur, **val1**. Si ces valeurs sont égales, il se déclare **leader**.
- (b) Sinon, il transmet la valeur **val1**, et reçoit la valeur **val2** reçue par son prédécesseur. Si **val1** n'est pas maximale parmi **val**, **val1**, **val2**, il passe en mode passif. Sinon, il affecte la valeur **val1** à la variable **leader** et reprend en (a).

1. Modéliser le protocole en Promela, pour un anneau de taille **N**, #définie.
2. Vérifier les propriétés suivantes en utilisant SPIN :
 - a) Un processus finit par toujours par se déclarer leader.
 - b) On n'obtient jamais deux leaders.

- c) Le processus leader termine avec la valeur de sa variable maximale.
- d) Le nombre d'exécutions de la boucle principale est borné par $N+1$

Exercice 2.4 (Exclusion mutuelle : algorithme de Dekker) L'algorithme de Dekker pour 2 processus P_i ($i = 0, 1$) utilise 3 variables Booléennes partagées `req[i]` et une variable `turn`. Les 2 processus exécutent en parallèle l'algorithme suivant (écrit pour le processus i).

```

1 while(true)
2   Section_Non_Critique() ;
3   while(true)
4     {
5       req[i] = 0;
6       if (req[1-i] == 1)
7         break;
8       if (turn == 1-i)
9         {
10        req[i] = 1;
11        if (turn == i)
12          break;
13        req[i] = 0;
14      }
15    }
16 Section_Critique();
17 req[i] = 1 ;
18 turn = 1-i
19 }
```

- Utiliser SPIN pour vérifier la propriété d'exclusion mutuelle.
- Vérifier que sous des hypothèses d'équité, un processus demandant à entrer dans sa section critique finira par y entrer.

Exercice 2.5 (Exclusion mutuelle : algorithme de Lamport) Dans l'algorithme en pseudo-code suivant, pour le thread i , l'entier N est fixé au départ. C'est le nombre de threads lancés en concurrence. La notation $<$ de la ligne 11 est l'ordre lexicographique : $(x, y) < (z, t)$ soit si $x < z$, soit si $x = z$ et $y < t$.

```

1 int choosing[N], ticket[N]; // Tableaux en accès partagé, initialisés a 0.
2
3 Thread(i) {
4   while (true) {
5     choosing[i] = 1;
6     ticket[i] = 1 + max(ticket[0], ..., ticket[N-1]);
7     choosing[i] = 0;
8     for (j = 0; j < N; j++) {
9       while (choosing[j] != 0)
10      ;
11      while ((ticket[j] != 0) && ((ticket[j], j) < (ticket[i], i)))
12      ;
13    }
14    section_critique();
15    ticket[i] = 0;
16    section_non_critique();
17  }
18 }
```

1. Représenter par une structure de Kripke l'un de ces threads.

2. Évaluer le nombre d'états du système obtenu avec n threads concurrents.
3. Montrer que si le tableau `choosing` n'est pas utilisé, l'algorithme n'assure *pas* l'exclusion mutuelle. On pourra considérer le cas de 2 processus, et trouver un ordonnancement dans lequel les deux processus arrivent en section critique.
 Note : Pour prouver que l'algorithme est correct, par exemple pour $N = 2$, on peut utiliser un *model-checker*.

Exercice 2.6 Pour chacune des propriétés de l'exercice 1.2, écrire une *never claim* permettant de vérifier la propriété sous SPIN. Idem pour la formule $p \cup q$.

Exercice 2.7 Évaluer le nombre d'états du source Promela suivant.

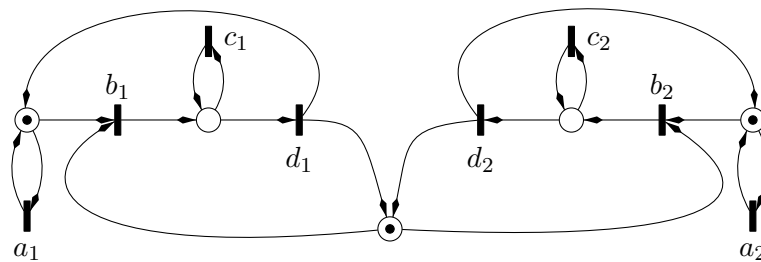
```

init {
    byte i = 0;
    do
        i = i+1
    od
}

```

À quoi peut-on s'attendre si on lance une vérification ? Même questions si on change le `byte` en `short`.

Exercice 2.8 Modéliser le réseau de Petri suivant en Promela.



Quels réseaux de Petri peut-on modéliser en Promela ? Proposer pour ces réseaux un codage systématique.

Exercice 2.9 Écrire l'algorithme *bakery* de Lamport pour l'exclusion mutuelle de n processus, et le code Promela pour vérifier, pour $n = 3$,

- s'il garantit l'exclusion mutuelle,
- s'il y a absence de blocage, et
- si tout processus voulant entrer en section critique finit par y parvenir.