Introduction à la vérification, 2021–2022, M1
Notes de cours et exercices

1 Logique temporelle linéaire LTL

Cette section a pour objectif de présenter la logique temporelle linéaire LTL, puis un algorithme de *model-checking* pour cette logique. L'objectif du ***model-checking***, ou *vérification de modèle*, est de vérifier qu'un système (ou plutôt, un modèle de système) satisfait, ou ne satisfait pas, une propriété. Par vérifier, on entend ici : «vérifier automatiquement, algorithmiquement». Il est irréaliste de vérifier des systèmes arbitraires : en effet, le théorème de Rice affirme que toute propriété sémantique des programme est soit triviale, soit indécidable. Ainsi, on ne peut pas tester algorithmiquement, étant donné le code d'un programme, si, au cours de son exécution,

- il s'arrête,
- il effectue une instruction précise,
- une variable prend une certaine valeur,
- une division par 0 est détectée, ...

Nous travaillerons donc pour commencer avec les modèles de systèmes simples. Les systèmes les plus simples que l'on considère s'appellent des ***structures de Kripke***. Une telle structure ressemble à un automate fini, sans état final. Elle possède donc un nombre fini d'états, reliés par des transitions. Les transitions entre états peuvent être étiquetées par des actions, par des conditions, ou pas étiquetées. Au cours de son exécution, le système représenté par cette structure démarre dans un état initial, et change d'état au cours de son exécution. Une exécution du système est donc une suite (finie ou infinie) d'états. Enfin, chaque état comporte un nombre fini de propriétés, venant formellement d'un ensemble de propriétés atomiques AP.

Cette logique a d'abord été étudiée par des philosophes et des linguistes, comme Hans Kamp en 1968, avant d'être introduite et utilisée en vérification par Amir Pnueli (prix Turing 1996). Elle permet de définir des ensembles de mots de façon intuitive. De façon pratique, ces ensembles de mots peuvent représenter des ensembles d'exécutions de systèmes, souhaitables ou pas. Autrement dit, LTL permet, de façon relativement simple, de définir des ensembles de comportements de systèmes.

On fixe un ensemble fini de ***propriétés atomiques*** AP. Intuitivement, ces propriétés sont des observations que l'on peut faire sur un système informatique.

Cette logique permet de décrire des langages de mots ***infinis*** sur l'alphabet $\Sigma = 2^{\mathsf{AP}}$. Chaque lettre d'un tel mot est donc un sous-ensemble de l'ensemble AP des propriétés atomiques. Par exemple, si $\mathsf{AP} = \{p, q, r\}$, l'alphabet Σ a 8 lettres (l'ensemble vide $\emptyset$, les singletons $\{p\}$, $\{q\}$, $\{r\}$, les ensembles à deux éléments $\{p, q\}$, $\{q, r\}$, $\{r, p\}$ ainsi que AP tout entier).

Nous allons considérer des systèmes effectuant évoluant de façon discrète (par opposition à continue). Ainsi, un tel système fait une action et change de configuration à des instants repérés par des entiers : 0, 1, 2... Le temps est donc considéré comme discret.

Une exécution d'un système est ainsi représentée par un mot infini : la k^{e} lettre de ce mot représente l'ensemble des propriétés atomiques vraies dans la configuration atteinte par le système au temps k.

La logique LTL, dont on présente ici la syntaxe, permet de spécifier des propriétés d'ensembles de telles exécutions. Autrement dit, une formule LTL définit un ensemble de mots infinis : l'ensemble des mots qui satisfont la formule.

1.1 Syntaxe de LTL

Soit $\Sigma = 2^{\mathsf{AP}}$. L'ensemble $\mathsf{LTL} = \mathsf{LTL}(\mathsf{X},\mathsf{U})$ est le plus petit ensemble de formules tel que

- $\mathsf{AP} \subseteq \mathsf{LTL}$.
- $\top \in \mathsf{LTL}$ et $\bot \in \mathsf{LTL}$.
- $\alpha \in \mathsf{LTL} \Longrightarrow \neg\alpha \in \mathsf{LTL}$,
- $\alpha, \beta \in \mathsf{LTL} \Longrightarrow \alpha \vee \beta \in \mathsf{LTL}$,
- $\alpha \in \mathsf{LTL} \Longrightarrow \mathsf{X}\alpha \in \mathsf{LTL}$,
- $\alpha, \beta \in \mathsf{LTL} \Longrightarrow \alpha\,\mathsf{U}\,\beta \in \mathsf{LTL}$.

Par exemple, sur $\mathsf{AP} = \{p, q, r\}$, on peut considérer la formule $\neg\Big(\top\,\mathsf{U}\,(p \wedge \neg(\mathsf{X}q\,\mathsf{U}\,r))\Big)$. X se lit neXt et U se lit Until.

1.2 Sémantique

On va évaluer (*in fine*) les formules de $\mathsf{LTL}(\mathsf{X},\mathsf{U})$ sur des mots sur l'alphabet $\Sigma = 2^{\mathsf{AP}}$:

- ou bien sur les mots finis : Σ^*, lorsqu'on s'intéresse aux programmes dont le comportement est fini.
- ou bien sur les mots infinis : Σ^ω, dans le cadre de la vérification des systèmes réactifs.

Soit $u = u_0 u_1 \cdots \in \Sigma^*$ ou Σ^ω un mot (fini ou infini) et i une position dans le mot (les positions sont numérotées à partir de 0). Si u est fini, la dernière position de u est donc $|u|-1$, où $|u|$ désigne la longueur de u. Soit α une formule de LTL. On définit la relation de satisfaction $u, i \models \alpha$, lue « u satisfait α à la position (ou au temps) i » (où $0 \leqslant i \leqslant |u|-1$), de la façon suivante.

- $u, i \models p$, pour $p \in \mathsf{AP}$, si $p \in u_i$;
- $u, i \models \alpha \vee \beta$ si $u, i \models \alpha$ ou $u, i \models \beta$;
- $u, i \models \neg\alpha$ si l'on n'a pas $u, i \models \alpha$;
- $u, i \models \mathsf{X}\alpha$ si $i+1 \leqslant |u|-1$ (pour les mots finis) et $u, i+1 \models \alpha$;
- $u, i \models \alpha\,\mathsf{U}\,\beta$ s'il existe un entier j qui satisfait les conditions suivantes :
  - $i \leqslant j \leqslant |u|-1$,
  - $u, j \models \beta$,
  - pour tout k tel que $i \leqslant k \leqslant j-1$, on a : $u, k \models \alpha$.

On dit qu'un mot u satisfait une formule α s'il la satisfait à l'instant 0, c'est-à-dire si $u, 0 \models \alpha$. Soit α une formule de $\mathsf{LTL}(\mathsf{X},\mathsf{U})$. Les langages de mots finis ou infinis définis par α sont :

$$\mathcal{L}(\alpha) = \{u \in \Sigma^* \mid u, 0 \models \alpha\}$$
$$\mathcal{L}^\omega(\alpha) = \{u \in \Sigma^\omega \mid u, 0 \models \alpha\}$$

On dit qu'un langage L est *exprimable* (ou *définissable*) dans $\mathsf{LTL}(\mathsf{X},\mathsf{U})$ s'il existe une formule α de $\mathsf{LTL}(\mathsf{X},\mathsf{U})$ telle que $L = \mathcal{L}(\alpha)$. On dit aussi que α *définit* L.

Remarque

- Toute propriété de LTL peut être décrite comme un automate de Büchi (cf. section 2.2).
- On utilise les abréviations true pour $p \vee \neg p$ (vrai), false pour $\neg$true (faux), $\alpha \wedge \beta$ pour $\neg(\neg\alpha \vee \neg\beta)$, $\alpha \Rightarrow \beta$ pour $\neg\alpha \vee \beta$ et $\alpha \Leftrightarrow \beta$ pour $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$.
- Par définition, si β est vraie à un instant, $\alpha \,\mathsf{U}\, \beta$ l'est aussi pour tout α.
- Si $\alpha \,\mathsf{U}\, \beta$ est vraie un instant, β doit être vraie dans le futur (avoir toujours α ne suffit pas).
- Plus généralement, $\alpha \,\mathsf{U}\, \beta = \beta \vee (\alpha \wedge \mathsf{X}(\alpha \,\mathsf{U}\, \beta))$.
- La validité d'une formule de LTL(X, U) ne dépend que du futur : pour v fini et $\alpha \in$ LTL(X, U), on a $v\,u, |v| \models \alpha$ ssi $u, 0 \models \alpha$.

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

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

Remarque

Un résultat fondamental sur les langages de mots infinis est l'équivalence entre la définissabilité en LTL, logique du premier ordre, par expression rationnelle sans étoile, et reconnaissabilité par des monoïdes apériodiques.

Exercice 1.2 1. Montrer qu'on peut remplacer les deux modalités X et U par l'unique modalité XU, définie par $\alpha\,\mathsf{XU}\,\beta \equiv \mathsf{X}(\alpha\,\mathsf{U}\,\beta)$, et que les deux modalités X et U se redéfinissent à partir de XU.

2. Vérifier que $\neg\mathsf{X}\alpha = \mathsf{X}\neg\alpha$, $\mathsf{X}(\alpha\vee\beta) = \mathsf{X}\alpha\vee\mathsf{X}\beta$.

3. On peut définir le dual R (*release*) de la modalité U, par $\alpha\,\mathsf{R}\,\beta = \neg(\neg\alpha\,\mathsf{U}\,\neg\beta)$. Vérifier que $\alpha\,\mathsf{R}\,\beta = \mathsf{G}\beta\vee(\beta\,\mathsf{U}\,(\alpha\wedge\beta))$.

Exercice 1.3 Trouver une formule LTL et un automate de Büchi (voir section suivante) décrivant chacun des langages suivants. L'ensemble des propositions est $\mathrm{AP} = \{q, r\}$.

1. Ensemble des mots commençant par une lettre vérifiant q.

2. Ensemble des mots ayant un nombre infini de positions satisfaisant q.

3. Ensemble des mots ayant un nombre infini de positions satisfaisant q et un nombre infini de positions satisfaisant r.

4. Ensemble des mots contenant un unique facteur $a\,b$ tel que $q\in a$ et $r\in b$.

5. Ensemble des mots ayant un nombre *fini* de positions satisfaisant q.

Exercice 1.4 L'opérateur *weak until* W est défini par

$$\alpha\,\mathsf{W}\,\beta \stackrel{\text{def}}{=} (\alpha\,\mathsf{U}\,\beta)\vee\mathsf{G}\alpha.$$

Trouver les équivalences qui sont des tautologies parmi les suivantes.

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

Exercice 1.5 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 :

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

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

1.3 Satisfaisabilité et model-checking : approche automates

Dans cette section, on se place dans le cadre de la vérification des comportements infinis. Pour une formule LTL α, on note $\mathcal{L}(\alpha) = \mathcal{L}^\omega(\alpha)$.

Le problème du model-checking est de vérifier si tous les comportements d'un système de transitions $\mathscr{S}$ dont les états vérifient des propriétés de AP vérifient une formule $\alpha \in \mathsf{LTL}_{2^{\mathsf{AP}}}$. L'idée est de transformer $\mathscr{S}$ en un automate $\mathscr{A}_\mathscr{S}$ qui accepte les observations de $\mathscr{S}$, puis de « compiler » la formule α en automate de mots infinis $\mathscr{A}_\alpha$, tel que $\mathcal{L}(\alpha) = \mathcal{L}(\mathscr{A}_\alpha)$. On peut de même construire $\mathscr{A}_{\neg\alpha}$. On construit par ailleurs facilement un automate (fini) qui accepte le comportement du système de transitions (fini) $\mathscr{S}$. Ces deux automates permettent de tester

- si $\mathcal{L}(\mathscr{A}_\alpha) = \varnothing$, autrement dit, si la formule α est satisfaisable ;
- si $\mathcal{L}(\mathscr{A}_\mathscr{S}) \cap \mathcal{L}(\mathscr{A}_{\neg\alpha}) = \varnothing$, autrement dit, si tous les comportements du système de transitions $\mathscr{S}$ sont corrects vis-à-vis de α.

On note $\mathsf{alph}(\alpha)$ l'ensemble des propositions atomiques intervenant dans α. Comme α ne peut parler que des propositions de $\mathsf{alph}(\alpha)$, on vérifie facilement que α est satisfaisable s'il existe un mot de $2^{\mathsf{alph}(\alpha)}$ qui satisfait α (projeter chaque lettre d'un mot de 2^{AP} sur $2^{\mathsf{alph}(\alpha)}$, en effaçant les propositions de $\mathsf{AP} \setminus \mathsf{alph}(\alpha)$, ne change pas la validité de α). On peut donc supposer que $\mathsf{alph}(\alpha) = \mathsf{AP}$.

2 Automates de Büchi

Les automates de Büchi sont des automates permettant d'accepter ou de rejeter des mots ***infinis***. Un ***automate de Büchi généralisé*** sur Σ est un automate $\mathscr{A} = (S, \to, S_0, F_1, \dots, F_\ell)$. Par rapport à un automate sur les mots, la structure est la même, à ceci près qu'il y a ℓ sous-ensembles d'états finaux $F_1, \dots, F_\ell$. Lorsque $\ell = 1$, on dit que $\mathscr{A}$ est un ***automate de Büchi***. Chaque sous-ensemble d'états F_i peut être vu comme une condition d'acceptation, qui sert à restreindre les calculs acceptants.

Un ***calcul*** (ou ***run***) sur $u \in \Sigma^\omega$ est une suite infinie d'états $\sigma = s_0 s_1 \cdots \in S^\omega$ telle que $s_0 \in S_0$, et pour $i \geqslant 0$, $s_i \xrightarrow{u_i} s_{i+1}$, où $u = u_0 u_1 \cdots$, $u_i \in \Sigma$. L'ensemble des états infiniment répétés dans σ est $\inf(\sigma) = \left\{ s \in S \mid \{ i \geqslant 0 \mid s = s_i \} \text{ est infini} \right\}$. Le calcul σ est acceptant si,

$$\forall i \in \{1, \dots, \ell\}, \quad \inf(\sigma) \cap F_i \neq \varnothing,$$

c'est-à-dire que le calcul rencontre infiniment souvent un état de F_i, pour chaque i.
Le langage $\mathcal{L}(\mathscr{A})$ accepté par $\mathscr{A}$ est l'ensemble des mots qui peuvent étiqueter un run acceptant.

Exercice 2.6 Quels sont les calculs acceptants dans un automate de Büchi généralisé lorsque $\ell = 0$?

Exercice 2.7 On se place sur l'alphabet $\{a, b\}$.

1. Construire un automate de Büchi acceptant le langage des mots ayant un nombre *infini* de b.
2. Construire un automate de Büchi acceptant le langage des mots ayant un nombre *fini* de b.

On dit qu'un automate de Büchi est déterministe lorsqu'il a un seul état initial, et lorsque la relation de transition est en fait une fonction : pour tout état q et toute lettre a, il y a ***au plus*** un état p tel que $q \xrightarrow{a} p$.

Exercice 2.8 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 2.9 Trouver un langage L reconnu par deux automates différents, qui ont un nombre minimal d'états parmi tous les automates de Büchi qui reconnaissent L.

Exercice 2.10 Soit $\mathcal{A}$ un automate de Büchi généralisé. Construire un automate de Büchi $\mathcal{B}$ qui reconnaît le même langage que $\mathcal{A}$.

Exercice 2.11 Soient $\mathcal{A}_1$ et $\mathcal{A}_2$ deux automates de Büchi généralisés, reconnaissant des langages L_1 et L_2. Construire un automate de Büchi généralisé $\mathcal{B}$ qui reconnaît $\mathcal{L}_1 \cap \mathcal{L}_2$.

2.1 Calcul d'un automate de Büchi acceptant les observations d'une structure de Kripke

Soit $\mathscr{S} = (S, S_0, \to, \mathsf{AP}, \pi)$ une structure de Kripke. On définit un automate de Büchi $\mathscr{A}_\mathscr{S}$ acceptant exactement les observations générées par la structure de Kripke $\mathscr{S}$. Ce n'est pas une opération difficile : une structure de Kripke est déjà presque un automate de Büchi. Il suffit juste de déplacer les étiquettes des états aux transitions. On acceptera ensuite n'importe quel calcul infini, et pour cela il suffit de rendre chaque état final.

Formellement, si $\mathscr{S} = (S, S_0, \to, \mathsf{AP}, \pi)$, il suffit de prendre $\mathscr{A}_\mathscr{S} = (S, S_0, \to, \mathsf{AP}, S)$ avec $q \xrightarrow{a} p$ ssi on a $q \longrightarrow p$ dans $\mathscr{S}$, et $a = \pi(q)$. On vérifie que les mots acceptés par l'automate de Büchi $\mathscr{A}_\mathscr{S}$ sont bien les observations $\pi(k_0)\pi(k_1)\cdots$ des chemins dans $\mathscr{S}$.

2.2 Calcul de l'automate reconnaissant les mots qui satisfont une formule LTL

Cette partie est la plus intéressante : à partir d'une formule α de LTL, on va construire un automate de Büchi $\mathscr{A}_\alpha$ (qui sera en fait un automate de Büchi généralisé) qui reconnaît exactement les mots infinis qui vérifient la formule α (et donc, rejette ceux qui ne satisfont pas α). La présentation suivante construit directement un automate de Büchi généralisé. Il existe d'autres algorithmes, par exemple construisant des automates alternants (voir [3]).

L'algorithme suivant construit un automate non déterministe, mais qui possède une propriété remarquable : malgré son non-déterminisme, chaque mot accepté ne l'est que grâce à **un unique calcul acceptant**.

La clôture $\mathsf{cl}(\alpha)$ (dite de Fisher-Ladner) d'une formule α est, de façon informelle, l'ensemble des sous-formules de α et leur négation. Un état de l'automate $\mathscr{A}_\alpha$ sera un ensemble « consistant » de telles sous-formules. L'idée est que le langage des mots lus à partir d'un état $A \subseteq \mathsf{cl}(\alpha)$ est le langage des mots vérifiant exactement A parmi les formules de $\mathsf{cl}(\alpha)$.

- $\alpha \in \mathsf{cl}'(\alpha)$,
- $\neg\beta \in \mathsf{cl}'(\alpha) \Rightarrow \beta \in \mathsf{cl}'(\alpha)$
- $\beta \vee \gamma \in \mathsf{cl}'(\alpha) \Rightarrow \beta, \gamma \in \mathsf{cl}'(\alpha)$
- $\mathsf{X}\beta \in \mathsf{cl}'(\alpha) \Rightarrow \beta \in \mathsf{cl}'(\alpha)$
- $\beta \,\mathsf{U}\, \gamma \in \mathsf{cl}'(\alpha) \Rightarrow \mathsf{X}(\beta \,\mathsf{U}\, \gamma), \beta, \gamma \in \mathsf{cl}'(\alpha)$

Enfin, on clôt cl' par négation pour obtenir cl, en identifiant $\neg\neg\varphi$ et φ.

Remarque

- On a facilement $|\mathsf{cl}(\alpha)| = O(|\alpha|)$.

Un état q de $\mathscr{A}_\alpha$ est un sous ensemble de $\mathsf{cl}(\alpha)$ vérifiant les propriétés suivantes (consistance).

- $\forall \beta \in \mathsf{cl}(\alpha)$, $\beta \in q \Leftrightarrow \neg\beta \notin q$.
- $\forall \beta \vee \gamma \in \mathsf{cl}(\alpha)$, $\beta \vee \gamma \in q \Leftrightarrow \beta \in q$ ou $\gamma \in q$.
- $\forall \beta \,\mathsf{U}\, \gamma \in \mathsf{cl}(\alpha)$, $\beta \,\mathsf{U}\, \gamma \in q \Leftrightarrow \gamma \in q$ ou $\Big(\beta \in q$ et $\mathsf{X}(\beta \,\mathsf{U}\, \gamma) \in q\Big)$.

Les états initiaux sont ceux « vérifiant » α. Les transitions assurent que les transitions se font correctement vis-à-vis des propositions atomiques à vérifier ainsi que vis-à-vis de la modalité X. Enfin, les conditions de Büchi généralisées assurent qu'un état qui promet $\varphi \,\mathsf{U}\, \psi$ finira par vérifier ψ.

- États initiaux $= \{q \mid \alpha \in q\}$.
- $q \xrightarrow{P} q'$ si $P = q \cap \mathsf{alph}(\alpha)$ et, pour tout $\mathsf{X}\beta \in \mathsf{cl}(\alpha)$, $\mathsf{X}\beta \in q \Leftrightarrow \beta \in q'$. Note : P peut être vide.
- Soient $\gamma_i \,\mathsf{U}\, \delta_i$ les formules Until apparaissant dans $\mathsf{cl}(\alpha)$. On ajoute une condition de Büchi généralisée pour chaque formule de ce type :

$$F_i = \{q \mid \neg(\gamma_i \,\mathsf{U}\, \delta_i) \in q \text{ ou } \delta_i \in q\}$$

Théorème. 2.1 Soit $x \in (2^{\mathsf{alph}(\alpha)})^\omega$. Alors, $x \models \alpha$ si et seulement si $x \in \mathcal{L}(\mathscr{A}_\alpha)$.

Démonstration. ($\Longrightarrow$) Soit $x = x_0 x_1 \cdots$ tel que $x \models \alpha$. On construit un run acceptant de $\mathscr{A}_\alpha$ sur x. On note $q_i = \{\beta \in \mathsf{cl}(\alpha) \mid x, i \models \beta\}$. On vérifie alors immédiatement que

- on a $\alpha \in q_0$;
- pour tout i, q_i est un état ;
- pour tout i, on a $q_i \xrightarrow{x_i} q_{i+1}$.

On en déduit que $\theta = q_0 q_1 \cdots$ est un run sur x. Il reste montrer qu'il est acceptant. Si ce n'était pas le cas, il existerait une formule $\beta_i \,\mathsf{U}\, \gamma_i$ de $\mathsf{cl}(\alpha)$ telle que $\inf(\theta) \cap F_i = \varnothing$. Donc, à partir d'un certain indice j_0, θ ne rencontre plus F_i. Ceci implique que $\neg(\beta_i \,\mathsf{U}\, \gamma_i) \notin q_n$ et $\gamma_i \notin q_n$ pour $n \geqslant j_0$, donc que $\beta_i \,\mathsf{U}\, \gamma_i \in q_n$ (par définition des états) et $\gamma_i \notin q_n$. Par définition de q_n, on a donc $x, j_0 \models \beta_i \,\mathsf{U}\, \gamma_i$ et $x, n \not\models \gamma_i$ pour $n \geqslant j_0$, contradiction.

($\Longleftarrow$) Soit $x \in \mathcal{L}(\mathscr{A}_\alpha)$. Il existe un run acceptant de $\mathscr{A}_\alpha$ sur x, on le note $\theta = q_0 q_1 \cdots$. On montre par induction structurelle sur $\beta \in \mathsf{cl}(\alpha)$ que pour tout $i \geqslant 0$, on a

$$x, i \models \beta \text{ ssi } \beta \in q_i \tag{1}$$

La propriété (1) se vérifie facilement si β est une proposition atomique (par définition des transitions), si β est de la forme $\neg\gamma$ ou $\gamma \vee \delta$ (par induction et consistance des états). Si $\beta = \mathsf{X}\gamma$, alors, $x, i \models \beta$ ssi $x, i+1 \models \gamma$ ssi $\gamma \in q_{i+1}$ (par induction structurelle) ssi $\beta \in q_i$ (par définition des transitions).

Reste le cas $\beta = \gamma \mathsf{U} \delta$. Supposons que $x, i \models \beta$ et montrons que $\beta \in q_i$. Soit $k \geqslant i$ tel que $x, k \models \delta$ et pour $i \leqslant j < k$, $x, j \models \gamma$. Si $k = i$, le résultat s'obtient directement par induction structurelle,

puisqu'alors $\delta \in q_i$, donc $\beta \in q_i$ par définition des états. Sinon, on obtient le résultat par induction sur $k-i$. Si $x,i \not\models \delta$, on a $x,i \models \gamma$ et $x,i \models \mathsf{X}\beta$, car β est équivalente à $\delta \vee (\gamma \wedge \mathsf{X}\beta)$. Donc $x,i \models \gamma$ et $x,i+1 \models \beta$. Par induction structurelle, $\gamma \in q_i$. Par induction sur $k-i$, $\beta \in q_{i+1}$. Par définition des transitions, $\mathsf{X}\beta \in q_i$. Par définition des états, q_i, qui contient γ et $\mathsf{X}\beta = \mathsf{X}(\gamma \,\mathsf{U}\, \delta)$, contient aussi β.

Réciproquement, supposons que $\beta \in q_i$ et montrons que $x,i \models \beta$. Comme β est une formule Until, $\beta = \gamma_j \,\mathsf{U}\, \delta_j$ pour un certain j et il lui correspond la condition de Büchi $F_j = \{q \mid \neg\beta \in q \text{ ou } \delta \in q\}$. Comme le run sur x est acceptant, il passe infiniment souvent par F_j. Soit k le plus petit indice après i dans le run tel que $q_k \in F_j$. Si $k = i$, $q_i \in F_j$, et puisque $\beta \in q_i$, $\neg\beta \notin q_i$, donc $\delta \in q_i$: on peut conclure par induction structurelle que $x,i \models \delta$, donc $x,i \models \beta$. Si $k > i$, on raisonne par induction sur $k-i$. On a $\beta \in q_i$, et $\delta \notin q_i$ (sinon, on serait dans un état de F_j). Par construction des états, on a dans q_i les propriétés $\mathsf{X}\beta$ et γ. Par induction structurelle, $x,i \models \gamma$. Par définition des transitions, $\beta \in q_{i+1}$. Par induction sur $k-i$, $x,i+1 \models \beta$, donc $x,i \models \mathsf{X}\beta$. Finalement, $x,i \models \gamma \wedge \mathsf{X}\beta$, donc $x,i \models \beta$.

En utilisant (1) et la définition des états initiaux, on obtient $x \models \alpha$. ∎

Exercice 2.12 Cet exercice décrit un algorithme alternatif de compilation de formules LTL et automates de Büchi, sur les transitions. On part d'une formule φ pour laquelle on veut construire un automate de Büchi équivalent.

La **première phase** de l'algorithme est de mettre 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 $\bot$ (false), p, $\neg p$ (où $p \in \mathrm{AP}$) en utilisant les opérateurs $\mathsf{X}, \mathsf{U}, \mathsf{R}, \vee, \wedge$. Notez que les négations ne sont autorisées que devant une formule atomique. Dans la suite, on suppose que φ est déjà mise en forme normale négative.

On décrit maintenant la **seconde phase**, qui explique comment construire l'automate. Un état sera, comme dans l'algorithme vu en cours, à nouveau un sous-ensemble de sous-formules de φ. L'état initial est $\{\varphi\}$. Un sous-ensemble Z de formules en forme normale négative est appelé ***réduit*** si :

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

Pour définir les transitions à partir d'un état Y, on forme un graphe orienté 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 $\alpha = \alpha_1 \vee \alpha_2$, $Y \to Z \cup \{\alpha_1\}$ et $Y \to Z \cup \{\alpha_2\}$.
- Si $\alpha = \alpha_1 \wedge \alpha_2$, $Y \to Z \cup \{\alpha_1, \alpha_2\}$,
- Si $\alpha = \alpha_1 \,\mathsf{R}\, \alpha_2$, $Y \to Z \cup \{\alpha_1, \alpha_2\}$ et $Y \to Z \cup \{\mathsf{X}\alpha, \alpha_2\}$
- Si $\alpha = \alpha_1 \,\mathsf{U}\, \alpha_2$, $Y \to Z \cup \{\alpha_2\}$ et $Y \xrightarrow{\alpha} Z \cup \{\mathsf{X}\alpha, \alpha_1\}$.

Il faut noter que la dernière arête est **marquée** par α. On définit maintenant :

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

Enfin, lorsque l'ensemble de formules Z est réduit, on définit :

$$\text{next}(Z) = \{\alpha \mid \mathsf{X}\alpha \in Z\}, \quad \text{et}$$
$$\Sigma_Z = \bigcap_{p \in Z} \Sigma_p \cap \bigcap_{\neg p \in Z} \Sigma_{\neg p}.$$

Les transitions depuis un état Y sont alors : $\{Y \xrightarrow{\Sigma_Z} \text{next}(Z) \mid Z \in \text{Red}(Y)\}$, où Q désigne l'ensemble des états.

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

L'automate que l'on construit est un automate de Büchi ***généralisé sur les transitions*** : il fonctionne comme les automates de Büchi généralisés, mais les conditions d'acceptation sont des ensembles **de transitions** (et non des ensembles d'états). Un calcul partant de l'état initial est accepté si, pour chaque condition d'acceptation, il passe infiniment souvent par une transition de cette condition.

L'automate correspondant à la formule φ est défini de la façon suivante :

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

1. Soit $\mathcal{A}$ un automate de Büchi sur les transitions. Donner un automate de Büchi standard reconnaissant le même langage que $\mathcal{A}$, et justifier sa construction en prouvant que les deux automates reconnaissant bien le même langage. Comment généraliser la construction aux automates généralisés ?

2. Donner l'automate produit par l'algorithme sur les formules suivantes :

   (a) $\varphi = p \,\mathsf{U}\, \mathsf{X}q$.

   (b) $\varphi = \mathsf{G}(p \implies \mathsf{XF}q)$.

3. (Difficile) En vous inspirant de la preuve faite en cours, démontrer la correction de l'algorithme ci-dessus.

3 Satisfaisabilité, model-checking : bornes inférieures

Pour les résultats de cette partie, voir [2] et [1]. Le but de cette partie est de montrer le théorème suivant.

Théorème 1

La satisfaisabilité d'une formule de LTL(U, X), SAT(U, X) et le model-checking MC(X, U) sont PSPACE-difficiles (donc PSPACE-complets).

On commence par la réduction suivante (de façon analogue, le problème d'évaluer un circuit Booléen pour une valuation donnée est plus facile que celui de déterminer l'existence d'une valuation satisfaisante).

Théorème 2

$\mathsf{MC}(\mathsf{U},\mathsf{X}) \leqslant_P \mathsf{SAT}(\mathsf{U},\mathsf{X})$

Démonstration. Soit $\mathscr{S} = (S, s_0, \to, \pi)$ un système de transitions et α une formule. On se place dans le cas du *model-checking* positif : on se demande s'il existe un chemin dans $\mathscr{S}$ qui satisfait α (ce qui revient au même vis-à-vis de la complexité, il suffit de changer α et $\neg\alpha$). On veut construire une formule ψ de taille polynomiale, telle que ψ est satisfaisable si et seulement s'il existe une exécution dans $\mathscr{S}$ qui satisfait α.

On se place sur un alphabet de propositions plus grand. On considère pour chaque $s \in S$ une nouvelle proposition atomique q_s et on définit les formules

$$\vartheta_s = \bigwedge_{p\in\pi(s)} p \wedge \bigwedge_{p\notin\pi(s)} \neg p \wedge \mathsf{X} \bigvee_{t\,|\,s\to t} q_t$$

et

$$\eta = \mathsf{G}\left(\bigvee_{s\in S} \vartheta_s \wedge \bigwedge_{s\neq t} (q_s \Rightarrow \neg q_t) \wedge \bigvee_{s\in S} q_s\right)$$

Finalement, soit $\psi = \alpha \wedge \eta \wedge q_{s_0}$. On vérifie alors facilement qu'il existe une exécution de $\mathscr{S}$ qui satisfait α si et seulement si ψ est satisfaisable. La réduction est clairement polynomiale. ∎

Il reste à montrer que le model-checking est PSPACE-difficile. On réduit **QBF** à MC.
Problème **QBF** :

Donnée Une formule $\gamma = Q_1 x_1 Q_2 x_2 \cdots Q_n x_n \gamma_0$, où $\gamma_0 = \bigwedge_{i=1}^{m} \bigvee_{j=1}^{k_i} a_{i,j}$, où chaque Q_i est un quantificateur et où $a_{i,j}$ est une variable ou sa négation.

Question γ est-elle valide?

Théorème 3

QBF est PSPACE-complet.

Proposition 4

γ est valide s'il existe un ensemble non vide $\mathscr{V}$ de valuations de variables, tel que

- $\mathscr{V}$ est valide : $\forall x \in \mathscr{V}$, $x \models \gamma_0$.
- $\mathscr{V}$ est clos : $\forall x \in \mathscr{V}$, si $Q_i = \forall$, $\exists y \in \mathscr{V}$, $y_{<i} = x_{<i}$ et $y_i = \neg x_i$.

Théorème 5

$\mathbf{QBF} \leqslant_P \mathsf{MC}(U)$.

Démonstration. Voir [1]. À γ, on associe un système de transitions $\mathscr{S}_\gamma$ de taille polynomiale. Pour chaque variable x_i, on associe un sous-système $g_i : s_i \to x_i^t, x_i^f \to e_i$. On ajoute un sommet e_0. On relie e_0 à s_1, e_1 à s_2, etc. Arrivé e_n, on continue en enchaînant m sous-systèmes, un pour chaque disjonction de γ_0. Le premier état du premier est $e_n = f_1$, le $i^{\text{ème}}$ est de la forme $f_i \to a_{i,1}, \ldots, a_{i,k_i} \to f_{i+1}$. De l'état f_{m+1}, on peut retourner dans s_i.

On a une notion de valuation courante lorsqu'on est dans la 2ème partie de $\mathscr{S}_\gamma$. Un chemin (infini) ξ définit donc un ensemble de valuations $\mathscr{V}(\xi)$.

Une condition suffisante pour qu'un chemin définisse un ensemble de valuations clos, est que, pour un x_j quantifié universellement, si le chemin visite e_{j-1}, alors il visite à la fois x_j^t et x_j^f avant une prochaine (éventuelle) visite à e_{j-1}. Ceci peut s'écrire

$$\psi_j = \mathsf{G}(e_{j-1} \Rightarrow ((\neg s_{j-1}\ \mathsf{U}\ x_j^t) \wedge (\neg s_{j-1}\ \mathsf{U}\ x_j^f))) \text{ et } \psi_{\text{clo}} = \bigwedge_{Q_j=\forall} \psi_j$$

Les exécutions ξ satisfaisant ψ_{clos} sont telles que $\mathscr{V}(\xi)$ est clos.

On peut aussi imposer que lorsque le chemin ξ rencontre un état $a_{i,j}$, la valuation courante satisfait $a_{i,j}$ tel que défini dans la formule (x_k ou $\neg x_k$).

$$\psi_{i,j} = \mathsf{G}\Big(x_k^f \Rightarrow \mathsf{G}\neg a_{i,j} \vee \neg a_{i,j}\ \mathsf{U}\ s_k\Big) \qquad \text{si } a_{i,j} = x_k$$
$$\psi_{i,j} = \mathsf{G}\Big(x_k^t \Rightarrow \mathsf{G}\neg a_{i,j} \vee \neg a_{i,j}\ \mathsf{U}\ s_k\Big) \qquad \text{si } a_{i,j} = \neg x_k$$

On définit $\psi_{\text{corr}} = \bigwedge_{i,j} \psi_{i,j}$ et $\alpha_\gamma = \psi_{\text{clo}} \wedge \psi_{\text{corr}}$. On vérifie alors que $\mathscr{S}$ vérifie α_γ si et seulement si γ est valide, et que $\mathscr{S}_\gamma$ et α_γ sont de tailles polynomiales. ∎

Corollaire 6

MC(U) et SAT(U) sont PSPACE-difficiles. ∎

4 Vérification de modèle avec Spin

Le *model checker* Spin a été développé par G. Holzmann et son équipe, d'abord aux Bell Labs puis à la Nasa. Il permet de vérifier des propriétés de systèmes ayant un nombre fini de configurations. Un tel système (ou protocole) se compose de plusieurs processus pouvant interagir par mémoire partagée, par rendez-vous, ou par envoi et réception de messages. Afin d'être vérifié, un tel système est d'abord modélisé dans un langage appelé Promela (pour **Pro**cess **Me**ta **La**nguage). Spin est un logiciel open source disponible sur `https://spinroot.com`. Il peut être compilé sur la plupart des systèmes d'exploitation. Ce site contient également une documentation complète (référence du langage, utilisation de Spin, tutoriels, etc.).

Il faut bien noter que Promela n'est pas un langage de programmation, même si sa syntaxe est proche de celle du langage C. C'est un langage de modélisation. Ainsi, il permet de modéliser des comportements non déterministes. Les protocoles pouvant être modélisés impliquent en général plusieurs processus ou threads, qui peuvent se synchroniser à l'aide de passage de messages. En revanche, les structures de données sont finies, et les expressions n'ont pas d'effet de bord. Il n'y a pas de pointeurs, pas de type `float` ou `double`, par de bibliothèques, etc.

Spin permet de vérifier des propriétés du protocole ou système modélisé en Promela. Il se compose

- d'un **interpréteur** du langage Promela. Cet interpréteur peut exécuter le code écrit en Promela, soit de façon aléatoire lorsqu'il y a des choix non déterministes, soit de façon guidée par l'utilisateur, ou par un contre-exemple qui aurait été détecté lors d'une phase de vérification (voir point suivant).
- d'un **vérificateur**, qui déplie le système pour tester si une propriété indiquée par l'utilisateur est vraie ou non.

Les propriétés peuvent être locales (comme de simples assertions), ou des propriétés exprimables en logique LTL. Ces propriétés doivent être intégrées au code Promela, et donc, écrites elles-mêmes en Promela. Pour écrire des propriétés LTL, l'utilisateur n'a pas besoin d'écrire lui-même le code Promela, car Spin implémente l'algorithme de transformation d'une formule LTL en automate, lui-même codé en Promela. Autrement dit, Spin peut lui-même générer du code Promela pour vérifier une propriété LTL de l'utilisateur.

D'un point de vue théorique, Spin ne fait que générer, de façon très optimisée, des automates à partir du code Promela et de formules. Il implémente ensuite une variation (à nouveau optimisée) des algorithmes de *model checking* vus en cours.

Exercice 4.13 On considère le programme suivant, écrit en pseudo-langage, pour 2 processus $i = 0, 1$.

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

1. Modéliser ce programme par une structure de Kripke avec des variables, et donner une trace d'exécution montrant que l'exclusion mutuelle n'est pas satisfaite.
2. Vérifier ce fait en utilisant le code Promela et le vérificateur Spin.

Exercice 4.14

1. Modéliser par une structure de Kripke l'algorithme de Peterson pour 2 processus.
2. Modéliser en Promela cet algorithme. En utilisant Spin, vérifiez :
   - si l'exclusion mutuelle est garantie.
   - si le système peut se bloquer.
   - si la famine d'un processus est exclue, c'est-à-dire si tout processus qui veut entrer en section critique y parviendra.
   - si la propriété d'attente bornée est satisfaite, c'est-à-dire tout processus qui souhaite entrer en section critique y parviendra (absence de famine), mais en plus, le nombre d'entrées en section critique de l'autre processus pendant le temps d'attente est borné.

   Vous pouvez utiliser des assertions, des états de fin, des états de progression, des formules LTL.

Exercice 4.15 On considère la modification suivante du protocole de Peterson :

1. En quoi la modification consiste-t-elle ?
2. Ce protocole garantit-il l'exclusion mutuelle ?

```
bool turn, req[2];
byte ncrit;

active [2] proctype P()
{
  byte i = _pid;

  assert(i == 0 || i == 1);

again:
  turn = 1 - i;
  req[i] = true;
  ! (req[1 - i] && (turn == 1 - i));

  /* Entrée en section critique */
  section_critique();
  /* Sortie de la section critique */

  req[i] = false;
  goto again
}
```

Exercice 4.16 Le protocole de Peterson commence par les affectations suivantes, où `me` désigne l'identifiant du processus courant et `other` l'identifiant de l'autre processus.

```
  req[me] = true;
  turn    = other;
```

Le protocole reste-t-il correct si on inverse l'ordre de ces deux affectations ?

Exercice 4.17 Vérifier si les propriétés d'exclusion mutuelle, absence de blocage, absence de famine et attente bornée sont satisfaites par l'algorithme de Dekker pour 2 processus.

Exercice 4.18 L'algorithme distribué de Dolev, Klawe et Rodeh (1982) implémente l'élection de leader dans un anneau unidirectionnel. Chaque processus débute avec sa propre identité, qui est un entier l'identifiant de façon unique dans l'anneau. L'objectif est qu'un unique processus se déclare "leader". On veut aussi que chaque processus exécute le *même* algorithme. L'algorithme DKR est le suivant :

- Chaque processus peut travailler dans l'un de deux modes : tant qu'il espère toujours être élu leader, un processus est en *mode actif*. Dès qu'il détecte qu'il ne sera pas déclaré leader, il devient *passif*. Dans ce mode, il se contente de transmettre les messages qu'il reçoit.
- Initiallement, tous les processus sont actifs.
- Chaque processus maintient une variable, appelée `leader`, qui vaut initialement sa propre identité.

  (*a*) Un processus commence en transmettant la valeur `val` de sa variable `leader` à sont

voisin dans l'anneau, et reçoit celle de son prédécesseur, disons `val1`. Si ces valeurs sont égales, alors il se déclare leader.

(b) Sinon, il transmet la valeur `val1`, et reçoit de son prédécesseur la valeur `val2` qui a été reçue par ce prédécesseur. Si `val1` n'est pas maximal parmi `val`, `val1`, `val2`, alors le processus bascule en mode passif. Sinon, il affecte la valeur `val1` à sa variable `leader`, et continue à l'étape (a).

1. Modéliser ce protocole en Promela, pour une taille #definie `N` de l'anneau.
2. Vérifier les propriétés suivantes avec SPIN, pour de petites valeurs de `N` :
   (a) Un processus se déclarera un jour leader.
   (b) Il ne peut pas y avoir 2 processus choisis comme leaders.
   (c) Le processus leader a, à la fin, la valeur maximale des variables "leader".
   (d) Le nombre d'itérations de la boucle principale de l'algorithme est bornée par `N+1`.

En utilisant le calcul de l'intersection d'automates de Büchi généralisés, le test du vide (linéaire en déterministe et NLOGSPACE), et le fait que $\mathscr{A}_\alpha$ est de taille $2^{O(|\alpha|)}$, on obtient :

Théorème. 4.2 La satisfaisabilité d'une formule α de LTL peut se tester en temps $2^{O(|\alpha|)}$.
Le model-checking de $\mathscr{S}$ vis-à-vis de $\alpha \in \mathsf{LTL}$ se teste en $|\mathscr{S}|2^{O(|\alpha|)}$.
Ces deux problèmes sont dans la classe de complexité PSPACE (qui contient la classe NP).

Exercice 4.19 Appliquer l'algorithme pour construire les automates correspondants de Büchi généralisés qui correspondent aux formules suivantes :

1. $\mathsf{X}p$,
2. $\mathsf{G}p$,
3. $\mathsf{GF}p$,
4. $p \,\mathsf{U}\, q$,
5. $\mathsf{X}(p \,\mathsf{U}\, q)$,

5 Systèmes bien structurés

On rappelle brièvement la définition d'un système bien structuré. Elle est basée sur la notion de wqo (*well quasi-order*).

Définition 7

Un ***wqo*** (ou ***beau préordre***, ou ***beau quasi-ordre*** en français) est une relation réflexive et transitive $\preceq$ sur un ensemble X, telle que pour toute suite infinie d'éléments de X :

$$x_0, x_1, x_2, \ldots,$$

il existe $i < j$ tels que $x_i \preceq x_j$.

Par abus de langage, si la relation $\preceq$ est un wqo sur l'ensemble X, on dit que $(X, \preceq)$ est un wqo. Les exemples les plus simples de wqo sont :

- n'importe que préordre si X est fini,
- l'ensemble $\mathbb{N}$ muni de l'ordre usuel.

Par contre, $\mathbb{Z}$ n'est pas un wqo muni de l'ordre usuel.

On rappelle quelques résultats démontrés en cours. On rappelle qu'une ***antichaîne*** (pour une relation) est une suite d'éléments deux à deux incomparables.

Théorème. 5.3 — Caractérisations alternatives des wqo. On a les deux propriétés suivantes :

1. Un ordre est un wqo si et seulement si, de toute suite infinie d'éléments, on peut extraire une suite croissante, au sens large.
2. Un ordre est un wqo si et seulement s'il n'existe ni suite décroissante strictement infinie, ni antichaîne infinie.

Théorème. 5.4 — Lemme de Dickson. Soient $(A, \preceq_A)$ et $(B, \preceq_B)$ deux ensembles munis chacun d'un wqo. Sur l'ensemble $(A \times B)$, on définit la relation $\preceq_{A\times B}$ ainsi :

$$(a, b) \preceq_{A\times B} (a', b') \qquad \text{ssi} \qquad a \preceq_A a' \text{ et } b \preceq_B b'.$$

Alors, $(A \times B, \preceq_{A\times B})$ est un wqo.

On en déduit que l'ensemble $\mathbb{N}^d$ muni de l'ordre «composante à composante» est un wqo (par récurrence sur d, en appliquant le lemme de Dickson en choisissant $A = \mathbb{N}^{d-1}$ et $B = \mathbb{N}$).

Un autre lemme permettant de fournir des exemples de wqo est le lemme de Higman. On définit l'***ordre sous-mot*** de la façon suivante : si $u, v \in A^*$ sont deux mots, on note $u \sqsubseteq v$ si on peut obtenir u à partir de v en supprimant des lettres. Formellement, $u \sqsubseteq v$ si soit $u = \varepsilon$, soit $u = a_1 \cdots a_n$ avec $n \geqslant 1$, chaque a_i est une lettre et $v = v_0 a_1 v_1 \cdots v_{n-1} a_n v_n$, où les v_i sont des mots (éventuellement vides).

Théorème. 5.5 — Lemme de Higman. Soit A un alphabet fini. Alors, $(A^*, \sqsubseteq)$ est un wqo.

Un ***système bien structuré*** est donné par un triplet $(S, \to, \preceq)$ où $(S, \to)$ est un système (potentiellement infini) de transitions, et $(S, \preceq)$ est un wqo satisfaisant la **condition de monotonie suivante** : si $s \to t$ et $s \preceq s'$, alors il existe t' tel que $s' \xrightarrow{*} t'$ et $t \preceq t'$.

Des exemples classiques de systèmes bien structurés sont les suivants :

- Les systèmes d'addition de vecteurs (VAS en anglais), équivalents aux réseaux de Petri ou aux systèmes d'addition de vecteurs avec états (VASS).
- Les automates communicants avec pertes, voir exercice ci-dessous.

On a vu en cours que pour certaines variations des systèmes bien structurés, on peut utiliser un développement fini des configurations accessibles à partir d'une configuration de départ $s_0 \in S$ pour répondre aux questions suivantes :

1. Depuis s_0, le système termine-t-il ?
2. Depuis s_0, est-il possible d'atteindre un nombre infini de configurations de S ?

On s'intéresse maintenant à un autre problème, le problème de ***couverture***, défini ainsi. Étant donné un système de transitions $(S, \to)$ et deux configurations $s, t \in S$, existe-t-il un élément t' tel que $s \xrightarrow{*} t'$ et $t \preceq t'$? Autrement dit : depuis s, peut-on atteindre un élément plus grand que t ? Ce problème est très naturel lorsque, comme dans le cas des VAS, une configuration peut être interprétée comme mesurant une quantité de ressources. La question est alors : depuis s, peut-on atteindre au moins autant de ressources que celles spécifiées par la configuration t ?

L'exercice suivant montre que, sous des conditions faibles, le problème de couverture est décidable pour les systèmes bien structurés.

Exercice 5.20 — Analyse en arrière des systèmes bien structurés. Soit $(S, \to, \preceq)$ un système bien structuré. Pour $I \subseteq S$, on note $\uparrow I$ l'ensemble des éléments qui dominent un élément de I :

$$\uparrow I \stackrel{\text{def}}{=} \{ s \in S \mid \exists r \in I, r \preceq s \}.$$

On appelle $\uparrow I$ la ***clôture par le haut*** de I. Clairement, $I \subseteq \uparrow I$ par réflexivité de $\preceq$. On dit qu'un ensemble I est ***fermé par le haut*** s'il est égal à sa clôture par le haut.

1. Montrer que pour tout ensemble $I \subseteq S$ fermé par le haut, il existe un sous-ensemble fini $B \subseteq I$ tel que $I = \uparrow B$. On appelle B une ***base*** de I.
2. Soit $I_1, I_2, \ldots$ une suite infinie d'ensembles fermés par le haut, qui est en plus croissante pour l'inclusion :
$$I_1 \subseteq I_2 \subseteq I_3 \cdots$$
Montrer qu'il existe un rang à partir duquel la suite stationne, c'est-à-dire qu'il existe k tel que :
$$I_k = I_{k+1} = I_{k+2} \cdots$$

L'ensemble des prédécesseurs immédiats d'une configuration $s \in S$ est défini par :

$$\mathsf{Pre}(s) = \{ s' \in S \mid s' \to s \}$$

Si $K \subseteq S$, on note $\mathsf{Pre}(K)$ l'ensemble des prédécesseurs immédiats d'au moins un élément de K, c'est-à-dire l'ensemble $\bigcup_{s \in K} \mathsf{Pre}(s)$. On suppose dans la suite que l'on a un algorithme qui, étant donné $s \in S$ en entrée, calcule une base finie de $\uparrow\mathsf{Pre}(\uparrow s)$, notée $b(s)$. On généralise cette notation aux ensembles : $b(K)$ est une base finie de $\uparrow\mathsf{Pre}(\uparrow K)$.

3. Montrer que cette condition est vérifiée concernant les VAS.
4. Montrer que si I est un sous-ensemble de S fermé par le haut, alors $\mathsf{Pre}^*(I)$ est aussi fermé par le haut.

5. Soit $I \subseteq S$ un ensemble fermé par le haut. On définit $I_0 = I$ et inductivement, $I_{n+1} = I_n \cup b(I_n)$. Montrer qu'il existe un indice k tel que

$$\uparrow I_k = \uparrow I_{k+1} = \uparrow I_{k+2} = \cdots$$

6. On suppose également à partir de cette question, qu'étant donnés $s, t \in S$, on peut tester si $s \preceq t$. Montrer qu'on peut calculer le premier indice k à partir duquel la suite $\uparrow I_n$ stationne, et montrer que $\uparrow I_k = \uparrow \bigcup_i I_i$.

7. Montrer par récurrence que pour tout n, on a :

$$\uparrow I_n \subseteq \mathsf{Pre}^*(I).$$

8. On note $\mathsf{Pre}^n(I)$ l'ensemble obtenu en appliquant l'opérateur $\mathsf{Pre}()$ n fois à I. Montrer par récurrence que pour tout n, on a $\uparrow \mathsf{Pre}^n(I) \subseteq \uparrow I_n$.

9. Déduire des questions précédentes que $\uparrow I_k$ est exactement $\mathsf{Pre}^*(I)$.

10. Montrer que le problème de couverture est décidable pour les systèmes bien structurés, sous les hypothèses faites plus haut.

Exercice 5.21 — Automates communicants avec pertes. Un *automate à une file* sur l'alphabet A est un triplet $\mathcal{A} = (A, Q, \delta)$ où Q est son ensemble fini d'*états*, et $\delta \subseteq Q \times (\{!, ?\} \times A) \times Q$ est sa *relation de transition*. Un tel automate manipule une file FIFO de messages. De façon informelle, ! s'interprète comme « envoyer » et ? comme « recevoir » : lorsque $\mathcal{A}$ effectue une transition étiquetée par $(!, a)$, le message a est ajouté en queue de file ; une transition étiquetée $(?, a)$ n'est franchissable que si le message le plus ancien dans la file (se trouvant en tête de file) est un a, auquel cas son franchissement retire ce message de la file.

Formellement, soit $S = \{(q, x) \mid q \in Q,\ x \in A^*\}$ l'ensemble des *configurations* de $\mathcal{A}$. Posons $B = (\{!, ?\} \times A)$. Pour $b \in B$, on définit la relation $\xrightarrow{b}$ sur S par $(q, x) \xrightarrow{b} (q', x')$ si $(q, b, q') \in \delta$, et

- si $b = (!, a)$, alors $x' = x\,a$ (ajout du message a dans la file) ;
- si $b = (?, a)$, alors $x = a\,x'$ (retrait du message a de la file).

Pour $z \in B^*$, on écrit $(q, x) \xrightarrow{z} (q', x')$ si soit $z = \varepsilon$ et $(q, x) = (q', x')$, soit $z = b_1 \cdots b_n$ avec $b_i \in B$ et il existe des états $q_0 = q, q_1, \ldots, q_n = q'$ et des mots $x_0 = x, x_1, \ldots, x_n = x'$ tels que $(q_{i-1}, x_{i-1}) \xrightarrow{b_i} (q_i, x_i)$ pour $i = 1, \ldots, n$. On définit enfin $\rightarrow$ sur S : $(q, x) \rightarrow (q', x')$ s'il existe $b \in B$ tel que $(q, x) \xrightarrow{b} (q', x')$. Ainsi, $\mathcal{A}$ définit un système de transitions ordonné $\langle S, \rightarrow, = \rangle$.

1. Soit l'automate à une file $\mathcal{A} = \big(\{a\}, \{q\}, \{(q, (!, a), q), (q, (?, a), q)\}\big)$. Décrire la relation $\rightarrow$ du système $\langle S, \rightarrow, = \rangle$ associé, ainsi que le langage $\{z \in B^* \mid (q, \varepsilon) \xrightarrow{z} (q, \varepsilon)\}$.

2. Montrer que le problème de couverture est indécidable pour les systèmes $\langle S, \rightarrow, = \rangle$ associés à des automates à une file. On pourra donner le principe de la simulation d'une machine de Turing M, en utilisant la file pour y écrire des configurations de M, en choisissant convenablement l'alphabet A.

3. On considère maintenant que les files peuvent perdre des messages : formellement, on augmente la relation $\rightarrow$ en y ajoutant $(q, y) \rightarrow (q, x)$ pour tout $q \in Q$ et tous mots $x, y \in A^*$ tels que $x \sqsubseteq y$. On définit la relation $\preceq$ sur S par $(p, x) \preceq (q, y)$ si $p = q$ et $x \sqsubseteq y$. Vérifier que $\preceq$ est un bel ordre sur S et montrer que le problème de couverture est décidable pour les systèmes de transitions ordonnés $\langle S, \rightarrow, \preceq \rangle$ associés à des automates à une file à pertes.

6 Automates temporisés

On a vu en cours qu'on peut modéliser une séquence d'actions ayant lieu dans le temps par un ***mot temporisé***. Un tel mot est de la forme :

$$(a_1, t_1)(a_2, t_2)\cdots(a_n, t_n),$$

où les a_i sont des lettres, et $0 \leqslant t_1 \leqslant t_2 \leqslant \cdots \leqslant t_n$ est une séquence croissante de réels (ou rationnels) positifs ou nuls. Intuitivement, chaque t_i représente le temps auquel l'action a_i.

Les automates temporisés permettent de reconnaître des langages de mots temporisés. On rappelle informellement la définition vue en cours : un tel automate se présente comme un automate fini, mais ses calculs utilisent un ensemble fini de ***chronomètres*** (aussi appelés horloges). Ses transitions comportent :

- une garde, qui est une condition sur les chronomètres. Ces conditions sont des conjonctions de formules atomiques, et les formules atomiques autorisées sont :
  - la formule $\top$, représentant la valeur «vrai»,
  - une comparaison entre une horloge et un entier.
  - une comparaison entre une différence d'horloges et un entier relatif.

  Les comparaisons autorisées sont $<, \leqslant, =, \geqslant, >$. Une transition n'est franchissable que si la garde est satisfaite.
- un ensemble d'horloges, éventuellement vide, qui désigne l'ensemble des horloges remises à zéro en empruntant la transition.

Exercice 6.22 — Automates temporisés simples. Donner des automates temporisés reconnaissant les langages de mots temporisés suivants sur l'alphabet $\{a, b\}$.

1. L'ensemble des mots temporisés tels que deux lettres successives sont toujours séparées d'au moins une unité de temps.
2. L'ensemble des mots temporisés dont la durée totale est une unité de temps.
3. L'ensemble des mots temporisés dont la suite temporelle est exactement l'ensemble des entiers naturels.
4. L'ensemble des mots temporisés contenant deux lettres distantes d'exactement une unité de temps.

Exercice 6.23 — Suppression des contraintes diagonales. Une ***contrainte diagonale*** dans un automate temporisé est une comparaison d'une différence d'horloges à une constante, *i.e.*, une contrainte de la forme $x - y \bowtie c$, où $\bowtie$ est la comparaison.

Montrer que pour tout automate temporisé, on peut construire un autre automate temporisé reconnaissant le même langage, sans contrainte diagonale.
Indication. Traiter le cas d'une seule contrainte diagonale, et utiliser deux copies de l'automate d'origine : l'une qui assure que la contrainte diagonale est vraie, l'autre qu'elle est fausse.

Exercice 6.24 — Automates temporisés avec transitions spontanées. On considère le langage des mots temporisés sur l'alphabet $\{a\}$ de la forme $\{(a, t_1)\cdots(a, t_n) \mid \forall i,\ t_i \in \mathbb{N}\}$.

1. Montrer que ce langage est reconnaissable par un automate temporisé, si on autorise

des ε-transitions.

2. Montrer que ce langage n'est pas reconnaissable par un automate temporisé, si on n'autorise pas les ε-transitions.

Exercice 6.25 — Clôture par union et intersection. Montrer que la classe des langages de mots temporisés reconnaissables par automate temporisé est fermée :

1. par union,
2. par intersection,

en expliquant, dans chacun des cas, comment construire un automate correspondant.

Exercice 6.26 — Non-clôture par complément. On considère les deux langages de mots temporisés suivants :

- le langage L des mots temporisés sur l'alphabet $\{a, b\}$ dans lesquels il existe une lettre (a, t) mais aucune lettre $(x, t+1)$, pour $x = a$ ou b.
- le langage K des mots temporisés de la forme $(a, t_1)\cdots(a, t_n)(b, t_{n+1})\cdots(b, t_m)$, avec $t_n < 1$, c'est-à-dire dont le mot support est dans a^*b^*, où le dernier a arrive avant le temps 1.

1. Montrer que L est reconnu par un automate temporisé.
2. Montrer que K est reconnu par un automate temporisé.
3. En utilisant le langage K, montrer que L n'est reconnu paar aucun automate temporisé.

Références

[1] S. Demri and P. Schnoebelen. The complexity of propositional linear temporal logics in simple cases. *Information and Computation*, 174(1) :84–103, 2002.

[2] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3) :84–103, 1985.

[3] M. Y. Vardi. Alternating automata and program verification. In *Computer Science Today*, pages 471–485. 1995. Disponible sous http://www.cs.rice.edu/~vardi/papers/vol1000.ps.gz.