IGL502/IGL752 – Techniques de vérification et de validation

NOTES DE COURS

Michael Blondin

15 décembre 2020

Avant-propos

La rédaction de ce document a été entamée à la session d'hiver 2019 comme notes complémentaires du cours « IGL502/IGL752 – Techniques de vérification et de validation » de l'Université de Sherbrooke, dans le but d'offrir des notes gratuites, libres, en français, et taillées sur mesure pour chaque cohorte. En particulier, la structure et le contenu se basent sur le plan cadre établi par le Département d'informatique. Ainsi, ce document évoluera au gré des sessions.

Ces notes sont une synthèse et non un livre: certains passages peuvent être « expéditifs » par rapport aux explications, exemples et discussions qui surgissent en classe ou dans les capsules vidéos. Ainsi, ces notes devraient d'abord être considérées comme un complément, par ex. pour réduire, voire éliminer, la prise de notes; comme références pour réaliser les devoirs; comme matériel de révision, etc.

Les six premiers chapitres s'inspirent fortement de présentations classiques sur LTL et CTL, en grande partie de [BK08]. Le chapitre 7 suit la présentation de [And98]. Le chapitre 8 puise sa théorie dans [BEM97, EHRS00]. Le chapitre 10 se base sur un diaporama de Mickael Randour (Université de Mons) basé sur [BK08]. De façon générale, les neufs premiers chapitres sont influencés par ma participation à l'enseignement des cours *Model Checking* et *Petri nets* offerts par Javier Esparza et son groupe (Université technique de Munich).

Si vous trouvez des coquilles ou des erreurs dans le document, ou si vous avez des suggestions, n'hésitez pas à me les indiquer sur GitHub (en ajoutant un « *issue* ») ou par courriel à `michael.blondin@usherbrooke.ca`. Je remercie notamment François Ladouceur (A20) et Maxime Routhier (A20) pour l'identification de nombreuses coquilles.

Légende

Observation.

Les passages compris dans une région comme celle-ci correspondent à des observations jugées intéressantes mais qui dérogent légèrement du contenu principal.

Remarque.

Les passages compris dans une région comme celle-ci correspondent à des remarques jugées intéressantes mais qui dérogent légèrement du contenu principal.

Les passages compris dans une région colorée sans bordure comme celle-ci correspondent à du contenu qui ne sera *pas* nécessairement présenté en classe, mais qui peut aider à une compréhension plus approfondie.

Les exercices marqués par « ★ » sont considérés plus avancés que les autres.
Les exercices marqués par « ★★ » sont difficiles ou dépassent le cadre du cours.

L'icône « » dans la marge fournit un lien vers des fichiers associés au passage.
L'icône « » indique que la section est en écriture (à lire à vos risques et périls!)

Table des matières

1

Systèmes de transition

Notre but est d'automatiser la vérification de systèmes, c.-à-d. de vérifier rigoureusement qu'un système donné satisfait une certaine spécification. Afin d'accomplir cette tâche, nous devons modéliser les systèmes formellement. Les « systèmes de transition » permettent une telle modélisation. Il s'agit de graphes dirigés dont les sommets représentent symboliquement l'état interne d'un système, et dont les arcs indiquent la façon dont les états peuvent évoluer. Les états dans lesquels un tel système peut débuter sont dits « initiaux », par ex. la première ligne de code d'un programme avec ses variables initialisées.

Formellement, un *système de transition* est un triplet $\mathcal{T} = (S, \to, I)$ tel que

— S est un ensemble, dont les éléments se nomment *états*,

— $\to \subseteq S \times S$ est la *relation de transition*,

— $I \subseteq S$ est l'ensemble des *états initiaux*.

Nous disons qu'un tel système $\mathcal{T}$ est *fini* si son ensemble d'états S est fini.

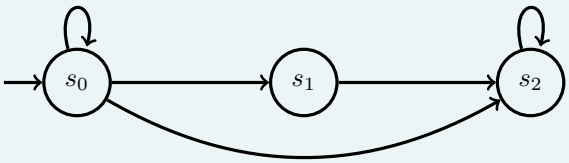
Voyons quelques exemples.

Exemple.

Considérons le système de transition fini $\mathcal{T}_1$ suivant:

$$\begin{aligned} S &\coloneqq \{s_0, s_1, s_2\}, \\ I &\coloneqq \{s_0\}, \\ \to &\coloneqq \{(s_0, s_0), (s_0, s_1), (s_0, s_2), (s_1, s_2), (s_2, s_2)\}. \end{aligned}$$

Le système $\mathcal{T}_1$ peut être représenté graphiquement ainsi:

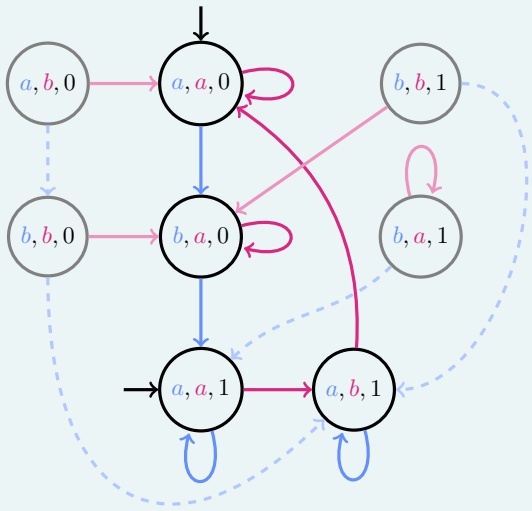


Exemple.

Considérons les deux processus suivants qui sont exécutés de façon concurrente et qui partagent une variable booléenne *tour* $\in \{0, 1\}$:

```
P0():                          P1():
   boucler:                       boucler:
a:    attendre(tour == 0)      a:    attendre(tour == 1)
b:    tour = 1                 b:    tour = 0
```

Le système de transition $\mathcal{T}_2$ naturellement associé à ce système est illustré ci-dessous. Ici, un état de la forme (x, y, z) indique que P_0 est à la ligne x, que P_1 est à la ligne y, et que *tour* $= z$. Les composantes et les transitions associées à P_0 (resp. P_1) apparaissent en cyan (resp. magenta). Les états non accessibles à partir des états initiaux sont plus pâles.



Remarque.

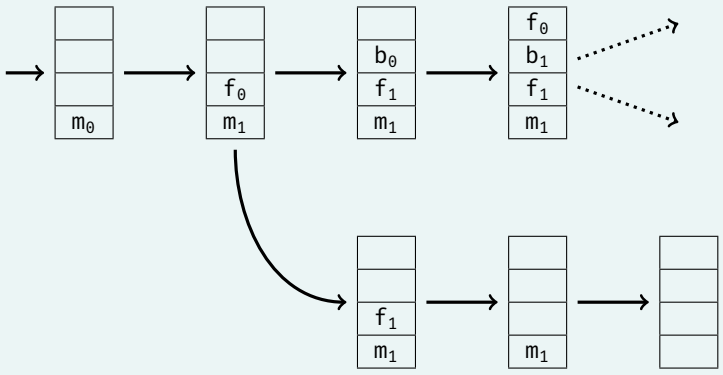
La plupart du temps, nous allons omettre les états d'un système de transition qui ne sont pas accessibles à partir d'un état initial; comme ceux plus pâles à l'exemple précédent. Toutefois, notons que déterminer si un état est accessible ou non possède un coût algorithmique.

Exemple.

Considérons le programme partiel suivant constitué de trois fonctions, dont le point d'entrée est la fonction `main`, et où « ? » dénote une valeur booléenne choisie de façon non déterministe:

```
   main():          foo():             bar():
m₀: foo()       f₀: si ?: bar()    b₀: foo()
m₁: return      f₁: return         b₁: si ?: return
                                   b₂: bar()
```

Le système de transition $\mathcal{T}_3$ associé à la pile d'appel de ce programme, avec `main` comme point d'entrée, est illustré ci-dessous.



Ce système est infini puisque, par exemple, cette exécution infinie génère une pile de profondeur arbitrairement grande:

`main()` `foo()` `bar()` `foo()` `bar()` $\cdots$.

1.1 Prédécesseurs et successeurs

Soit $\mathcal{T} = (S, \to, I)$ un système de transition. Nous écrivons $s \to t$ pour dénoter $(s, t) \in \to$. Si $s \to t$, nous disons que s est un *prédécesseur immédiat* de t, et que t est un *successeur immédiat* de s. L'ensemble des successeurs et prédécesseurs immédiats d'un état $s \in S$ sont respectivement dénotés:

$$\text{Post}(s) \coloneqq \{t \in S : s \to t\},$$
$$\text{Pre}(s) \coloneqq \{t \in S : t \to s\}.$$

Nous disons qu'un état $s \in S$ est *terminal* si $\text{Post}(s) = \emptyset$.

Nous écrivons $s \xrightarrow{*} t$ lorsque l'état t est accessible à partir de l'état s en zéro, une ou plusieurs transitions. En termes plus techniques, $\xrightarrow{*}$ est la fermeture réflexive et transitive de $\rightarrow$. En particulier, notons que $s \xrightarrow{*} s$ pour tout $s \in S$.

Si $s \xrightarrow{*} t$, nous disons que s est un *prédécesseur* de t, et que t est un *successeur* de s. L'ensemble des successeurs et prédécesseurs d'un état $s \in S$ sont respectivement dénotés:

$$\text{Post}^*(s) \coloneqq \{t \in S : s \xrightarrow{*} t\},$$
$$\text{Pre}^*(s) \coloneqq \{t \in S : t \xrightarrow{*} s\}.$$

En particulier, notons que $s \in \text{Post}^*(s)$ et $s \in \text{Pre}^*(s)$ pour tout $s \in S$.

Exemple.

Reconsidérons le système de transition $\mathcal{T}_1$. Nous avons:

$$\begin{array}{lll}
\text{Pre}(s_0) = \{s_0\} & \text{Pre}(s_1) = \{s_0\} & \text{Pre}(s_2) = \{s_0, s_1, s_2\}, \\
\text{Post}(s_0) = \{s_0, s_1, s_2\} & \text{Post}(s_1) = \{s_2\} & \text{Post}(s_2) = \{s_2\}, \\
\text{Pre}^*(s_0) = \{s_0\} & \text{Pre}^*(s_1) = \{s_0, s_1\} & \text{Pre}^*(s_2) = \{s_0, s_1, s_2\}, \\
\text{Post}^*(s_0) = \{s_0, s_1, s_2\} & \text{Post}^*(s_1) = \{s_1, s_2\} & \text{Post}^*(s_2) = \{s_2\}.
\end{array}$$

Remarque.

$\text{Post}^*(s)$ est aussi connu sous le nom d'*ensemble d'accessibilité* de s.

1.2 Chemins et exécutions

Soit $\mathcal{T} = (S, \rightarrow, I)$ un système de transition. Un *chemin fini* est une séquence finie d'états $s_0 s_1 \cdots s_n$ telle que $s_0 \rightarrow s_1 \rightarrow \cdots \rightarrow s_n$. Similairement, un *chemin infini* est une séquence infinie d'états $s_0 s_1 \cdots$ telle que $s_0 \rightarrow s_1 \rightarrow \cdots$. Un *chemin* est un chemin fini ou infini. Nous disons qu'un chemin est *initial* si $s_0 \in I$; et qu'il est maximal s'il est infini, ou s'il est fini et que son dernier état s_n est terminal, c.-à-d. si $\text{Post}(s_n) = \emptyset$. Une *exécution* de $\mathcal{T}$ est un chemin initial et maximal. Autrement dit, une exécution est une séquence qui débute dans un état initial et qui ne peut pas être étendue.

Exemple.

Reconsidérons le système de transition $\mathcal{T}_1$ du tout premier exemple. La séquence $\rho \coloneqq s_0 s_0 s_1 s_2$ est un chemin fini de $\mathcal{T}_1$ et la séquence $\rho' \coloneqq s_0 s_1 s_2 s_2 s_2 \cdots$ est un chemin infini de $\mathcal{T}_1$.

Le chemin ρ' est une exécution puisqu'il est initial et infini. Le chemin

ρ n'est pas initial puisque s_1 n'est pas initial. De plus, ρ n'est pas maximal puisque $\text{Post}(s_2) \neq \emptyset$. En fait, $\mathcal{T}_1$ ne possède *aucun* état terminal.

1.3 Structures de Kripke

Les systèmes de transition permettent de décrire les *comportements* d'un système. Nous étendons ce formalisme aux « structures de Kripke » qui ajoutent de l'information afin de raisonner sur des *propriétés* de ces comportements.

Une *structure de Kripke* est un quintuplet $\mathcal{T} = (S, \to, I, AP, L)$ tel que

— $(S, \to, I)$ est un système de transition,

— AP est un ensemble, dont les éléments sont appelés *propositions atomiques*,

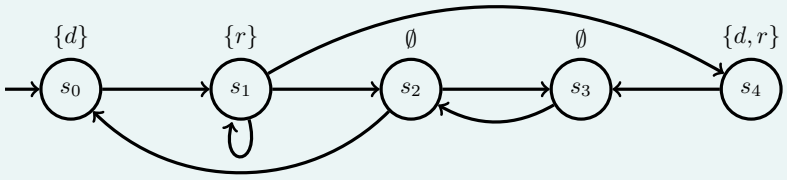
— $L \colon S \to 2^{AP}$ est une fonction dite d'*étiquetage*.

Rappelons que 2^{AP} dénote l'ensemble des sous-ensembles de AP, aussi connu sous le nom d'ensemble des parties de AP, et parfois dénoté $\mathcal{P}(AP)$.

Les propositions atomiques d'une structure de Kripke correspondent à des propriétés d'un système jugées intéressantes pour son analyse. La fonction L associe à chaque état un sous-ensemble de AP. Les propriétés décrites par le sous-ensemble $L(s)$ sont considérées satisfaites dans l'état s, et celles de $AP \setminus L(s)$ sont considérés non satisfaites dans l'état s. Par exemple, si $AP = \{p, q, r\}$ et $L(s) = \{p, q\}$, alors p et q sont vraies en s, et r est fausse en s.

Exemple.

Considérons la structure de Kripke $\mathcal{T}_4$ suivante, où $AP = \{d, r\}$:



Supposons que les propositions atomiques d et r correspondent respectivement à l'occurrence d'une *demande* et d'une *réponse* dans le contexte d'une communication entre deux machines. On peut se demander si:

1. dans chaque exécution, toute demande est éventuellement suivie d'une réponse?
2. il existe une exécution où une demande a lieu au même moment qu'une réponse?
3. chaque exécution possède un nombre infini de demandes?

La première propriété est ambiguë car il n'est pas clair si « suivie d'une réponse » permet l'occurrence d'une réponse en même temps qu'une demande.

Si cela est permis, alors la propriété est satisfaite par $\mathcal{T}_4$ puisque lorsqu'une demande est faite en s_0, elle est forcément suivie d'une réponse en s_1, et lorsqu'une demande est faite en s_4, elle est suivie au même moment d'une réponse. Si cela n'est pas permis, alors la propriété n'est pas satisfaite puisque la deuxième demande de l'exécution suivante n'est pas suivie d'une demande:

$$\rho \coloneqq s_0\ s_1\ s_4\ s_3\ s_2\ s_3\ s_2\ \cdots.$$

La seconde propriété est satisfaite, par exemple, par ρ. La troisième propriété n'est pas satisfaite puisque ρ ne possède que deux demandes.

Dans les chapitres subséquents, nous verrons comment de telles propriétés peuvent être formalisées en logique temporelle, plutôt qu'en français, afin d'éviter toute ambiguïté et d'automatiser leur vérification.

1.4 Explosion combinatoire

En général, la taille d'une structure de Kripke croît rapidement en fonction du système concret sous-jacent. Par exemple, un système de transition modélisant un programme avec n variables booléennes peut posséder jusqu'à 2^n états. Ce phénomène est connu sous le nom d'*explosion combinatoire*, ou « *state space explosion* » en anglais. Il existe plusieurs mesures pour contrer cette explosion. Tout d'abord, les outils de vérification génèrent normalement les systèmes de transitions *à la volée* plutôt qu'exhaustivement. De plus, d'autres techniques peuvent être utilisées lors de la modélisation ou de la vérification:

— *abstraction*: ignorer les données jugées non importantes pour réduire la taille du système de transition (de façon manuelle ou automatique);

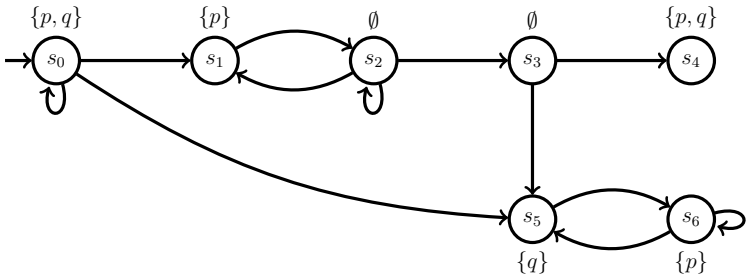
— *vérification symbolique*: utiliser des structures de données pouvant représenter plusieurs états symboliquement et manipuler directement ces structures;

— *approximations*: calculer un sous-ensemble ou sur-ensemble des états accessibles de façon symbolique afin de démontrer la présence ou l'absence d'erreurs.

Nous verrons certaines de ces approches plus tard.

1.5 Exercices

1.1) Considérons la structure de Kripke $\mathcal{T}$ suivante:



a) Décrivez Post(s), Pre(s), Post*(s) et Pre*(s) pour chaque état s.

b) Donnez la plus courte exécution de $\mathcal{T}$.

c) Donnez une exécution de $\mathcal{T}$ qui satisfait toujours p.

d) Donnez une exécution de $\mathcal{T}$ qui satisfait p infiniment souvent et qui ne satisfait pas q infiniment souvent.

1.2) Un *système de transition étiqueté* est un système de transition dont chaque transition possède une étiquette tirée d'un ensemble fini E.

a) Modélisez une variable booléenne x par un système de transition étiqueté avec $E \coloneqq \{\mathsf{x} \leftarrow \mathit{vrai}, \mathsf{x} \leftarrow \mathit{faux}, \mathsf{x} = \mathit{vrai}, \mathsf{x} = \mathit{faux}\}$.

b) Modélisez le programme ci-dessous par un système de transition étiqueté avec $E \coloneqq \{\mathsf{x} \leftarrow \mathit{vrai}, \mathsf{x} = \mathit{vrai}, \mathsf{x} = \mathit{faux}, \mathit{crit}, \mathit{noncrit}\}$.

```
   boucler
1  |  tant que x ≠ faux
   |  |  rien faire
2  |  /* section critique */
3  |  x ← vrai
4  |  /* section non critique */
```

c) Soient $\mathcal{T}_1 = (S_1, \rightarrow_1, I_1, E_1)$ et $\mathcal{T}_2 = (S_2, \rightarrow_2, I_2, E_2)$ des systèmes de transition étiquetés. Le *produit asynchrone* $\mathcal{T}_1 \parallel \mathcal{T}_2$ est le système de transition étiqueté obtenu en composant $\mathcal{T}_1$ et $\mathcal{T}_2$ avec une synchronisation faite uniquement sur leurs étiquettes communes. Construisez le produit asynchrone des deux systèmes modélisés précédemment.

Plus formellement, $\mathcal{T}_1 \parallel \mathcal{T}_2 \coloneqq (S, \rightarrow, I, E)$ où $S \coloneqq S_1 \times S_2$, $I \coloneqq I_1 \times I_2$, $E \coloneqq E_1 \cup E_2$, et

$$\begin{aligned}(s_1, s_2) \xrightarrow{e} (s_1', s_2') \iff_{\text{déf}} \; & (e \in E_1 \cap E_2 \wedge s_1 \xrightarrow{e} s_1' \wedge s_2 \xrightarrow{e} s_2') \vee \\ & (e \in E_1 \cap \overline{E_2} \wedge s_1 \xrightarrow{e} s_1' \wedge s_2 = s_2') \vee \\ & (e \in \overline{E_1} \cap E_2 \wedge s_1 = s_1' \wedge s_2 \xrightarrow{e} s_2').\end{aligned}$$

d) En général, quels sont les comportements de $T_1 \parallel T_2$ lorsque T_1 et T_2 partagent le même ensemble d'étiquettes? Et lorsqu'ils n'ont aucune étiquette en commun?

1.3) Considérons un état s d'un système de transition fini $\mathcal{T}$. Décrivez la composante fortement connexe de s en utilisant les ensembles Post(s), Pre(s), Post$^*(s)$ et/ou Pre$^*(s)$.

1.4) Soit un chemin infini $s_0 \to s_1 \to \cdots$ d'un système de transition fini $\mathcal{T}$. Expliquez pourquoi il existe forcément un indice $i \in \mathbb{N}$ tel que les états $s_i, s_{i+1}, \ldots$ appartiennent tous à une même composante fortement connexe de $\mathcal{T}$.

2

Logique temporelle linéaire (LTL)

La correction d'un système dépend de propriétés satisfaites par ses exécutions. Afin de vérifier formellement que de telles propriétés sont satisfaites, nous devons les modéliser formellement plutôt qu'en français, d'une part pour éviter toute ambiguïté, et d'autre part afin d'avoir un objet manipulable par un algorithme. La logique est l'outil idéal pour accomplir cette tâche. La logique propositionnelle n'est pas suffisante pour modéliser des propriétés intéressantes de systèmes puisqu'elle ne permet pas de raisonner sur les comportements; elle ne possède aucune notion de temps. Par exemple, la logique propositionnelle ne permet pas de décrire « chaque fois qu'un processus désire entrer dans sa section critique, il y entre éventuellement ». Afin de pallier ce problème, nous introduisons une logique temporelle; une logique qui étend la logique propositionnelle avec des opérateurs permettant de raisonner sur le temps. De telles logiques temporelles sont classiques dans le domaine de la vérification (et dans une moindre mesure en intelligence artificielle symbolique.) Dans ce chapitre, nous nous concentrons sur la logique temporelle linéaire.

2.1 Syntaxe

Soit *AP* un ensemble de propositions atomiques. La syntaxe de la *logique temporelle linéaire (LTL)* sur *AP* est définie par la grammaire suivante:

$$\varphi ::= \mathit{vrai} \mid p \mid \varphi \wedge \varphi \mid \neg\varphi \mid \mathsf{X}\varphi \mid \varphi \,\mathsf{U}\, \varphi$$

où $p \in AP$. Autrement dit, l'ensemble des formules LTL sur *AP* est défini récursivement par ces règles:

- *vrai* est une formule LTL;
- Si $p \in AP$, alors p est une formule LTL;
- Si φ_1 et φ_2 sont des formules LTL, alors $\varphi_1 \wedge \varphi_2$ est une formule LTL;
- Si φ est une formule LTL, alors $\neg\varphi$ est une formule LTL;

— Si φ est une formule LTL, alors $\mathsf{X}\varphi$ est une formule LTL;
— Si φ_1 et φ_2 sont des formules LTL, alors $\varphi_1 \mathsf{U} \varphi_2$ est une formule LTL.

Remarque.

Nous utilisons le symbole X plutôt que le symbole ◯ utlisé dans d'autres ouvrages comme [BK08]. Les opérateurs X et U se nomment respectivement ***nex**t* (« suivant ») et ***u**ntil* (« jusqu'à »).

2.2 Sucre syntaxique

Avant d'expliquer la sémantique, c.-à-d. le sens associé aux formules LTL, nous introduisons d'autres opérateurs qui rendent la lecture et l'écriture de formules plus agréable. Ceux-ci sont définis en fonction des opérateurs déjà introduits:

$$
\begin{aligned}
\mathit{faux} &\coloneqq \neg \mathit{vrai} \\
\varphi_1 \vee \varphi_2 &\coloneqq \neg(\neg\varphi_1 \wedge \neg\varphi_2) \\
\varphi_1 \rightarrow \varphi_2 &\coloneqq \neg\varphi_1 \vee \varphi_2 \\
\varphi_1 \leftrightarrow \varphi_2 &\coloneqq (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1) \\
\varphi_1 \oplus \varphi_2 &\coloneqq (\varphi_1 \wedge \neg\varphi_2) \vee (\neg\varphi_1 \wedge \varphi_2) \\
\mathsf{F}\varphi &\coloneqq \mathit{vrai}\ \mathsf{U}\ \varphi \\
\mathsf{G}\varphi &\coloneqq \neg\mathsf{F}\neg\varphi
\end{aligned}
$$

Les opérateurs $\neg, \wedge, \vee, \rightarrow, \leftrightarrow, \oplus$ sont dits *logiques*, et les opérateurs $\mathsf{X}, \mathsf{F}, \mathsf{G}, \mathsf{U}$ sont dits *temporels* (ou parfois des *modalités*).

Remarque.

Nous utilisons les symboles F et G plutôt que les symboles ◇ et □ utlisés dans d'autres ouvrages comme [BK08]. Ces opérateurs se nomment respectivement ***f**inally* (« finalement ») et ***g**lobally* (« globalement »).

2.3 Sémantique

Nous associons un sens formel aux formules LTL. Celles-ci raisonnent sur des séquences infinies qui représenteront les exécutions d'un système. Formellement, un *mot infini* sur un alphabet fini Σ est une fonction $\sigma : \mathbb{N} \to \Sigma$ que nous considérons comme une séquence infinie $\sigma(0)\sigma(1)\sigma(2)\cdots$. Nous écrivons Σ^ω pour dénoter l'ensemble des mots infinis sur Σ. Pour tous mots finis $u \in \Sigma^*$ et $v \in \Sigma^+$, nous dénotons par uv^ω le mot infini $uvvv\cdots$, donc u suivi de v répété infiniment souvent. Pour tous $\sigma \in \Sigma^\omega$ et $i \in \mathbb{N}$, nous définissons $\sigma[i..] \coloneqq \sigma(i)\sigma(i+1)\cdots$. Autrement dit, $\sigma[i..]$ est le suffixe infini de σ obtenu en débutant à l'indice i.

Exemple.

Les mots infinis a^ω, ab^ω et $(ab)^\omega$ sur alphabet $\Sigma = \{a, b\}$ correspondent respectivement: à la lettre a répétée pour toujours; à la lettre a suivie de la lettre b répétée pour toujours; et à une alternation infinie entre les lettres a et b. Par exemple, le mot ab^ω est formellement la fonction σ telle que $\sigma(0) = a$ et $\sigma(i) = b$ pour tout $i > 0$.

Dans le but de raisonner sur les exécutions de structures de Kripke, nous utiliserons l'alphabet $\Sigma = 2^{AP}$. Ainsi, une lettre est ici un *ensemble* de propositions atomiques et non une unique proposition (cela peut paraître contre-intuitif!)

Exemple.

Le mot infini $\emptyset\{p\}\{p, q\}^\omega$ sur alphabet $2^{\{p,q\}}$ correspond à une exécution où ni p, ni q ne sont vraies au premier moment; où seulement p est vraie au second moment; et où p et q sont vraies pour toujours par la suite.

Nous utiliserons la notation $\sigma \models \varphi$ afin d'indiquer que « le mot σ satisfait la formule φ ». Pour tout mot infini $\sigma \in (2^{AP})^\omega$, nous définissons cette notion ainsi:

$$
\begin{aligned}
&\sigma \models \textit{vrai} \\
&\sigma \models p && \overset{\text{déf}}{\iff} && p \in \sigma(0) \\
&\sigma \models \varphi_1 \wedge \varphi_2 && \overset{\text{déf}}{\iff} && \sigma \models \varphi_1 \wedge \sigma \models \varphi_2 \\
&\sigma \models \neg\varphi && \overset{\text{déf}}{\iff} && \sigma \not\models \varphi \\
&\sigma \models \mathsf{X}\varphi && \overset{\text{déf}}{\iff} && \sigma[1..] \models \varphi \\
&\sigma \models \varphi_1 \,\mathsf{U}\, \varphi_2 && \overset{\text{déf}}{\iff} && \exists j \geq 0 : (\sigma[j..] \models \varphi_2) \wedge (\forall 0 \leq i < j : \sigma[i..] \models \varphi_1).
\end{aligned}
$$

En mots, ces règles spécifient respectivement que:

— tout mot satisfait trivialement la formule « *vrai* »;

— un mot satisfait une proposition atomique si elle apparaît dans l'ensemble à la première position du mot;

— un mot satisfait la conjonction de formules si elle satisfait ces formules;

— un mot satisfait la négation d'une formule si elle ne la satisfait pas;

— un mot satisfait $\mathsf{X}\varphi$ s'il satisfait φ lorsqu'on retranche l'ensemble à la première position du mot;

— un mot satisfait $\varphi_1 \,\mathsf{U}\, \varphi_2$ si l'un de ses suffixes satisfait φ_2 et que les suffixes précédents satisfont tous φ_1.

La figure 2.1 illustre la sémantique de différentes formules LTL.

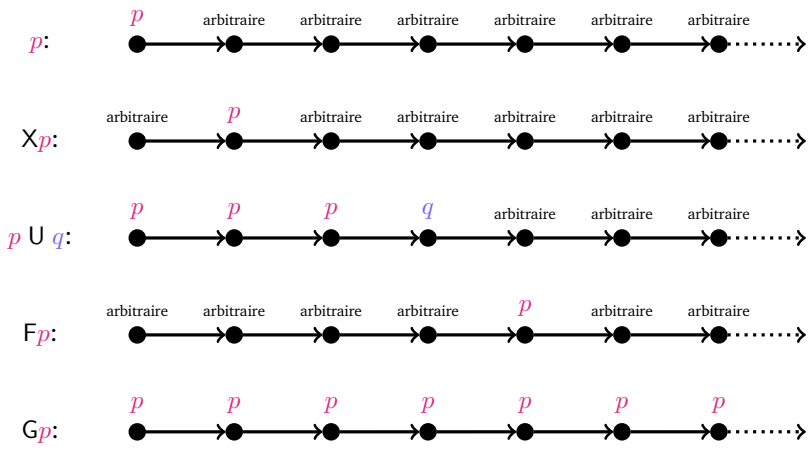


FIGURE 2.1 – Illustration de la sémantique de la logique temporelle linéaire. Chaque ligne représente un mot w infini où les cercles sont les positions de w et l'étiquette au-dessus du $i^{\text{ème}}$ cercle indique les propositions de $w(i)$.

Exemple.

Soient $AP = \{p, q\}$ et le mot infini $\sigma \in (2^{AP})^\omega$ tel que

$$\sigma \coloneqq \{p\}\ \emptyset\ \{q\}\ \{p, q\}\ (\{p\}\{q\})^\omega.$$

Nous avons:

$$\begin{aligned}
&\sigma \models p, && \sigma \not\models q,\\
&\sigma \not\models \mathsf{X}p, && \sigma \not\models \mathsf{X}q,\\
&\sigma \models \neg\mathsf{X}p, && \sigma \models \neg\mathsf{X}q,\\
&\sigma \not\models p\ \mathsf{U}\ q, && \sigma \models q\ \mathsf{U}\ p,\\
&\sigma \models \mathsf{GF}p, && \sigma \not\models \mathsf{FG}p,\\
&\sigma \models \mathsf{G}(q \rightarrow \mathsf{F}p), && \sigma \models \mathsf{FG}(p \oplus q).
\end{aligned}$$

Les opérateurs temporels F et G sont particulièrement utiles. Intuitivement, F spécifie qu'une propriété est éventuellement satisfaite au moins une fois, et G spécifie qu'une propriété est toujours satisfaite. Il est possible de démontrer formellement que leur sémantique correspond bien à celle illustrée à la figure 2.1:

Proposition 1. *Soit AP un ensemble de propositions atomiques. Pour tout* $\sigma \in (2^{AP})^\omega$ *et pour toute formule LTL* φ *sur AP, nous avons:*

(a) $\sigma \models \mathsf{F}\varphi \iff \exists j \geq 0 : \sigma[j..] \models \varphi$, *et*

(b) $\sigma \models \mathsf{G}\varphi \iff \forall j \geq 0 : \sigma[j..] \models \varphi$.

Démonstration.

(a)

$$\begin{aligned}
\sigma \models \mathsf{F}\varphi &\iff \sigma \models \mathit{vrai}\ \mathsf{U}\ \varphi && \text{(par déf. de F)}\\
&\iff \exists j \geq 0 : (\sigma[j..] \models \varphi) \wedge (\forall 0 \leq i < j : \sigma[i..] \models \mathit{vrai}) && \text{(par déf. de U)}\\
&\iff \exists j \geq 0 : \sigma[j..] \models \varphi && \text{(par déf. de } \mathit{vrai}\text{)}.
\end{aligned}$$

(b)

$$\begin{aligned}
\sigma \models \mathsf{G}\varphi &\iff \sigma \models \neg\mathsf{F}\neg\varphi && \text{(par déf. de G)}\\
&\iff \neg(\sigma \models \mathsf{F}\neg\varphi) && \text{(par déf. de } \neg\text{)}\\
&\iff \neg(\exists j \geq 0 : \sigma[j..] \models \neg\varphi) && \text{(par (a))}\\
&\iff \neg(\exists j \geq 0 : \neg(\sigma[j..] \models \varphi)) && \text{(par déf. de } \neg\text{)}\\
&\iff \forall j \geq 0 : \neg(\neg(\sigma[j..] \models \varphi)) && \text{(loi de De Morgan)}\\
&\iff \forall j \geq 0 : \sigma[j..] \models \varphi && \text{(double négation).}
\end{aligned}$$

□

2.4 Équivalences

L'ensemble des mots qui satisfont une formule LTL φ sur *AP* se dénote par:

$$[\![\varphi]\!] \coloneqq \left\{\sigma \in (2^{AP})^{\omega} : \sigma \models \varphi\right\}.$$

Certains concepts de la logique propositionnelle s'étendent naturellement à la logique temporelle linéaire. Soient φ et φ' deux formules LTL. Nous disons que:

— φ est une *tautologie* si $[\![\varphi]\!] = (2^{AP})^{\omega}$;
— φ n'est *pas satisfaisable* si $[\![\varphi]\!] = \emptyset$;
— φ et φ' sont *équivalentes* si $[\![\varphi]\!] = [\![\varphi']\!]$.

Nous écrivons $\varphi \equiv \varphi'$ pour dénoter que deux formules sont équivalentes. Notons que toute tautologie est équivalente à la formule *vrai*, et que toute formule non satisfaisable est équivalente à la formule *faux*.

Nous répertorions quelques équivalences entre formules LTL qui peuvent simplifier la spécification ou l'analyse de propriétés.

2.4.1 Distributivité

Les opérateurs temporels se distribuent ainsi sur les opérateurs logiques:

$$\begin{aligned}
\mathsf{X}(\varphi_1 \vee \varphi_2) &\equiv \mathsf{X}\varphi_1 \vee \mathsf{X}\varphi_2,\\
\mathsf{X}(\varphi_1 \wedge \varphi_2) &\equiv \mathsf{X}\varphi_1 \wedge \mathsf{X}\varphi_2,\\
\mathsf{F}(\varphi_1 \vee \varphi_2) &\equiv \mathsf{F}\varphi_1 \vee \mathsf{F}\varphi_2,\\
\mathsf{G}(\varphi_1 \wedge \varphi_2) &\equiv \mathsf{G}\varphi_1 \wedge \mathsf{G}\varphi_2,\\
(\varphi_1 \wedge \varphi_2)\ \mathsf{U}\ \psi, &\equiv (\varphi_1\ \mathsf{U}\ \psi) \wedge (\varphi_2\ \mathsf{U}\ \psi),\\
\psi\ \mathsf{U}\ (\varphi_1 \vee \varphi_2), &\equiv (\psi\ \mathsf{U}\ \varphi_1) \vee (\psi\ \mathsf{U}\ \varphi_2).
\end{aligned}$$

Remarque.

En général, la formule $\mathsf{F}(\varphi_1 \wedge \varphi_2)$ n'est pas équivalente à $\mathsf{F}\varphi_1 \wedge \mathsf{F}\varphi_2$. Par exemple, pour $AP = \{p, q\}$ et $\sigma \coloneqq (\{p\}\{q\})^\omega$, nous avons:

$$\sigma \not\models \mathsf{F}(p \wedge q),$$
$$\sigma \models \mathsf{F}p \wedge \mathsf{F}q.$$

Similairement, G ne se distribue par sur la disjonction:

$$\sigma \models \mathsf{G}(p \vee q),$$
$$\sigma \not\models \mathsf{G}p \vee \mathsf{G}q.$$

2.4.2 Dualité

La négation interagit de cette façon avec les opérateurs temporels:

$$\neg\mathsf{X}\varphi \equiv \mathsf{X}\neg\varphi,$$
$$\neg\mathsf{F}\varphi \equiv \mathsf{G}\neg\varphi,$$
$$\neg\mathsf{G}\varphi \equiv \mathsf{F}\neg\varphi.$$

2.4.3 Idempotence

Les opérateurs temporels satisfont ces règles d'idempotence:

$$\mathsf{FF}\varphi \equiv \mathsf{F}\varphi,$$
$$\mathsf{GG}\varphi \equiv \mathsf{G}\varphi,$$
$$\varphi_1 \,\mathsf{U}\, (\varphi_1 \,\mathsf{U}\, \varphi_2) \equiv \varphi_1 \,\mathsf{U}\, \varphi_2.$$

2.4.4 Absorption

Les opérateurs temporels satisfont ces règles d'absorption:

$$\mathsf{FGF}\varphi \equiv \mathsf{GF}\varphi,$$
$$\mathsf{GFG}\varphi \equiv \mathsf{FG}\varphi.$$

Ainsi, il n'existe que quatre combinaisons des opérateurs F et G: $\{\mathsf{F}, \mathsf{G}, \mathsf{FG}, \mathsf{GF}\}$. Ceux-ci spécifient respectivement « éventuellement », « toujours », » éventuellement toujours » et « infiniment souvent ».

2.5 Propriétés d'un système

Voyons maintenant comment établir la connexion entre système et spécification, c.-à-d. entre structures de Kripke et formules LTL. Considérons une structure de

Kripke $\mathcal{T} = (S, \to, I, AP, L)$. La *trace* d'une exécution infinie $s_0 s_1 \cdots$ de $\mathcal{T}$ est définie par

$$\text{trace}(s_0 s_1 \cdots) \coloneqq L(s_0)L(s_1)\cdots .$$

Autrement dit, la trace d'une exécution est obtenue en remplaçant chaque état par son étiquette qui indique les propositions atomiques qu'il satisfait.

L'ensemble des traces de $\mathcal{T}$ est dénoté

$$\text{Traces}(\mathcal{T}) \coloneqq \{\text{trace}(w) : w \text{ est une exécution infinie de } \mathcal{T}\}.$$

Nous disons que $\mathcal{T}$ *satisfait* une propriété LTL φ sur *AP* si:

$$\text{Traces}(\mathcal{T}) \subseteq [\![\varphi]\!].$$

En mots: $\mathcal{T}$ satisfait φ si la trace de *chacune* de ses exécutions infinies satisfait φ.

Exemple.

Considérons la structure de Kripke $\mathcal{T}$ illustrée à la figure 2.2. Nous avons:

$$\mathcal{T} \models p, \qquad \mathcal{T} \models \mathsf{GF}p,$$
$$\mathcal{T} \not\models \mathsf{G}p, \qquad \mathcal{T} \not\models (\neg q)\ \mathsf{U}\ q.$$

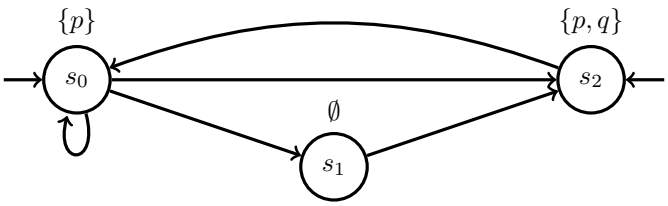
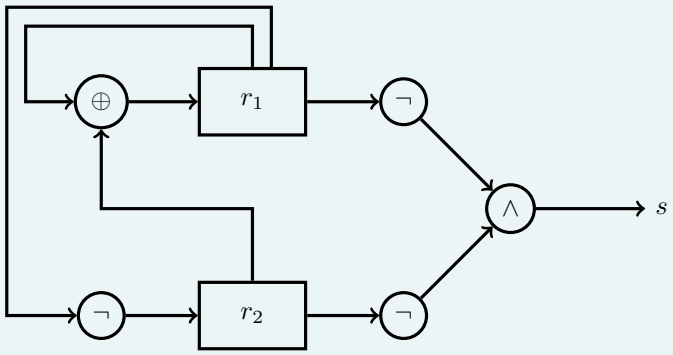


FIGURE 2.2 – Une structure de Kripke avec $AP = \{p, q\}$.

Notons qu'il est possible qu'une propriété *et* sa négation ne soient pas satisfaites par un système. Par exemple, pour la structure de Kripke de la figure 2.2, nous avons $\mathcal{T} \not\models p \wedge q$ *et* $\mathcal{T} \not\models \neg(p \wedge q)$ puisque toute exécution qui débute en s_0 ne satisfait pas $p \wedge q$ et toute exécution qui débute en s_2 ne satisfait pas $\neg(p \wedge q)$. En particulier, il faut donc tenir compte de tous les états initiaux.

Exemple.

Considérons le circuit logique $\mathcal{C}$ ci-dessous qui émet un bit à chaque trois cycles (inspiré de l'exemple 5.11 de [BK08]). Ce circuit est constitué des registres r_1 et r_2, et de la sortie s.



Cherchons à spécifier que la sortie de $\mathcal{C}$ vaut 1 à chaque cycle divisible par trois: $1, 0, 0, 1, 0, 0, \ldots$. Soit $AP = \{s\}$ où la proposition atomique s indique que la sortie du circuit vaut 1. Nous spécifions d'abord que la sortie de $\mathcal{C}$ doit valoir 1 au moins une fois par cycle de taille trois:

$$\varphi_{\geq 1} \coloneqq \mathsf{G}(s \vee \mathsf{X}s \vee \mathsf{XX}s).$$

Nous spécifions ensuite que la sortie vaut 1 au plus une fois par cycle de taille trois:

$$\varphi_{\leq 1} \coloneqq \mathsf{G}(s \rightarrow (\mathsf{X}\neg s \wedge \mathsf{XX}\neg s)).$$

La formule $\psi = s \wedge \varphi_{\geq 1} \wedge \varphi_{\leq 1}$ spécifie que la sortie de $\mathcal{C}$ évolue bien de la façon suivante: $1, 0, 0, 1, 0, 0, \ldots$. Notons que $\mathcal{C}$ satisfait φ si et seulement si ses registres sont initialisés à $r_1 = 0$ et $r_2 = 0$.

2.6 Types de propriétés

Nous discutons brièvement des types de propriétés pouvant être modélisées en LTL et plus généralement à l'aide de mots infinis.

2.6.1 Invariants

Un *invariant* est une propriété qui doit être satisfaite à tout moment d'une exécution. En LTL, un invariant s'écrit sous la forme « $\mathsf{G}\varphi$ », où φ est une formule propositionnelle. Par exemple, l'*exclusion mutuelle* de deux processus est un invariant qui s'écrit sous la forme $\mathsf{G}\neg(c_1 \wedge c_2)$. Une telle propriété peut être démontrée ou infirmée à l'aide d'un parcours de graphe sur la structure de Kripke. De plus, un témoin de l'invalidité d'un invariant peut être identifié algorithmiquement, par exemple à l'aide d'un parcours en profondeur où l'on utilise une pile afin de stocker un chemin qui mène à un état qui enfreint la propriété (par ex. voir [BK08, algo. 4, p. 110]). Ainsi, on peut automatiser la vérification d'un

invariant pour une structure de Kripke donnée, et retourner une « explication » du bogue le cas échéant.

2.6.2 Sûreté

Informellement, une *propriété de sûreté* est une propriété qui indique que « rien de mauvais ne se produit ». Une propriété de sûreté qui n'est pas satisfaite peut être réfutée par un préfixe fini d'une trace. Par exemple, reconsidérons la structure de Kripke $\mathcal{T}$ illustrée à la figure 2.2 et la propriété $\varphi = \mathsf{G}(p \vee q)$. La structure $\mathcal{T}$ ne satisfait pas φ tel que démontré par la « trace finie »: $\{p, q\}\{p\}\emptyset$. En effet, ni p, ni q sont vraies à la troisième position, donc la suite n'a pas d'importance. En particulier, les invariants sont des propriétés de sûreté.

Formellement, nous disons qu'une propriété φ sur *AP* est une *propriété de sûreté* si tout mot infini $\sigma \notin [\![\varphi]\!]$ satisfait:

$$\exists i \in \mathbb{N},\ \forall \sigma' \in (2^{AP})^{\omega},\ \sigma[0..i]\sigma' \notin [\![\varphi]\!].$$

En mots, cela signifie que si un mot infini σ enfreint φ, alors il doit posséder un préfixe fini qui enfreint φ peu importe la façon dont on l'étend. Intuitivement, ce préfixe $\sigma[0..i]$ est celui qui « témoigne » de l'erreur.

2.6.3 Vivacité

Informellement, une *propriété de vivacité* est une propriété dont les bons comportements se manifestent « vers l'infini ». Contrairement aux propriétés de sûreté, une propriété de vivacité ne peut pas être réfutée à l'aide d'un préfixe fini. Voici quelques exemples de propriétés de vivacité:

(a) $\mathsf{GF}p$: p est satisfaite infiniment souvent;

(b) $\mathsf{FG}p$: p est éventuellement toujours satisfaite (*propriété de persistance*);

(c) $\mathsf{G}(p \rightarrow \mathsf{F}q)$: lorsque p est satisfaite, q est éventuellement satisfaite.

Afin d'illustrer la vivacité, considérons la propriété (a). Si elle est enfreinte par un système $\mathcal{T}$, on ne peut pas s'en convaincre à l'aide d'un préfixe fini w. Par exemple, considérons $w = \emptyset\emptyset\{q\}\emptyset$. Bien que w ne contienne aucune occurrence de p, il est possible qu'après quatre transitions, $\mathcal{T}$ visite infiniment souvent p. En fait, ici on ne peut même pas se convaincre que $\mathcal{T}$ satisfait la propriété à l'aide d'un préfixe fini. Par exemple, bien que $w' \coloneqq \{p\}\{p, q\}\{p\}$ visite p plusieurs fois, il y a peut-être moyen de compléter w' dans $\mathcal{T}$ sans plus jamais visiter p.

Formellement, nous disons qu'une propriété φ sur *AP* est une *propriété de vivacité* si pour tout mot fini $w \in (2^{AP})^*$, il existe un mot infini $\sigma \in (2^{AP})^{\omega}$ tel que $w\sigma \in [\![\varphi]\!]$. Plus informellement, cela signifie que toute « trace finie » w peut être étendue de telle sorte à satisfaire φ.

2.7 Équité

Certaines propriétés d'un système concurrent peuvent parfois être trivialement enfreintes en raison d'exécutions inéquitables qui priorisent systématiquement certains processus. Par exemple, considérons un système constitué de n processus $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n$. Pour tout $1 \leq i \leq n$, définissons les propositions atomiques:

$$p_i \coloneqq \mathcal{P}_i \text{ peut exécuter une instruction,}$$
$$q_i \coloneqq \mathcal{P}_i \text{ exécute une instruction.}$$

Considérons la trace $\sigma \coloneqq \{p_1, q_1, p_2\}^\omega$. Cette trace correspond à une exécution où seules les instructions du processus $\mathcal{P}_1$ sont exécutées, bien que les instructions du processus $\mathcal{P}_2$ soient exécutables. En particulier, nous avons:

$$\sigma \models \mathsf{FG}(p_2 \wedge \neg q_2).$$

Autrement dit, à partir d'un certain point, le processus $\mathcal{P}_2$ est toujours exécutable, mais n'est jamais exécuté. En pratique, cela pourrait correspondre à un ordonnanceur qui ignore complètement un processus pour aucune raison valable. Normalement, ce type de comportement n'est pas réaliste et il faut donc faire l'hypothèse d'une certaine notion d'« équité » entre les processus. Nous survolons deux telles notions d'équité.

Équité faible. Posons $\varphi_i \coloneqq \mathsf{FG}(p_i \wedge \neg q_i)$ pour tout $1 \leq i \leq n$. Nous disons qu'une trace $\sigma \in (2^{AP})^\omega$ d'un système à n processus est *faiblement équitable* ssi:

$$\sigma \models \bigwedge_{i=1}^{n} \neg\varphi_i.$$

Autrement dit, σ est faiblement équitable ssi pour chaque processus $\mathcal{P}_i$, ce n'est pas le cas qu'éventuellement celui-ci est toujours exécutable et jamais exécuté.

Il est possible de spécifier que toutes les traces faiblement équitables d'un système doivent satisfaire une propriété ψ, et d'ignorer les traces non équitables avec cette formule:

$$\left(\bigwedge_{i=1}^{n} \neg\varphi_i\right) \rightarrow \psi.$$

Ainsi, la vérification d'une propriété ψ, sous hypothèse que les exécutions sont faiblement équitables, ne requiert pas de machinerie supplémentaire. En effet, la formule résultante est également une formule LTL.

Notons que:

$$\begin{aligned}
\neg\varphi_i &= \neg\mathsf{FG}(p_i \wedge \neg q_i) && \text{(par définition de } \varphi_i\text{)} \\
&\equiv \neg(\mathsf{FG}p_i \wedge \mathsf{FG}\neg q_i) && \text{(par distributivité de FG, voir exercice 2.9))} \\
&\equiv \neg\mathsf{FG}p_i \vee \neg\mathsf{FG}\neg q_i && \text{(par la loi de De Morgan)} \\
&\equiv \neg\mathsf{FG}p_i \vee \mathsf{GF}q_i && \text{(par dualité)} \\
&\equiv \mathsf{FG}p_i \rightarrow \mathsf{GF}q_i && \text{(par défininition de } \rightarrow\text{).}
\end{aligned}$$

Ainsi, de façon équivalente, qui est celle utilisée dans d'autres ouvrages comme le manuel de référence [BK08], une trace σ est faiblement équitable ssi:

$$\sigma \models \bigwedge_{i=1}^{n} (\mathsf{FG}p_i \rightarrow \mathsf{GF}q_i).$$

Équité forte. Considérons la trace $(\{p_1, q_1, p_2\}\{p_1, q_1\})^\omega$. Cette trace est faiblement équitable (vérifiez-le!) Toutefois, bien que le processus $\mathcal{P}_2$ soit exécutable infiniment souvent, il n'est jamais exécuté. La notion d'équité faible peut être raffinée afin d'éliminer ce type de traces. Posons $\varphi'_i := (\mathsf{GF}p_i \wedge \mathsf{FG}\neg q_i)$ pour tout $1 \leq i \leq n$. Nous disons qu'une trace σ est *fortement équitable* ssi

$$\sigma \models \bigwedge_{i=1}^{n} \neg\varphi'_i.$$

Autrement dit, une trace n'est *pas* fortement équitable ssi un processus est exécutable infiniment souvent, mais n'est exécuté qu'un nombre fini de fois.

Comme dans le cas de l'équité faible, la vérification d'une propriété ψ, sous hypothèse que les exécutions sont fortement équitables, ne requiert pas de machinerie supplémentaire; il suffit de considérer la formule $(\bigwedge_{i=1}^{n} \neg\varphi'_i) \rightarrow \psi$.

Notons que:

$$\begin{aligned}
\neg\varphi'_i &= \neg(\mathsf{GF}p_i \wedge \mathsf{FG}\neg q_i) && \text{(par définition de } \varphi'_i\text{)} \\
&\equiv \neg\mathsf{GF}p_i \vee \neg\mathsf{FG}\neg q_i && \text{(par la loi de De Morgan)} \\
&\equiv \mathsf{GF}p_i \rightarrow \neg\mathsf{FG}\neg q_i && \text{(par définition de } \rightarrow\text{)} \\
&\equiv \mathsf{GF}p_i \rightarrow \mathsf{GF}q_i && \text{(par dualité).}
\end{aligned}$$

Ainsi, de façon équivalente, qui est celle utilisée par d'autres ouvrages comme le manuel de référence [BK08], une trace σ est fortement équitable ssi:

$$\sigma \models \bigwedge_{i=1}^{n} (\mathsf{GF}p_i \rightarrow \mathsf{GF}q_i).$$

Remarque.

L'unique propriété linéaire qui est à la fois de sûreté et de vivacité est la propriété triviale *vrai*. Toute autre propriété linéaire est de sûreté, de vivacité, ou l'intersection d'une propriété de sûreté et de vivacité.

2.8 Spin et Promela

Spin est un outil qui permet de vérifier automatiquement des systèmes concurrents finis décrits dans le langage Promela et dont les propriétés sont spécifiées

par des assertions ou des formules LTL (avec ou sans équité). Le développement de cet outil a débuté en 1980 dans le « groupe Unix » de Bell Labs. Spin est maintenu à jour et distribué sous licence libre. Le langage Promela supporte des types booléens et entiers, les tableaux, les structures, les canaux de communication (finis ou de type *rendez-vous*), les processus (avec identités), les blocs atomiques, différents types de flôt de contrôles (boucle, sélection, saut, etc.) Sa syntaxe s'apparente à celle de C, à l'exception notable de l'ajout du non déterminisme qui permet de modéliser des comportements imprévisibles. À l'interne, Spin convertit le code Promela vers une structure de Kripke.

Par exemple, reconsidérons l'algorithme d'exclusion mutuelle de Lamport à deux processus tel qu'introduit au début du cours:

Processus $\mathcal{A}$

```
1.  tant que vrai:
2.    x = vrai
3.    tant que y: rien faire
4.    # section critique
5.    x = faux
```

Processus $\mathcal{B}$

```
1.  tant que vrai:
2.    y = vrai
3.    si x alors:
4.      y = faux
5.      tant que x: rien faire
6.      aller à 2
7.    # section critique
8.    y = faux
```

Cet algorithme se modélise ainsi dans le langage Promela:

```
bool x = false;
bool y = false;

init
{
  atomic
  {
    run A()
    run B()
  }
}

proctype A()
{
  do
    :: true ->
enter:
       x = true
wait:
       do
         :: y    -> skip
         :: else -> break
       od
```

```
critical:
       skip
leave:
       x = false
  od
}

proctype B()
{
  do
    :: true ->
enter:
       y = true
wait:
       if
         :: x ->
            y = false

            do
              :: x    -> skip
              :: else -> break
            od

            goto enter

         :: else -> goto critical
       fi
critical:
       skip
leave:
       y = false
  od
}
```

Ces propriétés LTL pertinentes à l'analyse de l'algorithme:

$\varphi_1 \coloneqq \neg(\mathsf{F}(\text{« } \mathcal{A} \textit{ est dans sa section critique } \text{» } \wedge \text{« } \mathcal{B} \textit{ est dans sa section critique } \text{»}))$

$\varphi_2 \coloneqq \mathsf{G}(\text{« } \mathcal{B} \textit{ désire entrer dans sa section critique } \text{» } \rightarrow \mathsf{F}\text{ « } \mathcal{B} \textit{ est dans sa section critique } \text{»})$

se spécifient respectivement ainsi:

```
  ltl p1 { !(<> (A@critical && B@critical)) }
  ltl p2 {   [] (B@enter -> <> B@critical)  }
```

Spin peut vérifier que la propriété φ_1 est satisfaite. Cependant, φ_2 ne l'est pas, même sous équité forte, puisque dans cette exécution infinie le processus

$\mathcal{B}$ atteint sa ligne 2 infiniment souvent sans jamais atteindre sa ligne 7:

$$
\begin{aligned}
(\mathcal{A}_1, \mathcal{B}_1, \mathsf{x} = \mathit{faux}, \mathsf{y} = \mathit{faux}) &\xrightarrow{\mathcal{A}} (\mathcal{A}_2, \mathcal{B}_1, \mathsf{x} = \mathit{faux}, \mathsf{y} = \mathit{faux}) \\
&\xrightarrow{\mathcal{A}} (\mathcal{A}_3, \mathcal{B}_1, \mathsf{x} = \mathit{vrai}, \mathsf{y} = \mathit{faux}) \\
&\xrightarrow{\mathcal{B}} (\mathcal{A}_3, \mathcal{B}_2, \mathsf{x} = \mathit{vrai}, \mathsf{y} = \mathit{faux}) \\
&\xrightarrow{\mathcal{B}} (\mathcal{A}_3, \mathcal{B}_3, \mathsf{x} = \mathit{vrai}, \mathsf{y} = \mathit{vrai}) \\
&\xrightarrow{\mathcal{B}} (\mathcal{A}_3, \mathcal{B}_4, \mathsf{x} = \mathit{vrai}, \mathsf{y} = \mathit{vrai}) \\
&\xrightarrow{\mathcal{B}} (\mathcal{A}_3, \mathcal{B}_5, \mathsf{x} = \mathit{vrai}, \mathsf{y} = \mathit{faux}) \\
&\xrightarrow{\mathcal{A}} (\mathcal{A}_4, \mathcal{B}_5, \mathsf{x} = \mathit{vrai}, \mathsf{y} = \mathit{faux}) \\
&\xrightarrow{\mathcal{A}} (\mathcal{A}_5, \mathcal{B}_5, \mathsf{x} = \mathit{vrai}, \mathsf{y} = \mathit{faux}) \\
&\xrightarrow{\mathcal{A}} (\mathcal{A}_1, \mathcal{B}_5, \mathsf{x} = \mathit{faux}, \mathsf{y} = \mathit{faux}) \\
&\xrightarrow{\mathcal{B}} (\mathcal{A}_1, \mathcal{B}_6, \mathsf{x} = \mathit{faux}, \mathsf{y} = \mathit{faux}) \\
&\xrightarrow{\mathcal{B}} (\mathcal{A}_1, \mathcal{B}_2, \mathsf{x} = \mathit{faux}, \mathsf{y} = \mathit{faux}) \\
&\xrightarrow{\mathcal{A}} (\mathcal{A}_2, \mathcal{B}_2, \mathsf{x} = \mathit{faux}, \mathsf{y} = \mathit{faux}) \\
&\xrightarrow{\mathcal{A}} (\mathcal{A}_3, \mathcal{B}_2, \mathsf{x} = \mathit{vrai}, \mathsf{y} = \mathit{faux}) \\
&\xrightarrow{\mathcal{B}} \dots
\end{aligned}
$$

2.9 Étude de cas: protocole de Needham-Schroeder

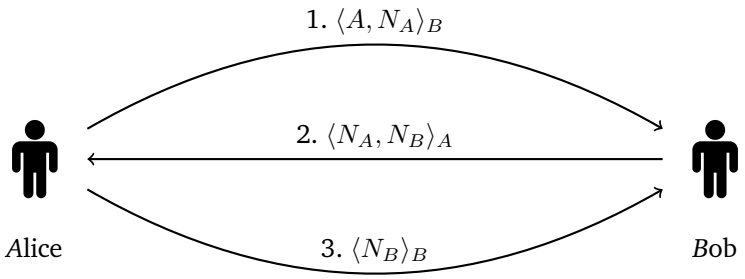
Considérons le scénario où deux entités, Alice et Bob, désirent établir un secret commun sur un réseau non sûr, avec un accès à la cryptographie à clé publique. Plus en détails, Alice et Bob possèdent respectivement un secret N_A et N_B (par ex. des entiers) et veulent s'échanger ces secrets sans qu'une autre entité l'apprenne. Chaque entité peut chiffrer un message m avec la clé publique de l'entité X, que nous dénotons $\langle m \rangle_X$, et uniquement X peut déchiffrer $\langle m \rangle_X$.

Le protocole de Needham-Schroeder [1] tente d'accomplir cette tâche ainsi:

1. Alice envoie $\langle A, N_A \rangle_B$, c.-à-d. son identité et son secret, à Bob;
2. Bob déchiffre le message, apprend l'identité d'Alice et lui envoie $\langle N_A, N_B \rangle_B$;
3. Alice déchiffre le message, se convainc qu'elle interagit bien avec Bob puisqu'il a pu lui renvoyer son secret, et elle apprend ainsi le secret de Bob; Alice envoie $\langle N_B \rangle_B$ à Bob, ce qui le convainc qu'il interagit bien avec Alice.

Schématiquement, le protocole devrait donc se comporter ainsi:

1. Il s'agit ici de la variante décrite dans [Mer01]



Cependant, nous supposons qu'une entité adversariale, Eve, peut intercepter et envoyer des messages sur le réseau. Ainsi, le protocole est de la forme:

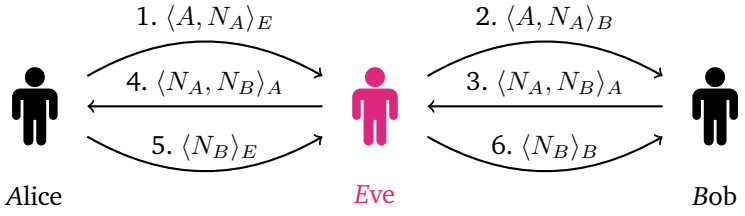
1. Alice envoie $\langle A, N_A \rangle_X$;
2. Bob reçoit $\langle Y, M \rangle_B$ et envoie $\langle M, N_B \rangle_Y$;
3. Alice reçoit $\langle K, L \rangle_A$ et envoie $\langle L \rangle_X$.

Le protocole doit donc résister aux attaques cryptographiques. Plus précisément, les propriétés suivantes doivent être satisfaites lorsqu'Alice et Bob ne détectent aucune incongruité:

— Seul X peut apprendre N_A (donc si $X = B$, alors Eve n'apprend *pas* N_A);

— Seul Y peut apprendre N_B (donc si $Y = A$, alors Eve n'apprend *pas* N_B).

Une attaque a été identifiée 18 ans après la parution du protocole par l'informaticien Gavin Lowe à l'aide d'un outil de vérification algorithmique. Voici un schéma de l'attaque, où Eve se fait passer à la fois pour Alice et Bob:



L'attaque peut être identifiée automatiquement, par exemple, en modélisant le protocole dans le langage Promela, en spécifiant les propriétés en LTL, et en les analysant à l'aide de l'outil Spin. Par exemple, voici la modélisation et la spécification fournies par le chercheur Stephan Merz dans [Mer01], où certaines simplifications sont faites, par ex. Eve ne peut stocker qu'un message intercepté.

2.10 Exercices

2.1) Soit $AP = \{\mathit{vert}, \mathit{jaune}, \mathit{rouge}\}$ l'ensemble où chaque proposition atomique indique si un feu de circulation est allumé ou non. Dites en français ce que spécifient ces formules LTL:

a) $\mathsf{F}(\mathit{vert} \wedge \neg \mathit{jaune} \wedge \neg \mathit{rouge})$

b) $\mathsf{G}(\mathit{rouge} \rightarrow \neg \mathsf{X}\, \mathit{vert})$

c) $\mathsf{G}(\mathit{rouge} \rightarrow (\mathit{rouge}\ \mathsf{U}\ (\mathit{jaune} \wedge \mathsf{X}(\mathit{jaune}\ \mathsf{U}\ \mathit{vert}))))$

(*tiré en partie de [BK08, ex. 5.4]*)

2.2) Pour chaque formule de l'exercice précédent, donnez un mot infini de votre choix qui satisfait la formule et un qui ne la satisfait pas.

2.3) Soit $AP = \{\mathit{vert}, \mathit{jaune}, \mathit{rouge}\}$ l'ensemble où chaque proposition atomique indique si un feu de circulation est allumé ou non. Spécifiez ces propriétés en LTL:

a) Il y a toujours exactement un feu allumé.

b) Le feu rouge est allumé infiniment souvent.

c) On ne peut pas passer du vert au rouge sans passer par le jaune.

2.4) Soit $AP = \{d, r\}$ l'ensemble où d indique qu'une demande est faite à un serveur et r indique qu'une réponse est envoyée par le serveur. Spécifiez ces propriétés en LTL:

a) Toute demande est éventuellement suivie d'une réponse.

b) Il y a éventuellement toujours des demandes.

c) Au moins une demande a lieu en même temps qu'une réponse.

d) Il y a un nombre infini de réponses.

2.5) Montrez ces équivalences:

a) $\mathsf{F}(\varphi_1 \vee \varphi_2) \equiv (\mathsf{F}\varphi_1) \vee (\mathsf{F}\varphi_2)$

b) $\mathsf{G}(\varphi_1 \wedge \varphi_2) \equiv (\mathsf{G}\varphi_1) \wedge (\mathsf{G}\varphi_2)$

c) $\mathsf{FGF}\varphi \equiv \mathsf{GF}\varphi$

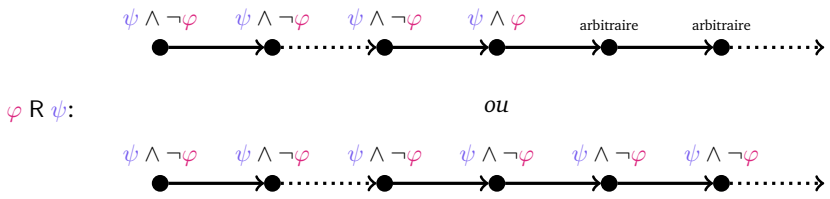
d) $\mathsf{GFG}\varphi \equiv \mathsf{FG}\varphi$

2.6) On pourrait être tenté de croire que $\varphi\ \mathsf{U}\ \psi$ signifie

« φ est satisfaite tant que ψ ne l'est pas ». (*)

Cependant, cela n'est pas le cas comme U force ψ à être éventuellement satisfaite. Définissez un nouvel opérateur binaire W, en fonction des opérateurs existants, qui se comporte comme (*).

2.7) Définissez un nouvel opérateur temporel R, en fonction des opérateurs existants, tel que φ R ψ spécifie « la validité de φ permet de relâcher celle de ψ » Schématiquement, cela correspond à:



2.8) Lesquelles de ces équivalences sont vraies? Pensez à une justification.

a) $(\varphi \vee \psi)\ \mathsf{U}\ \psi \stackrel{?}{\equiv} \varphi\ \mathsf{U}\ \psi$.

b) $\varphi\ \mathsf{U}\ (\varphi \vee \psi) \stackrel{?}{\equiv} \varphi\ \mathsf{U}\ \psi$.

c) $\neg(\varphi\ \mathsf{U}\ \psi) \stackrel{?}{\equiv} (\neg\varphi)\ \mathsf{U}\ (\neg\psi)$

d) $\mathsf{X}(\varphi\ \mathsf{U}\ \psi) \stackrel{?}{\equiv} (\mathsf{X}\varphi)\ \mathsf{U}\ (\mathsf{X}\psi)$

2.9) Convainquez-vous de la règle de distributivité $\mathsf{FG}(\varphi_1 \wedge \varphi_2) \equiv \mathsf{FG}\varphi_1 \wedge \mathsf{FG}\varphi_2$, préférablement avec une démonstration.

2.10) Posons $\varphi := \mathsf{GF}p \rightarrow \mathsf{FG}(q \vee r)$ et $\psi := (r\ \mathsf{U}\ \mathsf{X}p)\ \mathsf{U}\ (q \wedge \neg\mathsf{XX}s)$. Dites si ces mots infinis sur $AP := \{p, q, r, s\}$ satisfont respectivement φ et ψ:

a) $\emptyset^\omega$

b) $\{p, q, r, s\}^\omega$

c) $\{p, q\}^\omega$

d) $\{r\}\emptyset\{p, q, s\}^\omega$

e) $\{r\}\emptyset(\{p, q\}\{r, s\})^\omega$

f) $\{r\}\emptyset\{p\}\{q, r\}(\{p, s\}\emptyset)^\omega$

(*tiré d'exercices enseignés à l'Université technique de Munich*)

3

Langages ω-réguliers

Dans les deux premiers chapitres, nous avons vu comment modéliser différents systèmes à l'aide de structures de Kripke, et comment spécifier leurs propriétés en logique temporelle linéaire. Une question demeure: comment vérifier systématiquement qu'une structure de Kripke satisfait une propriété LTL? Autrement dit, comment faire le pont afin de tester algorithmiquement si $\text{Traces}(\mathcal{T}) \subseteq [\![\varphi]\!]$?

Dans cette optique, observons que $\text{Traces}(\mathcal{T})$ et $[\![\varphi]\!]$ sont tous deux des sous-ensembles de Σ^ω où $\Sigma := 2^{AP}$, donc des langages de mots infinis sur l'alphabet Σ. Nous chercherons donc à représenter ces langages symboliquement par des automates. Ceux-ci serviront de structures de données pouvant être manipulées algorithmiquement.

Il est bien connu que les langages reconnus par les automates finis (déterministes ou non) correspondent précisément aux langages décrits par les expressions régulières. Ces formalismes opèrent sur des mots *finis*. Dans notre contexte, nous devons plutôt raisonner sur des mots infinis. Nous introduisons donc les expressions ω-régulières, puis les automates de Büchi, qui étendent naturellement ces formalismes au cas infini.

3.1 Expressions ω-régulières

3.1.1 Expressions régulières

Soit Σ un alphabet fini, donc un ensemble fini de symboles appelés *lettres*. Rappelons que les expressions régulières, sur un alphabet Σ, sont décrites par cette grammaire, où $a \in \Sigma$ et ε dénote le mot vide:

$$r ::= r^* \mid (r \cdot r) \mid (r + r) \mid a \mid \varepsilon$$

Le *langage* $\mathcal{L}(r) \subseteq \Sigma^*$ d'une expression régulière r est défini récursivement par:

$$\begin{aligned}
\mathcal{L}(r^*) &\coloneqq \bigcup_{n\in\mathbb{N}} \{w_1 w_2 \cdots w_n : w_1, w_2, \ldots, w_n \in \mathcal{L}(r)\},\\
\mathcal{L}(r \cdot r') &\coloneqq \{uv : u \in \mathcal{L}(r), v \in \mathcal{L}(r')\},\\
\mathcal{L}(r + r') &\coloneqq \mathcal{L}(r) \cup \mathcal{L}(r'),\\
\mathcal{L}(a) &\coloneqq \{a\},\\
\mathcal{L}(\varepsilon) &\coloneqq \{\varepsilon\}.
\end{aligned}$$

3.1.2 Syntaxe

La syntaxe des *expressions ω-régulières* sur alphabet Σ est définie par cette grammaire, où $a \in \Sigma$:

$$\begin{aligned}
s &::= r^\omega \mid (r \cdot s) \mid (s + s)\\
r &::= r^* \mid (r \cdot r) \mid (r + r) \mid a \mid \varepsilon
\end{aligned}$$

Cette syntaxe est restreinte par l'interdiction d'utiliser l'opérateur r^ω lorsque le langage de r est vide ou contient le mot vide ε. Ainsi, par exemple, $(a + \varepsilon)^\omega$ et $\emptyset^\omega$ ne sont *pas* des expressions ω-régulières valides. En mots, l'ensemble des expressions ω-régulières sur alphabet Σ est donc défini récursivement par:

— Si r est une expression régulière telle que $\varepsilon \notin \mathcal{L}(r)$ et $\mathcal{L}(r) \neq \emptyset$, alors r^ω est une expression ω-régulière;

— Si r est une expression régulière et s est une expression ω-régulière, alors $(r \cdot s)$ est une expression ω-régulière;

— Si s et s' sont des expressions ω-régulières, alors $(s+s')$ est une expression ω-régulière.

Exemple.

Ces quatre expressions sur $\Sigma \coloneqq \{a, b\}$ sont toutes ω-régulières:

$$(a + b)^\omega \qquad a(a + b)^\omega$$
$$(ab)^\omega \qquad b^*(aa^*bb^*)^\omega$$

3.1.3 Sémantique

Nous associons un *langage* $\mathcal{L}(s) \subseteq \Sigma^\omega$ à chaque expression ω-régulière s, défini récursivement par:

$$\begin{aligned}\mathcal{L}(r^\omega) &\coloneqq \{w_0 w_1 \cdots : w_i \in \mathcal{L}(r) \text{ pour tout } i \in \mathbb{N}\},\\ \mathcal{L}(r \cdot s) &\coloneqq \{w\sigma : w \in \mathcal{L}(r), \sigma \in \mathcal{L}(s)\},\\ \mathcal{L}(s + s') &\coloneqq \mathcal{L}(s) \cup \mathcal{L}(s').\end{aligned}$$

Ainsi, l'opérateur r^ω est une variante de l'opérateur r^* où plutôt que de concaténer des mots de $\mathcal{L}(r)$ un nombre fini de fois, on en concatène une infinité.

Nous disons qu'un langage est *ω-régulier* s'il peut être décrit par une expression ω-régulière.

Exemples.

Soit $\Sigma \coloneqq \{a, b\}$. Les expressions ω-régulières suivantes sur alphabet Σ décrivent respectivement ces langages ω-réguliers:

$(a+b)^\omega$: ensemble de tous les mots (infinis),

$a(a+b)^\omega$: ensemble des mots qui débutent par la lettre a,

$(ab)^\omega$: ensemble contenant l'unique mot $ababab\cdots$,

$b^*(aa^*bb^*)^\omega$: ensemble des mots avec une infinité de a *et* de b,

$(a+b)^*b^\omega$: ensemble des mots avec un nombre fini de a,

$(a(a+b))^\omega$: ensemble des mots avec un a à chaque position paire.

3.2 Automates de Büchi

Les expressions ω-régulières sont pratiques afin de décrire des langages, mais moins adaptées à la manipulation algorithmique. Nous introduisons donc le modèle équivalent des automates de Büchi. Un automate fini classique accepte un mot fini w s'il atteint un état acceptant en complétant la lecture de w. Cette notion ne fait pas de sens dans le contexte d'un mot infini puisqu'il n'y a pas de fin au mot. Un automate de Büchi accepte donc plutôt un mot infini σ si sa lecture visite infiniment souvent un état acceptant. Formellement, un *automate de Büchi* est un quintuplet $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ où

— Q est un ensemble fini dont les éléments sont appelés *états*,

— Σ est un *alphabet* fini,

— $\delta\colon Q \times \Sigma \to 2^Q$ est la *fonction de transition*,

— $Q_0 \subseteq Q$ est un ensemble d'états dits *initiaux*,

— $F \subseteq Q$ est un ensemble d'états dits *acceptants*.

Exemple.

Considérons les automates de Büchi $\mathcal{A}$ et $\mathcal{B}$ illustrés à la figure 3.1.

L'automate $\mathcal{A}$ débute dans l'état initial p, lit la lettre a, puis boucle pour toujours sur l'état acceptant q sur une lettre arbitraire. Ainsi, cet automate accepte le langage $a(a+b)^\omega$ des mots qui débutent par a.

L'automate $\mathcal{B}$ débute dans l'état initial p. Il lit la lettre a en se déplaçant ou bien dans l'état acceptant q ou r. Dans le premier cas, seule la lettre a peut être lue, alors que dans le second, seule la lettre b peut être lue. Ainsi, cet automate accepte le langage des mots qui débutent par a suivie d'une même lettre répétée indéfiniment, c.-à-d. $a(a^\omega + b^\omega)$.

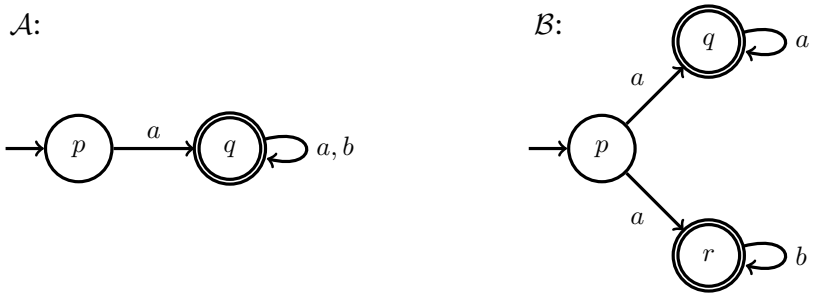


FIGURE 3.1 – Exemples d'automates de Büchi. L'automate $\mathcal{A}$ accepte le langage $a(a+b)^\omega$ et l'automate $\mathcal{B}$ accepte le langage $a(a^\omega + b^\omega)$.

3.2.1 Langage d'un automate

Nous écrivons $p \xrightarrow{a} q$ pour dénoter que $q \in \delta(p, a)$, et ainsi indiquer que l'automate possède une *transition* de p vers q étiquetée par la lettre a. Nous disons qu'un mot infini $\sigma \in \Sigma^\omega$ est *accepté* par un automate de Büchi $\mathcal{A}$ s'il existe une suite (infinie) d'états $q_0, q_1, \ldots \in Q$ telle que

— $q_0 \xrightarrow{\sigma(0)} q_1 \xrightarrow{\sigma(1)} \cdots$,
— $q_0 \in Q_0$, et
— $q_i \in F$ pour une infinité de $i \in \mathbb{N}$.

Autrement dit, un mot infini σ est accepté par $\mathcal{A}$ s'il est possible de lire ses lettres à partir d'un état initial via une suite de transitions qui visite infiniment souvent des états acceptants. En général, il peut exister plusieurs manières de lire un même mot, il suffit donc que l'une d'elles satisfasse ces critères.

Le *langage* d'un automate de Büchi $\mathcal{A}$, dénoté $\mathcal{L}(\mathcal{A})$, est l'ensemble des mots infinis qu'il accepte:

$$\mathcal{L}(\mathcal{A}) \coloneqq \{\sigma \in \Sigma^\omega : \sigma \text{ est accepté par } \mathcal{A}\}.$$

Exemple.

Reconsidérons les automates de Büchi $\mathcal{A}$ et $\mathcal{B}$ illustrés à la figure 3.1.

Pour l'automate $\mathcal{A}$, nous avons:

$$p \xrightarrow{a} q, \quad q \xrightarrow{a} q, \quad q \xrightarrow{b} q.$$

Cet automate accepte $abab\cdots$ car $p \xrightarrow{a} q \xrightarrow{b} q \xrightarrow{a} q \xrightarrow{b} \cdots$.

Pour l'automate $\mathcal{B}$, nous avons:

$$p \xrightarrow{a} q, \quad p \xrightarrow{a} r, \quad q \xrightarrow{a} q, \quad r \xrightarrow{b} r.$$

L'automate n'accepte pas le mot $abaaa\cdots$ car la seule manière de lire ab consiste à se déplacer dans l'état r où il est impossible de lire d'autres a.

Tel que mentionné plus tôt, nous avons

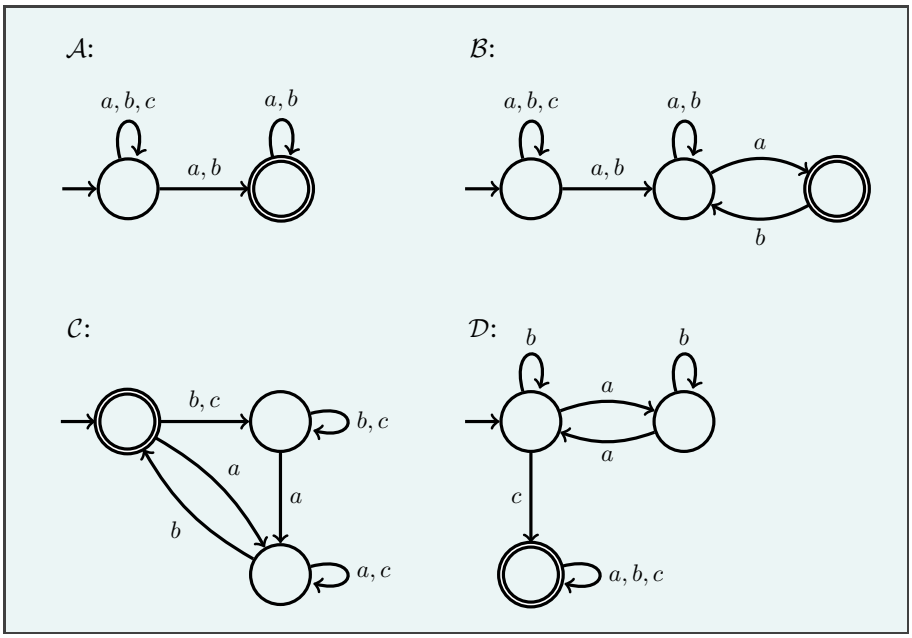
$$\mathcal{L}(\mathcal{A}) = \mathcal{L}(a(a+b)^\omega), \text{ et}$$
$$\mathcal{L}(\mathcal{B}) = \mathcal{L}(a(a^\omega + b^\omega)).$$

Exemples.

Identifions des automates de Büchi qui acceptent ces langages ω-réguliers (pensez-y d'abord):

langage	**expression ω-régulière**
mots avec un nombre fini de c	$(a+b+c)^*(a+b)^\omega$
mots avec un nombre infini de a, un nombre infini de b et un nombre fini de c	$(a+b+c)^*(aa^*bb^*)^\omega$
mots avec un nombre infini de a et un nombre infini de b	$((b+c)^*a(a+c)^*b)^\omega$
mots avec au moins un c et un nombre pair de a avant le premier c	$(b^*ab^*a)^*b^*c(a+b+c)^\omega$

Ces langages sont acceptés respectivement par ces automates:



3.2.2 Déterminisme et expressivité

Nous disons qu'un automate de Büchi est *déterministe* si $|Q_0| = 1$, et $|\delta(q, a)| \leq 1$ pour tout état $q \in Q$ et toute lettre $a \in \Sigma$. Autrement dit, un automate de Büchi est déterministe s'il ne possède qu'un seul état initial, et aucun état ayant deux transitions sortantes étiquetées par la même lettre. Par exemple, l'automate $\mathcal{A}$ de la figure 3.1 est déterministe, mais l'automate $\mathcal{B}$ est non déterministe puisque $p \xrightarrow{a} q$ et $p \xrightarrow{a} r$.

Contrairement aux automates classiques sur les mots finis, le déterminisme est strictement plus faible que le non déterminisme dans le contexte des mots infinis. Par exemple, il n'existe *aucun* automate de Büchi déterministe qui accepte le langage $(a + b)^* a^\omega$, c'est-à-dire le langage des mots qui possèdent un nombre fini de b.

Mentionnons également que les automates de Büchi (non déterministes) et les expressions ω-régulières capturent précisément la même classe de langages (voir par ex. [BK08, théorème 4.32]):

Proposition 2 ([McN66])**.** *L'ensemble des langages acceptés par les automates de Büchi correspond précisément aux langages ω-réguliers.*

3.3 Intersection d'automates de Büchi

Soient $\mathcal{A}$ et $\mathcal{B}$ deux automates de Büchi sur un alphabet commun. Si l'on voit chacun de ces automates comme une structure de données qui stocke symboliquement un langage, il est intéressant d'implémenter une opération qui per-

met de les combiner afin d'obtenir une représentation de leurs mots communs. Pour y arriver, il suffit de construire un automate qui accepte l'intersection de leurs langages. Nous montrons donc comment construire un automate de Büchi, nommé $\mathcal{A} \cap \mathcal{B}$, tel que $\mathcal{L}(\mathcal{A} \cap \mathcal{B}) = \mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{B})$. Autrement dit, cet automate acceptera précisément les mots acceptés par $\mathcal{A}$ *et* $\mathcal{B}$.

3.3.1 Construction à partir d'un exemple

Avant de présenter la construction générale de $\mathcal{A} \cap \mathcal{B}$, considérons le cas particulier des automates $\mathcal{A}$ et $\mathcal{B}$ illustrés à la figure 3.2. Le langage de l'automate $\mathcal{A}$ est l'ensemble des mots qui possèdent une infinité de a et qui ne contiennent pas d'occurrence de bab. Le langage de l'automate $\mathcal{B}$ est l'ensemble des mots possédant une infinité de b. Ainsi, l'intersection de ces deux langages est l'ensemble des mots avec une infinité de a, une infinité de b, et qui ne contiennent pas bab.

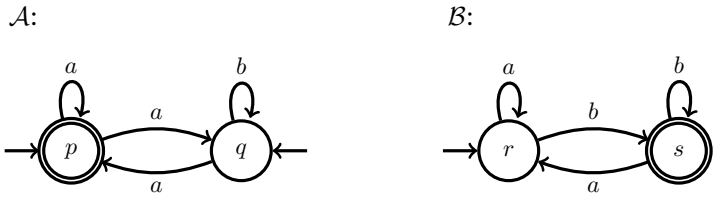


FIGURE 3.2 – Deux automates de Büchi $\mathcal{A}$ et $\mathcal{B}$ pour lesquels nous désirons construire l'intersection.

Nous construisons d'abord un automate $\mathcal{C}$ qui simule les automates $\mathcal{A}$ et $\mathcal{B}$ simultanément « en parallèle ». Cela peut être réalisé en opérant sur le produit cartésien des états de $\mathcal{A}$ et $\mathcal{B}$. Les états initiaux de $\mathcal{C}$ correspondent aux paires où le premier état est initial dans $\mathcal{A}$ et où le second état est initial dans $\mathcal{B}$. Les transitions indiquent l'évolution respective des états de $\mathcal{A}$ et $\mathcal{B}$. L'automate $\mathcal{C}$ obtenu de cette manière est illustré à la figure 3.3.

L'automate $\mathcal{C}$ peut lire exactement les mots pouvant être lus dans $\mathcal{A}$ et $\mathcal{B}$. On pourrait donc être tenté de conclure que leur intersection est vide car il est impossible d'atteindre les états acceptants de $\mathcal{A}$ et $\mathcal{B}$ simultanément, c'est-à-dire la paire (p, s). Ce n'est pourtant pas le cas puisque, par exemple, $\mathcal{A}$ et $\mathcal{B}$ acceptent tous deux le mot infini $(aab)^\omega$. L'automate $\mathcal{C}$ ne donne donc pas assez d'information pour décider quels mots doivent être acceptés ou refusés.

Afin de pallier ce problème, nous ajoutons une troisième composante aux états de $\mathcal{C}$ qui indique lequel de $\mathcal{A}$ et $\mathcal{B}$ est le prochain à devoir franchir un état acceptant. L'automate $\mathcal{A} \cap \mathcal{B}$ obtenu de cette façon est illustré à la figure 3.4.

L'idée derrière l'automate $\mathcal{A} \cap \mathcal{B}$ est la suivante. Lorsque l'automate est dans l'état $(x, y, \mathcal{A})$ et que x est un état acceptant de $\mathcal{A}$, alors l'automate passe à la copie de droite. Similairement, lorsque l'automate est dans l'état $(x, y, \mathcal{B})$ et que y est un état acceptant de $\mathcal{B}$, alors l'automate passe à la copie de gauche.

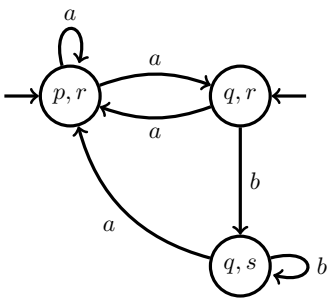


FIGURE 3.3 – Produit des automates $\mathcal{A}$ et $\mathcal{B}$ de la figure 3.2.

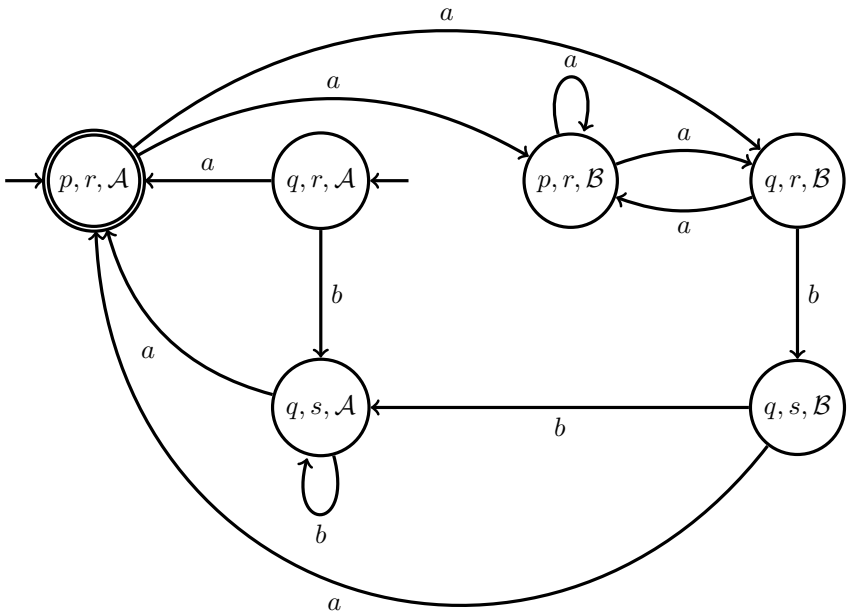


FIGURE 3.4 – Intersection des automates $\mathcal{A}$ et $\mathcal{B}$ de la figure 3.2.

Cela permet d'alterner indéfiniment entre les états acceptants de $\mathcal{A}$ et $\mathcal{B}$, *tout en* simulant les deux automates en parallèle. Les états acceptants de $\mathcal{A} \cap \mathcal{B}$ sont ceux de $\mathcal{A}$ dans la copie de gauche. Nous pourrions également choisir ceux de $\mathcal{B}$ dans la copie de droite; mais il est crucial de ne choisir que l'*une* de ces deux options afin d'assurer que $\mathcal{A} \cap \mathcal{B}$ alterne indéfiniment entre les deux copies.

3.3.2 Construction générale

Présentons maintenant la construction dans sa généralité. Soient les automates de Büchi $\mathcal{A}$ et $\mathcal{B}$ tels que

$$\mathcal{A} = (Q_\mathcal{A}, \Sigma, \delta_\mathcal{A}, Q_{0,\mathcal{A}}, F_\mathcal{A}), \text{ et}$$
$$\mathcal{B} = (Q_\mathcal{B}, \Sigma, \delta_\mathcal{B}, Q_{0,\mathcal{B}}, F_\mathcal{B}).$$

L'automate $\mathcal{A} \cap \mathcal{B} \coloneqq (Q, \Sigma, \delta, Q_0, F)$ est défini de la façon suivante.

Ses états, états initiaux et états acceptants sont définis respectivement par:

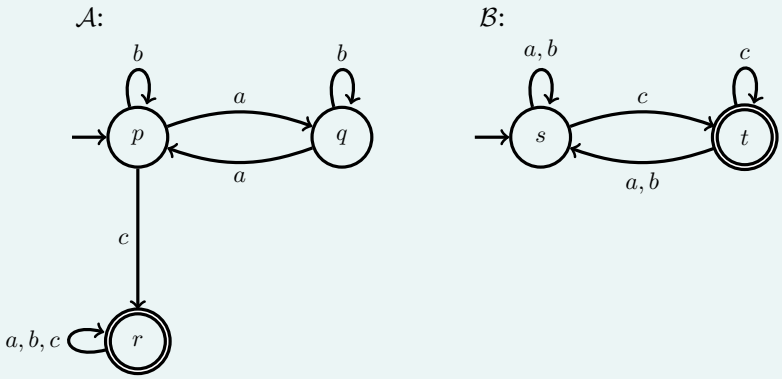
$$\begin{aligned} Q &\coloneqq Q_\mathcal{A} \times Q_\mathcal{B} \times \{\mathcal{A}, \mathcal{B}\}, \\ Q_0 &\coloneqq Q_{0,\mathcal{A}} \times Q_{0,\mathcal{B}} \times \{\mathcal{A}\}, \\ F &\coloneqq F_\mathcal{A} \times Q_\mathcal{B} \times \{\mathcal{A}\}. \end{aligned}$$

La fonction de transition δ est définie par la règle suivante. Si $q_\mathcal{A} \xrightarrow{a} r_\mathcal{A}$ dans $\mathcal{A}$, et $q_\mathcal{B} \xrightarrow{a} r_\mathcal{B}$ dans $\mathcal{B}$, alors nous ajoutons à $\mathcal{A} \cap \mathcal{B}$ la transition:

$$(q_\mathcal{A}, q_\mathcal{B}, I) \xrightarrow{a} (r_\mathcal{A}, r_\mathcal{B}, I') \quad \text{où} \quad I' \coloneqq \begin{cases} \mathcal{B} & \text{si } I = \mathcal{A} \text{ et } q_\mathcal{A} \in F_\mathcal{A}, \\ \mathcal{A} & \text{si } I = \mathcal{B} \text{ et } q_\mathcal{B} \in F_\mathcal{B}, \\ I & \text{sinon.} \end{cases}$$

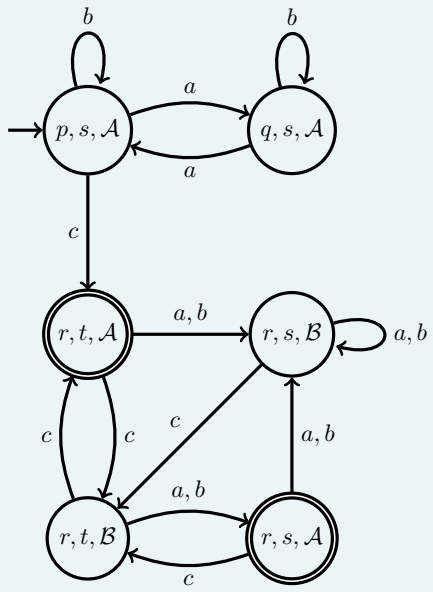
Exemple.

Voyons un autre exemple de la construction de l'intersection. Considérons ces automates de Büchi $\mathcal{A}$ et $\mathcal{B}$:



L'automate $\mathcal{A}$ accepte les mots qui contiennent au moins un c et un nombre pair de a avant la première occurrence de c. L'automate $\mathcal{B}$ accepte les mots qui contiennent une infinité d'occurrences de c.

L'automate $\mathcal{A} \cap \mathcal{B}$ obtenu selon notre construction est comme suit:



Cet automate accepte les mots qui contiennent une infinité de c et un nombre pair de a avant la première occurrence de c.

3.4 Exercices

3.1) Donnez une expression ω-régulière et un automate de Büchi pour chacun de ces langages:

a) $L \subseteq \{a, b\}^\omega$ tel que $\sigma \in L$ ssi le nombre de a entre deux b est pair;

b) Comme en (a) mais sur alphabet $\{a, b, c\}$;

c) $L \subseteq \{a, b, c\}^\omega$ tel que $\sigma \in L$ ssi σ ne contient aucun b qui suit immédiatement un a.

3.2) Ces expressions décrivent-elles le même langage?

a) $(b^*a^*ab)^\omega$ et $(a^*ab + b^*ba)^\omega$;

b) $a(b^*a + b)^\omega$ et $(ab^*a + ab)^\omega$.

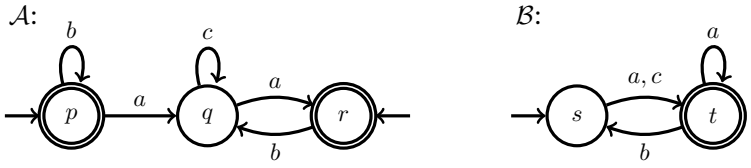
3.3) Expliquez comment transformer une expression ω-régulière arbitraire en un automate de Büchi. Pensez à la construction des opérateurs ω, $\cdot$ et $+$.

3.4) ★ Expliquez comment transformer un automate de Büchi non déterministe arbitraire en une expression ω-régulière. Pensez d'abord à un automate avec un seul état initial et un seul état final.

3.5) Comment peut-on construire l'union de deux automates de Büchi?

3.6) Étant donné un automate fini déterministe classique $\mathcal{A}$, il suffit d'inverser les états acceptants et non acceptants afin d'obtenir un automate qui accepte $\overline{\mathcal{L}(\mathcal{A})}$. Est-ce aussi le cas pour les automates de Büchi déterministes?

3.7) Construisez l'intersection $\mathcal{A} \cap \mathcal{B}$ de ces deux automates de Büchi:



3.8) Considérons un automate de Büchi qui possède plusieurs états initiaux. Est-il possible d'obtenir un automate équivalent avec un seul état initial?

3.9) Qu'en est-il des états acceptants? Autrement dit, est-ce toujours possible d'obtenir un automate de Büchi équivalent avec un seul état acceptant?

3.10) ★ Montrez qu'il n'existe pas d'automate de Büchi *déterministe* qui accepte le langage des mots avec un nombre fini de b sur alphabet $\Sigma := \{a, b\}$; autrement dit, le langage décrit par $(a + b)^*a^\omega$.

4

Vérification algorithmique de formules LTL

Dans ce chapitre, nous survolons l'automatisation de la vérification de spécifications LTL d'une structure de Kripke. Celle-ci s'appuiera sur la construction et la manipulation d'automates de Büchi.

4.1 LTL vers automates de Büchi

Soit *AP* un ensemble de propositions atomiques. Posons $\Sigma \coloneqq 2^{AP}$. Pour toute formule LTL φ sur *AP*, l'ensemble $[\![\varphi]\!]$ est un sous-ensemble de Σ^ω et ainsi un langage de mots infinis sur alphabet Σ. Nous pouvons ainsi chercher à convertir une formule LTL vers un automate de Büchi.

Exemple.

Pour $AP = \{p, q\}$, nous avons:

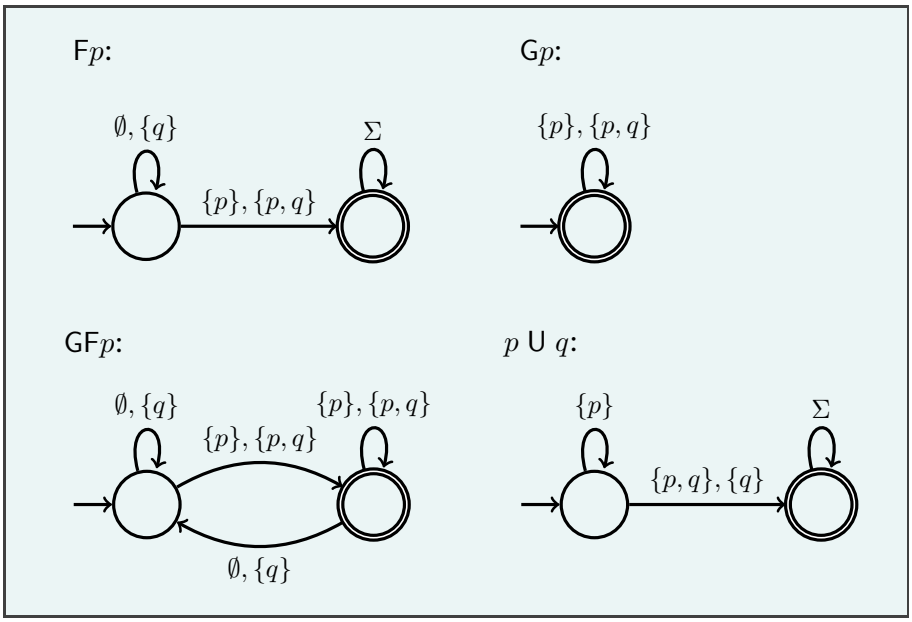
$$[\![\mathsf{F}p]\!] = \underbrace{(\emptyset + \{q\})^*}_{\text{pas de } p} \underbrace{(\{p\} + \{p,q\})}_{\text{occurrence de } p} \Sigma^\omega,$$

$$[\![\mathsf{G}p]\!] = \underbrace{(\{p\} + \{p,q\})^\omega}_{\text{occurrence de } p},$$

$$[\![\mathsf{GF}p]\!] = (\underbrace{(\emptyset + \{q\})^*}_{\text{pas de } p} \underbrace{(\{p\} + \{p,q\})}_{\text{occurrence de } p})^\omega,$$

$$[\![p \mathsf{U} q]\!] = \underbrace{(\{p\} + \{p,q\})^*}_{\text{occurrence de } p} \underbrace{(\{q\} + \{p,q\})}_{\text{occurrence de } q} \Sigma^\omega.$$

Ainsi, ces formules peuvent être représentées par ces automates:



Notons que le langage associé à une formule LTL s'exprime récursivement en fonction de la répétition, la concaténation, l'union, l'intersection et la complémentation:

$$\begin{aligned}
[\![\varphi \wedge \psi]\!] &= [\![\varphi]\!] \cap [\![\psi]\!] & [\![\mathsf{X}\varphi]\!] &= \Sigma[\![\varphi]\!] \\
[\![\varphi \vee \psi]\!] &= [\![\varphi]\!] \cup [\![\psi]\!] & [\![\mathsf{F}\varphi]\!] &= \Sigma^*[\![\varphi]\!] \\
[\![\neg\varphi]\!] &= \overline{[\![\varphi]\!]} & [\![\mathsf{G}\varphi]\!] &= \overline{\Sigma^*\overline{[\![\varphi]\!]}} \\
[\![\mathit{vrai}]\!] &= \Sigma^\omega & [\![p]\!] &= \bigcup_{\{p\}\subseteq A\subseteq AP} A\Sigma^\omega
\end{aligned}$$

$$[\![\varphi \ \mathsf{U}\ \psi]\!] = \bigcup_{j\in\mathbb{N}} \left[\left(\bigcap_{0\leq i<j} \Sigma^i[\![\varphi]\!] \right) \cap \ \Sigma^j[\![\psi]\!] \right].$$

Nous pourrions ainsi chercher à construire un automate de Büchi récursivement. Cependant, au moins deux problèmes surgissent. Premièrement, la complémentation d'un automate de Büchi est loin d'être simple et s'appuie sur des constructions relativement complexes. Il est connu que:

Proposition 3. *Pour tout automate de Büchi $\mathcal{A}$ de n états, il existe un automate de Büchi $\mathcal{B}$ tel que $\mathcal{L}(\mathcal{B}) = \overline{\mathcal{L}(\mathcal{A})}$ et $\mathcal{B}$ possède $2^{\mathcal{O}(n \log n)}$ états. De plus, cette borne peut être atteinte pour certaines familles d'automates.*

Deuxièmement, l'identité ci-dessus pour l'opérateur temporel U utilise une union *infinie*, ce qui correspond, à priori, à composer une infinité d'automates.

Ces enjeux peuvent être surmontés en construisant directement un automate de Büchi [1] dont les états représentent essentiellement des ensembles de sous-formules à vérifier simultanément à la volée. Il existe plusieurs telles constructions, assez complexes, qui offrent certains compromis en théorie et en pratique. Par exemple, la description de [BK08, Thm. 5.41] mène à ce résultat classique:

Théorème 1. *Pour toute formule LTL φ sur AP, il est possible de construire un automate de Büchi $\mathcal{A}_\varphi$ de $2^{\mathcal{O}(|\varphi|)}$ états tel que $\mathcal{L}(\mathcal{A}_\varphi) = [\![\varphi]\!]$. De plus, certaines familles de formules requièrent un nombre exponentiel d'états.*

En pratique, cette taille exponentielle ne se manifeste pas systématiquement. Il existe aussi plusieurs approches pour chercher à produire des automates avec peu d'états ou à réduire la taille d'automates déjà construits. Par exemple, des outils modernes comme Spot [DLF+16] et Owl [KMS18] sont entièrement dédiés à ces tâches. Une présentation de ces techniques dépasse le cadre du cours.

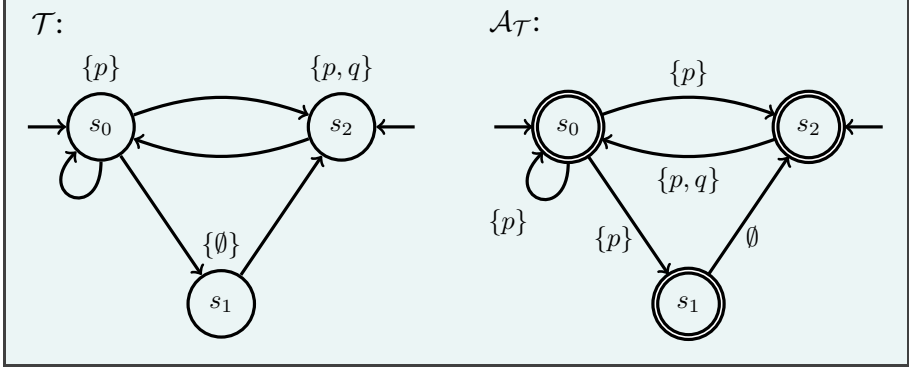
4.2 Structures de Kripke vers automates de Büchi

Afin de comparer une formule LTL à une structure de Kripke, on peut transformer la structure en un automate de Büchi. Cette conversion s'avère bien plus simple puisqu'une structure de Kripke s'apparente déjà grandement à un automate. Voyons comment procéder.

Soit $\mathcal{T} = (S, \to, I, AP, L)$ une structure de Kripke. Nous associons à $\mathcal{T}$ un automate de Büchi $\mathcal{A}_\mathcal{T}$ tel que $\mathcal{L}(\mathcal{A}_\mathcal{T}) = \text{Traces}(\mathcal{T})$. L'automate $\mathcal{A}_\mathcal{T}$ est obtenu à partir de $\mathcal{T}$ en rendant *tous* ses états acceptants et en étiquetant chaque transition $s \to t$ par l'étiquette $L(s)$. Plus précisément, l'automate $\mathcal{A}_\mathcal{T}$ est défini par $\mathcal{A}_\mathcal{T} := (S, 2^{AP}, \delta, I, S)$ où δ est défini par:

$$\delta(q, a) := \{r \in Q : q \to r \text{ et } L(q) = a\} \qquad \text{pour tous } q \in Q, a \in 2^{AP}.$$

Exemple.



1. En fait, il s'agirait normalement d'un automate de Büchi *généralisé*.

Nous avons comme conséquence quasi immédiate de la définition que:

Proposition 4. $\mathcal{L}(\mathcal{A}_\mathcal{T}) = \text{Traces}(\mathcal{T})$ *pour toute structure de Kripke* $\mathcal{T}$.

Démonstration. Soit E l'ensemble des exécutions infinies de $\mathcal{T}$. Nous avons:

$$\begin{aligned}
\sigma \in \text{Traces}(\mathcal{T}) &\iff \exists s_0 s_1 \cdots \in E : \sigma = \text{trace}(s_0 s_1 \cdots) \\
&\iff \exists s_0 s_1 \cdots \in E : \sigma = L(s_0)L(s_1)\cdots \\
&\iff \exists s_0, s_1, \ldots \in S : s_0 \to s_1 \to \cdots \text{ et } \sigma = L(s_0)L(s_1)\cdots \\
&\iff \exists s_0, s_1, \ldots \in S : s_0 \xrightarrow{\sigma(0)} s_1 \xrightarrow{\sigma(1)} \cdots \text{ dans } \mathcal{A}_\mathcal{T} \\
&\iff \sigma \in \mathcal{L}(\mathcal{A}_\mathcal{T}) \qquad (*)
\end{aligned}$$

où (*) découle du fait que tous les états de $\mathcal{A}_\mathcal{T}$ sont acceptants. □

Observation.

Bien que *tous* les états de l'automate $\mathcal{A}_\mathcal{T}$ soient acceptants, cela ne signifie pas qu'il accepte forcément tous les mots. En effet, les transitions sortantes d'un état sont toutes étiquetées par la *même* lettre. En particulier, nous avons $\emptyset^\omega \notin \mathcal{L}(\mathcal{A}_\mathcal{T})$ dans l'exemple précédent.

4.3 Vérification d'une spécification

Afin de vérifier qu'une structure de Kripke $\mathcal{T}$ satisfait une certaine formule LTL φ, il suffit de:

1. Construire l'automate de Büchi $\mathcal{A}_{\neg\varphi}$,
2. Construire l'automate de Büchi $\mathcal{A}_\mathcal{T}$, et
3. Tester si $\mathcal{L}(\mathcal{A}_\mathcal{T}) \cap \mathcal{L}(\mathcal{A}_{\neg\varphi}) = \emptyset$.

En effet, la validité de cette procédure découle de ces équivalences:

$$\begin{aligned}
\mathcal{T} \models \varphi &\iff \text{Traces}(\mathcal{T}) \subseteq [\![\varphi]\!] \\
&\iff \text{Traces}(\mathcal{T}) \cap \overline{[\![\varphi]\!]} = \emptyset \\
&\iff \text{Traces}(\mathcal{T}) \cap [\![\neg\varphi]\!] = \emptyset \\
&\iff \mathcal{L}(\mathcal{A}_\mathcal{T}) \cap \mathcal{L}(\mathcal{A}_{\neg\varphi}) = \emptyset.
\end{aligned}$$

Rappelons que nous avons vu, à la section 3.3, un algorithme qui peut produire un automate $\mathcal{B}$ tel $\mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{A}_\mathcal{T}) \cap \mathcal{L}(\mathcal{A}_{\neg\varphi})$. Ainsi, la vérification de $\mathcal{T} \models \varphi$ se réduit au problème qui consiste à déterminer si $\mathcal{L}(\mathcal{B}) = \emptyset$. Voyons comment résoudre ce problème algorithmiquement.

4.3.1 Lassos

Nous présentons d'abord une condition suffisante et nécessaire afin de déterminer si le langage d'un automate de Büchi est vide. Soit $\mathcal{B} = (Q, \Sigma, \delta, Q_0, F)$ un automate de Büchi. Un *lasso* de $\mathcal{B}$ est une séquence de la forme

$$q_0 \xrightarrow{u} r \xrightarrow{v} r,$$

où $q_0 \in Q_0$, $r \in F$, $u \in \Sigma^*$ et $v \in \Sigma^+$. La figure 4.1 illustre un tel lasso.

L'existence d'un lasso caractérise le vide d'un automate de Büchi:

Proposition 5. *Soit $\mathcal{B}$ un automate de Büchi. Nous avons $\mathcal{L}(\mathcal{B}) \neq \emptyset$ si et seulement si $\mathcal{B}$ possède un lasso.*

Démonstration. $\Leftarrow$) Supposons que $\mathcal{B}$ possède un lasso. Par définition, il existe $q_0 \in Q_0$, $r \in F$, $u \in \Sigma^*$ et $v \in \Sigma^+$ tels que $q_0 \xrightarrow{u} r \xrightarrow{v} r$. Notons que

$$q_0 \xrightarrow{u} r \xrightarrow{v} r \xrightarrow{v} r \xrightarrow{v} \cdots .$$

Ainsi, $uv^\omega \in \mathcal{L}(\mathcal{B})$, ce qui implique que $\mathcal{L}(\mathcal{B})$ est non vide.

$\Rightarrow$) Supposons que $\mathcal{L}(\mathcal{B}) \neq \emptyset$. Il existe un mot infini $v_0 v_1 \cdots \in \Sigma^\omega$ et des états $q_0, q_1, \ldots \in Q$ tels que $q_0 \in Q_0$, $q_i \in F$ pour une infinité d'indices i, et

$$q_0 \xrightarrow{v_0} q_1 \xrightarrow{v_1} q_2 \xrightarrow{v_2} \cdots .$$

Puisque F est fini, il existe forcément deux indices $i < j$ tels que $q_i = q_j \in F$. Nous obtenons donc le lasso suivant:

$$q_0 \xrightarrow{v_0 v_1 \cdots v_{i-1}} q_i \xrightarrow{v_i v_{i+1} \cdots v_{j-1}} q_i. \qquad \square$$

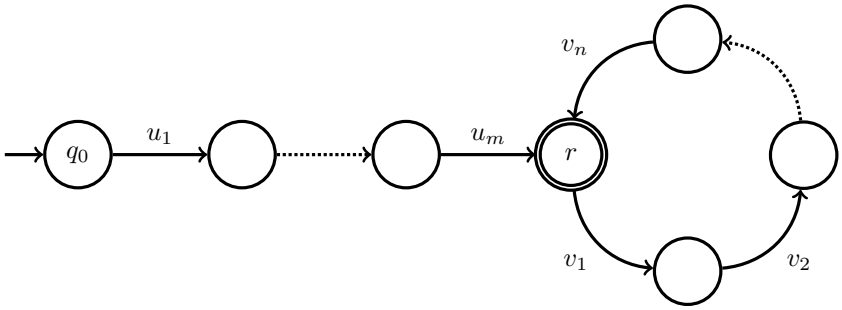


FIGURE 4.1 – Schéma d'un lasso où $|u| = m \geq 0$ et $|v| = n > 0$.

4.3.2 Détection algorithmique de lassos

Afin de détecter un lasso dans un automate de Büchi, il suffit de:

1. calculer l'ensemble R des états accessibles à partir des états initiaux;
2. pour tout état *acceptant* $f \in R$, chercher un chemin non vide de f vers f.

Une telle procédure est décrite à l'algorithme 1, où l'instruction « **terminer avec** x » court-circuite l'exécution et retourne la valeur x.

Algorithme 1 : Détection naïve de lasso par parcours en profondeur.

Entrées : Automate de Büchi $\mathcal{B} = (Q, \Sigma, \delta, Q_0, F)$
Sorties : $\mathcal{L}(\mathcal{B}) = \emptyset$?

```
// Calculer l'ensemble R des états accessibles
R ← ∅
dfs(q):
    ajouter q à R
    pour r : q → r
        si r ∉ R alors dfs(r)
pour q ∈ Q_0
    si q ∉ R alors
        dfs(q)
// Chercher un état acceptant de R qui peut s'atteindre
pour f ∈ F ∩ Q
    S_f ← ∅
    cycle(q):
        ajouter q à S_f
        pour r : q → r
            si r = f alors terminer avec « non vide »
            sinon si r ∉ S_f alors cycle(r)
    cycle(f)
terminer avec « vide »
```

Soient n et m, respectivement, le nombre d'états et de transitions de l'automate. Puisque dans le pire cas R contient tous les états (donc n états), et que la complexité d'une recherche est de $\mathcal{O}(n + m)$, l'algorithme 1 fonctionne en temps $\mathcal{O}(n(n + m)) = \mathcal{O}(n^2 + nm)$ dans le pire cas.

En pratique, les structures de Kripke possèdent une quantité massive d'états. Une complexité quadratique n'est donc pas raisonnable. Il existe des algorithmes de complexité *linéaire* pour détecter des lassos. Une approche consiste à:

1. Effectuer un parcours en profondeur à partir des états initiaux Q_0 et numéroter le moment où l'exploration de chaque état débute et termine;
2. Ordonner les états acceptants découverts selon la terminaison de leur exploration; en d'autres mots, obtenir un post-ordre;

3. Réexplorer l'automate, cette fois à partir de chaque état $f \in F$, en post-ordre, et chercher à atteindre f_i pour identifier un cycle.

Cette procédure, décrite à l'algorithme 2, n'a pas à reconsidérer un état plus d'une fois au second parcours. En effet, on peut démontrer que s'il y a un cycle de f_j vers f_j qui traverse un état marqué précédemment par f_i, où $i < j$, alors ce cycle aurait été détecté plus tôt lors de l'exploration de f_i (car F est en post-ordre). Cela mène donc à une complexité de $\mathcal{O}(n + m)$.

Algorithme 2 : Détection linéaire de lasso par parcours post-ordre.

Entrées : Automate de Büchi $\mathcal{B} = (Q, \Sigma, \delta, Q_0, F)$
Sorties : $\mathcal{L}(\mathcal{B}) = \emptyset$?

```
// Premier parcours de B
```
parcourir $\mathcal{B}$ en profondeur à partir de Q_0
en obtenir $[f_1, f_2, \ldots, f_k]$ les états acceptants découverts, en post-ordre

```
// Second parcours pour identifier un cycle acceptant
```
$S \leftarrow \emptyset$ `// états marqués`

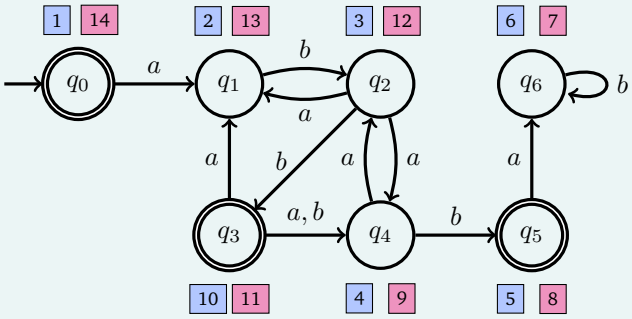
`cycle(`q, c`):`
- **ajouter** q **à** S
- **pour** $r : q \to r$
  - **si** $r = c$ **alors terminer avec** « *non vide* »
  - **sinon si** $r \notin S$ **alors** `cycle(`r, c`)`

pour $i \in [1..k]$
- **si** $f_i \notin S$ **alors**
  - `cycle(`f_i, f_i`)` `// ∃ chemin non vide de` f_i `vers` f_i`?`

terminer avec « *vide* »

Exemple.

Exécutons l'algorithme 2 sur l'automate de Büchi illustré ci-dessous. Considérons le premier parcours en profondeur. Les deux nombres encadrés en couleur à gauche et droite d'un état q_i indiquent respectivement le moment où l'exploration a débuté en q_i et terminé en q_i.



Suite à cette première fouille, les états acceptants sont ordonnés par $[q_5, q_3, q_0]$ car leur exploration a respectivement terminé aux moments $[8, 11, 14]$. La deuxième exploration à partir de q_5 visite $[q_5, q_6]$ et ainsi aucun cycle n'est identifié. La seconde exploration à partir de q_3 visite $[q_4, q_5, q_2, q_1, q_3]$ et ainsi un cycle de q_3 vers lui-même est découvert. Par conséquent, l'algorithme identifie un lasso de la forme

$$q_0 \xrightarrow{*} q_3 \xrightarrow{+} q_3.$$

Remarquons que les lettres sur les transitions ne jouent aucun rôle. De plus, d'autres ordres d'exploration sont possibles selon l'ordre dans lequel les successeurs d'un état sont considérés.

Un désavantage de l'algorithme 2 est qu'il doit explorer l'automate en entier avant d'effectuer la seconde fouille. Cela s'évite en *imbriquant* la seconde exploration dans la première. Cette approche établie par [CVWY90] est décrite à l'algorithme 3, telle que présentée dans [Esp19] [2]. Cet algorithme requiert peu de mémoire: il suffit de stocker deux bits par état dans une table de hachage pour indiquer s'il appartient aux ensembles S_1 ou S_2. De plus, la procédure peut être adaptée pour retourner explicitement un lasso lorsqu'elle en détecte un.

4.4 Sommaire

L'approche algorithmique complète est schématisée à la figure 4.2.

Remarquons que la construction de $\mathcal{T}$, $\mathcal{A}_\mathcal{T}$ et $\mathcal{A}_\mathcal{T} \cap \mathcal{A}_\varphi$, ainsi que le test du vide peuvent être être effectués *simultanément* à la volée. En effet, imaginons qu'un modèle soit fourni dans un langage de description comme Promela. Un outil de vérification comme Spin peut tester si $\mathcal{L}(\mathcal{A}_\mathcal{T} \cap \mathcal{A}_\varphi) = \emptyset$ en exécutant, par exemple, l'algorithme 3 qui génère les états de $\mathcal{A}_\mathcal{T} \cap \mathcal{A}_\varphi$ au besoin, où les états internes de $\mathcal{A}_\mathcal{T}$ sont eux-mêmes obtenus à la volée à partir du code Promela.

2. Voir [Esp19, chapitre 13], par exemple, pour une preuve formelle que cet algorithme fonctionne, ainsi qu'une présentation d'autres algorithmes de détection de lasso.

Algorithme 3 : Détection linéaire de lasso par parcours imbriqués.

Entrées : Automate de Büchi $\mathcal{B} = (Q, \Sigma, \delta, Q_0, F)$
Sorties : $\mathcal{L}(\mathcal{B}) = \emptyset$?

$S_1 \leftarrow \emptyset$; $S_2 \leftarrow \emptyset$

`dfs1(`q`):` `// Chercher un état acceptant`
- **ajouter** q **à** S_1
- **pour** $r : q \to r$
  - **si** $r \notin S_1$ **alors** `dfs1(`r`)`
- **si** $q \in F$ **alors**
  - $c \leftarrow q$
  - `dfs2(`q`)`

`dfs2(`q`):` `// Chercher un cycle vers` c
- **ajouter** q **à** S_2
- **pour** $r : q \to r$
  - **si** $r = c$ **alors terminer avec** « *non vide* »
  - **sinon si** $r \notin S_2$ **alors** `dfs2(`r`)`

pour $q \in Q_0$
- **si** $q \notin S_1$ **alors**
  - `dfs1(`q`)`

terminer avec « *vide* »

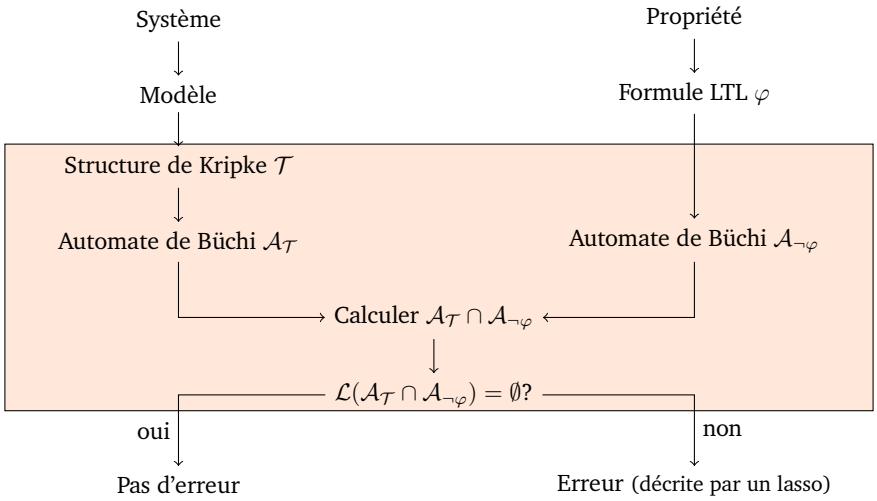


FIGURE 4.2 – Sommaire de l'approche de vérification algorithmique.

Cela permet d'identifier des erreurs sans avoir à construire la structure et les automates en entier. Dans le cas où aucune erreur n'est identifiée, on explore tout l'espace. Cependant, il existe des techniques plus sophistiquées comme la réduction par ordre partiel qui permet de réduire le nombre d'états.

Remarque.

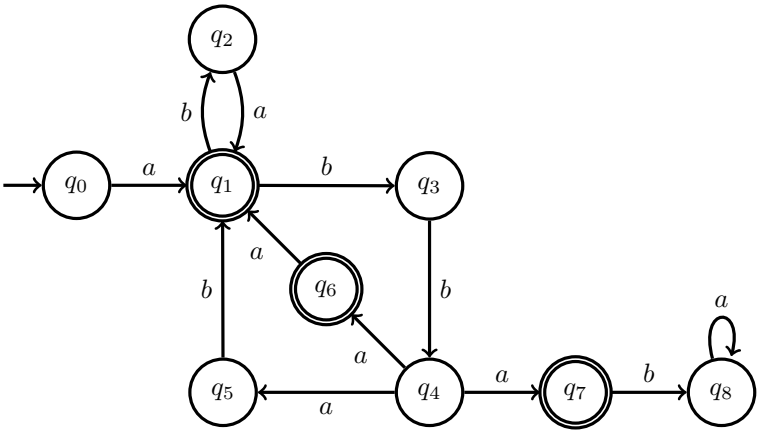
Si vous connaissez la théorie de la complexité, sachez que le problème de décision qui consiste à déterminer si une structure de Kripke satisfait une formule LTL est PSPACE-complet.

4.5 Exercices

4.1) Donnez une expression ω-régulière et un automate de Büchi pour ces formules LTL sur $AP = \{p, q\}$:

a) $\mathsf{X}p$
b) $\mathsf{FG}p$
c) $\mathsf{G}(p\ \mathsf{U}\ q)$
d) $p\ \mathsf{U}\ (\mathsf{X}q)$
e) $p \vee q$
f) $\mathsf{G}(p \rightarrow \mathsf{F}q)$
g) $\mathsf{G}[(\mathsf{X}\neg p) \rightarrow (\mathsf{F}(p \wedge q))]$

4.2) Considérons cet automate de Büchi $\mathcal{A}$:



a) Testez si $\mathcal{L}(\mathcal{A}) = \emptyset$ à l'aide des algorithmes présentés.

b) Expliquez pourquoi l'algorithme 3 n'identifie pas de lasso qui utilise q_2, et ce peu importe l'ordre dans lequel les parcours en profondeur effectuent leurs choix de successeurs lors des explorations.

Remarque.

Cela montre que l'algorithme 3 n'identifie pas nécessairement une « erreur minimale » car le plus petit lasso est

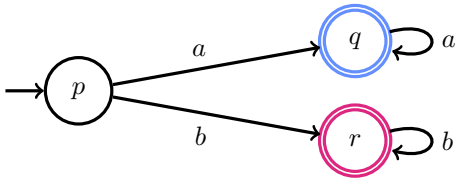
$$q_0 \xrightarrow{a} q_1 \xrightarrow{b} q_2 \xrightarrow{a} q_1.$$

(*tiré d'un exercice enseigné à l'Université technique de Munich*)

4.3) Plusieurs conversions de formules LTL vers automates de Büchi passent par un modèle intermédiaire: les *automates de Büchi généralisés*. Un tel automate est un quintuplet $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ où les quatre premières

composantes sont comme pour les automates de Büchi, mais où $F \subseteq 2^Q$ est un *ensemble* d'états acceptants. Un mot $\sigma \in \Sigma^\omega$ appartient au langage $\mathcal{L}(\mathcal{A})$ si, et seulement si, il est possible de lire σ dans $\mathcal{A}$ en visitant chaque ensemble de F infiniment souvent.

a) Quel langage est accepté par l'automate de Büchi $\mathcal{A} = (\{p, q, r\}, \{a, b\}, \delta, \{p\}, \{\{q\}, \{r\}\})$ ci-dessous?



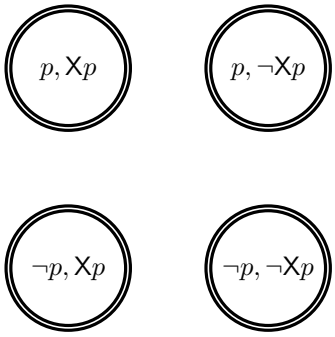
b) Donnez un automate de Büchi généralisé *déterministe* qui accepte:

$\{\sigma \in \Sigma^\omega$: chaque lettre de Σ apparaît infiniment souvent dans $\sigma.\}$

c) Montrez que les automates de Büchi généralisés sont aussi expressifs que les automates de Büchi. Autrement dit, montrez que l'on peut accepter les mêmes langages en passant d'un modèle à l'autre.

d) Appliquez votre transformation de (c) à l'automate obtenu en (b).

4.4) Soit $AP := \{p\}$. Complétez l'automate de Büchi ci-dessous en lui ajoutant des transitions de façon à ce que l'ensemble des mots infinis de 2^{AP} lisibles à partir de chaque état q soit précisément celui des mots infinis qui satisfont les deux formules inscrites dans q. Choisissez ensuite les états initiaux qui font en sorte que l'automate résultat accepte le langage $[\![\mathsf{X}p]\!]$.



(*adapté de [BK08, Fig. 5.21].*)

4.5) ★ Montrez qu'aucun automate de Büchi *déterministe* n'accepte $[\![\mathsf{FG}p]\!]$.

4.6) ★★ (*Requiert des connaissances en théorie du calcul*) Un *chemin hamiltonien* d'un graphe est un chemin qui visite tous ses sommets exactement

une fois. Montrez que le problème, qui consiste à déterminer si un graphe dirigé possède un chemin hamiltonien, se réduit en temps polynomial au problème de vérification de formules LTL.

(*tiré de [BK08]*)

Observation.

Cela montre que le problème de vérification est NP-difficile.

5

Logique temporelle arborescente (CTL)

La logique temporelle linéaire étudiée jusqu'ici ne permet pas de raisonner sur les différents *choix* possibles d'un système. Par exemple, considérons la structure de Kripke $\mathcal{T}$ illustrée à la gauche de la figure 5.1. On ne peut spécifier en LTL une propriété telle que: « il est toujours possible de revenir à l'état initial de $\mathcal{T}$ ». Cette limitation est due à la conception linéaire du temps inhérente à LTL. La logique temporelle arborescente (CTL) adopte une autre conception: le temps arborescent. Cette modélisation considère tous les chemins possibles d'un système sous une arborescence infinie. Par exemple, l'arborescence associée à $\mathcal{T}$ est illustrée à la droite de la figure 5.1. Les formules CTL permettent de décrire des propriétés qui dépendent des choix de cette arborescence. Nous étudierons cette logique dans ce chapitre.

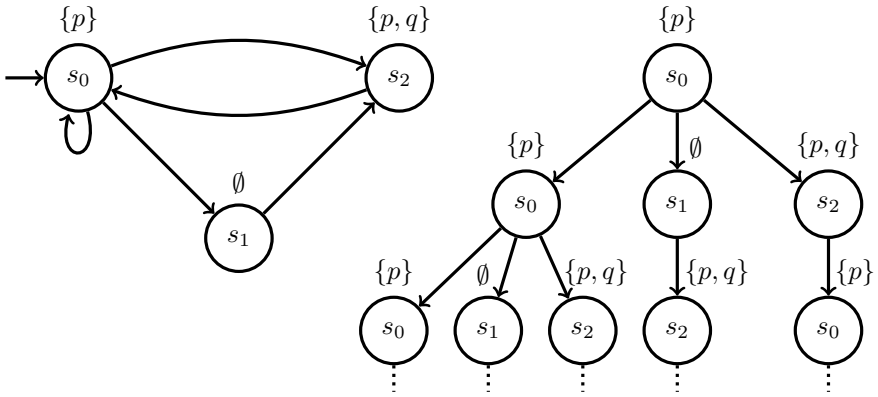


FIGURE 5.1 – Une structure de Kripke $\mathcal{T}$ (à gauche), et son arbre de calcul (à droite), c.-à-d. le « déroulement » des chemins de $\mathcal{T}$ sous forme d'arborescence.

Exemple.

La structure de Kripke $\mathcal{T}$ illustrée à la figure 5.1 ne satisfait pas la formule $\mathsf{F}(p \wedge q)$ sous LTL. Toutefois il est *toujours possible* d'éventuellement satisfaire $p \wedge q$ car chaque état de l'arbre de calcul de $\mathcal{T}$ peut atteindre s_2 en zéro ou une étape. Nous dénoterons cette propriété CTL par $\forall\mathsf{G}\exists\mathsf{F}(p \wedge q)$.

5.1 Syntaxe

Soit *AP* un ensemble de propositions atomiques. La syntaxe de la *logique temporelle arborescente (CTL)* sur *AP* est définie à partir de cette grammaire:

$$\Phi ::= \mathit{vrai} \mid p \mid (\Phi \wedge \Phi) \mid \neg\Phi \mid \exists\varphi \mid \forall\varphi$$
$$\varphi ::= \mathsf{X}\Phi \mid (\Phi \ \mathsf{U} \ \Phi)$$

où $p \in \mathit{AP}$. Nous disons qu'une telle formule Φ est une *formule d'état* et qu'une telle formule φ est une *formule de chemin*. Une *formule CTL* est une formule d'état. Ainsi, les formules de chemin ne sont qu'un concept intermédiaire pour définir les formules CTL.

Comme pour LTL, les opérateurs logiques $\vee$, $\rightarrow$, $\leftrightarrow$ et $\oplus$ se définissent naturellement à partir de $\wedge$ et $\neg$. Nous introduisons des opérateurs supplémentaires qui seront utiles afin de modéliser des propriétés:

$$\exists\mathsf{F}\Phi := \exists(\mathit{vrai} \ \mathsf{U} \ \Phi)$$
$$\forall\mathsf{F}\Phi := \forall(\mathit{vrai} \ \mathsf{U} \ \Phi)$$
$$\exists\mathsf{G}\Phi := \neg\forall\mathsf{F}\neg\Phi$$
$$\forall\mathsf{G}\Phi := \neg\exists\mathsf{F}\neg\Phi$$

Informellement, ces quatre formules pourraient se lire ainsi:

— $\exists\mathsf{F}\Phi$: « Φ peut être satisfaite »,

— $\forall\mathsf{F}\Phi$: « Φ est inévitablement satisfaite »,

— $\exists\mathsf{G}\Phi$: « Φ est possiblement toujours satisfaite »,

— $\forall\mathsf{G}\Phi$: « Φ est invariablement satisfaite ».

Exemple.

Les formules $\forall\mathsf{X}\exists\mathsf{F}(p \vee q)$ et $\exists\mathsf{F}\forall\mathsf{G}(\exists(p \ \mathsf{U} \ \neg(q \wedge r)))$ sont syntaxiquement valides, alors que $\exists\mathsf{FG}p$, $\exists(p \ \mathsf{U} \ q)\forall\mathsf{G}p$ et $\forall\neg(p \ \mathsf{U} \ q)$ ne le sont pas.

Remarque.

Nous utilisons les symboles X, F et G plutôt que les symboles $\bigcirc$, $\Diamond$ et $\Box$ utlisés dans d'autres ouvrages comme [BK08].

5.2 Sémantique

Contrairement à LTL, la sémantique d'une formule CTL dépend d'un système donné. Soit $\mathcal{T} = (S, \to, I, AP, L)$ une structure de Kripke. Nous associons un sens formel aux formules CTL en fonction de $\mathcal{T}$. Les formules d'état sont interprétées sur les états S et les formules de chemin sur les chemins infinis de $\mathcal{T}$.

Pour tout état $s \in S$, nous disons que:

$$
\begin{aligned}
& s \models \mathit{vrai}, \\
& s \models p && \overset{\text{déf}}{\iff} p \in L(s), \\
& s \models \neg\Phi && \overset{\text{déf}}{\iff} \neg(s \models \Phi), \\
& s \models \Phi_1 \wedge \Phi_2 && \overset{\text{déf}}{\iff} (s \models \Phi_1) \wedge (s \models \Phi_2), \\
& s \models \exists\varphi && \overset{\text{déf}}{\iff} \sigma \models \varphi \text{ pour un certain chemin infini de } \mathcal{T} \text{ débutant en } s, \\
& s \models \forall\varphi && \overset{\text{déf}}{\iff} \sigma \models \varphi \text{ pour tout chemin infini de } \mathcal{T} \text{ débutant en } s.
\end{aligned}
$$

Pour tout chemin infini σ de $\mathcal{T}$, nous disons que:

$$
\begin{aligned}
& \sigma \models \mathsf{X}\Phi && \overset{\text{déf}}{\iff} \sigma(1) \models \Phi, \\
& \sigma \models \Phi_1 \,\mathsf{U}\, \Phi_2 && \overset{\text{déf}}{\iff} \exists j \geq 0 : \left[(\forall 0 \leq i < j : \sigma(i) \models \Phi_1) \wedge (\sigma(j) \models \Phi_2)\right].
\end{aligned}
$$

Remarque.

Contrairement à LTL, l'opérateur U ne raisonne pas sur des suffixes infinis $\sigma[0..], \sigma[1..], \ldots, \sigma[j..]$, mais bien sur des états $\sigma(0), \sigma(1), \ldots, \sigma(j)$.

En particulier, pour tout état s et toute formule d'état Φ, nous avons:

$$
\begin{aligned}
& s \models \exists\mathsf{F}\Phi && \iff \exists \text{ chemin } \sigma \text{ débutant en } s \text{ tel que } \exists j \geq 0 : \sigma(j) \models \Phi, \\
& s \models \forall\mathsf{F}\Phi && \iff \forall \text{ chemin } \sigma \text{ débutant en } s \text{ tel que } \exists j \geq 0 : \sigma(j) \models \Phi, \\
& s \models \exists\mathsf{G}\Phi && \iff \exists \text{ chemin } \sigma \text{ débutant en } s \text{ tel que } \forall j \geq 0 : \sigma(j) \models \Phi, \\
& s \models \forall\mathsf{G}\Phi && \iff \forall \text{ chemin } \sigma \text{ débutant en } s \text{ tel que } \forall j \geq 0 : \sigma(j) \models \Phi.
\end{aligned}
$$

L'intuition derrière la sémantique des formules CTL est illustrée à la figure 5.2.

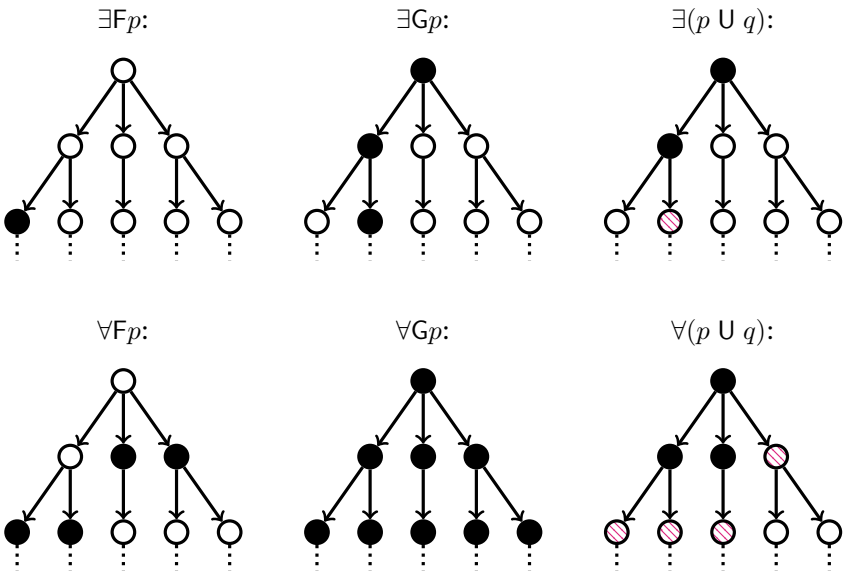


FIGURE 5.2 – Illustration de la sémantique de CTL. Chaque arborescence correspond à l'arbre de calcul d'une structure de Kripke. Les états pleins (en noir) satisfont p et les états hachurés (en magenta) satisfont q.

Exemple.

Soit $\mathcal{T}$ la structure de Kripke illustrée à la figure 5.1. Nous avons:

$$s_0 \models \exists \mathsf{F} q, \qquad s_0 \not\models \forall \mathsf{F} q,$$
$$s_0 \models \exists \mathsf{G} p, \qquad s_0 \not\models \forall \mathsf{G} p,$$
$$s_0 \models \exists (p \ \mathsf{U} \ q), \qquad s_0 \not\models \forall (p \ \mathsf{U} \ q),$$
$$s_0 \models \forall \mathsf{G} \exists \mathsf{F} (p \wedge q), \qquad s_0 \not\models \exists \mathsf{G} \forall \mathsf{F} (p \wedge q).$$

5.3 Propriétés d'un système

Soient $\mathcal{T} = (S, \rightarrow, I, AP, L)$ une structure de Kripke et Φ une formule CTL sur propositions atomiques AP. L'ensemble des états de $\mathcal{T}$ qui *satisfont* Φ est dénoté:

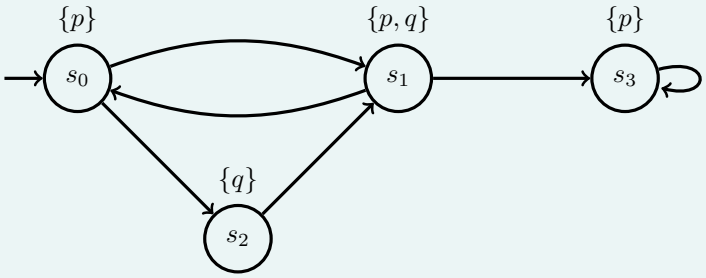
$$[\![\Phi]\!] := \{s \in S : s \models \Phi\}.$$

Nous disons que $\mathcal{T}$ *satisfait* Φ, dénoté $\mathcal{T} \models \Phi$ ssi $I \subseteq [\![\Phi]\!]$. Autrement dit:

$$\mathcal{T} \models \Phi \overset{\text{déf}}{\iff} \forall s_0 \in I \ s_0 \models \Phi.$$

Exemple.

Considérons cette structure de Kripke et ces formules tirées de [BK08, ex. 6.7].



Nous avons:

$\mathcal{T} \models \exists \mathsf{X} p$ $\quad$ $\mathcal{T} \not\models \forall \mathsf{G} p$

$\mathcal{T} \not\models \forall \mathsf{X} p$ $\quad$ $\mathcal{T} \models \exists \mathsf{F} \exists \mathsf{G} p$

$\mathcal{T} \models \exists \mathsf{G} p$ $\quad$ $\mathcal{T} \models \forall (p \mathsf{U} q)$

5.4 Équivalences

Nous disons que deux formules CTL Φ_1 et Φ_2 sont *équivalentes*, dénoté $\Phi_1 \equiv \Phi_2$, si $[\![\Phi_1]\!] = [\![\Phi_2]\!]$ pour toute structure de Kripke. Ainsi, deux formules sont équivalentes si elles ne peuvent pas être distinguées par une structure de Kripke.

Nous répertorions quelques équivalences entre formules CTL qui peuvent simplifier la spécification ou l'analyse de propriétés.

5.4.1 Distributivité

Les opérateurs temporels quantifiés se distribuent de cette façon sur les opérateurs logiques:

$$\exists \mathsf{F}(\Phi_1 \vee \Phi_2) \equiv (\exists \mathsf{F} \Phi_1) \vee (\exists \mathsf{F} \Phi_2),$$
$$\forall \mathsf{G}(\Phi_1 \wedge \Phi_2) \equiv (\forall \mathsf{G} \Phi_1) \wedge (\forall \mathsf{G} \Phi_2).$$

Démontrons l'une de ces équivalences:

Proposition 6. $\exists \mathsf{F}(\Phi_1 \vee \Phi_2) \equiv (\exists \mathsf{F} \Phi_1) \vee (\exists \mathsf{F} \Phi_2)$

Démonstration. Soient $\mathcal{T} = (S, \to, I, AP, L)$ une structure de Kripke, $s \in S$ un état de $\mathcal{T}$, et X l'ensemble des chemins infinis de $\mathcal{T}$ qui débutent en s. Nous

avons:

$$\begin{aligned}
s \models \exists \mathsf{F}(\Phi_1 \vee \Phi_2) &\iff s \models \exists(\textit{vrai}\ \mathsf{U}\ (\Phi_1 \vee \Phi_2)) \\
&\iff \exists \sigma \in X : \sigma \models (\textit{vrai}\ \mathsf{U}\ (\Phi_1 \vee \Phi_2)) \\
&\iff \exists \sigma \in X : \exists j \geq 0 : \sigma(j) \models (\Phi_1 \vee \Phi_2) \\
&\iff \exists \sigma \in X : \exists j \geq 0 : (\sigma(j) \models \Phi_1) \vee (\sigma(j) \models \Phi_2) \\
&\iff \exists \sigma \in X : \exists j \geq 0 : (\sigma(j) \models \Phi_1) \vee \\
&\qquad \exists \sigma \in X : \exists j \geq 0 : (\sigma(j) \models \Phi_2) \\
&\iff \exists \sigma \in X : \sigma \models \exists(\textit{vrai}\ \mathsf{U}\ \Phi_1) \vee \\
&\qquad \exists \sigma \in X : \sigma \models \exists(\textit{vrai}\ \mathsf{U}\ \Phi_2) \\
&\iff s \models \exists \mathsf{F}\Phi_1 \vee s \models \exists \mathsf{F}\Phi_2. \qquad \square
\end{aligned}$$

Remarque.

Notons qu'en général l'inverse n'est pas vrai:

$$\forall \mathsf{F}(\Phi_1 \vee \Phi_2) \not\equiv (\forall \mathsf{F}\Phi_1) \vee (\forall \mathsf{F}\Phi_2),$$
$$\exists \mathsf{G}(\Phi_1 \wedge \Phi_2) \not\equiv (\exists \mathsf{G}\Phi_1) \wedge (\exists \mathsf{G}\Phi_2).$$

5.4.2 Dualité

La négation interagit de cette façon avec les opérateurs temporels quantifiés:

$$\neg\exists \mathsf{X}\Phi \equiv \forall \mathsf{X}\neg\Phi,$$
$$\neg\forall \mathsf{X}\Phi \equiv \exists \mathsf{X}\neg\Phi,$$
$$\neg\exists \mathsf{G}\Phi \equiv \forall \mathsf{F}\neg\Phi,$$
$$\neg\forall \mathsf{G}\Phi \equiv \exists \mathsf{F}\neg\Phi.$$

5.4.3 Idempotence

Les opérateurs temporels quantifiés satisfont ces règles d'idempotence:

$$\forall \mathsf{G}\,\forall \mathsf{G}\Phi \equiv \forall \mathsf{G}\Phi,$$
$$\forall \mathsf{F}\,\forall \mathsf{F}\Phi \equiv \forall \mathsf{F}\Phi,$$
$$\exists \mathsf{G}\,\exists \mathsf{G}\Phi \equiv \exists \mathsf{G}\Phi,$$
$$\exists \mathsf{F}\,\exists \mathsf{F}\Phi \equiv \exists \mathsf{F}\Phi.$$

5.5 Exercices

5.1) Considérons la structure de Kripke illustrée à l'exemple de la section 5.3. Satisfait-elle ces formules?

a) $\exists(p \,\mathsf{U}\, (\neg p \wedge \exists(\neg p \,\mathsf{U}\, q)))$;

b) $\exists(p \,\mathsf{U}\, (\neg p \wedge \forall(\neg p \,\mathsf{U}\, q)))$;

c) $\forall(p \,\mathsf{U}\, (\neg p \wedge \exists(\neg p \,\mathsf{U}\, q)))$;

d) $\forall(p \,\mathsf{U}\, (\neg p \wedge \forall(\neg p \,\mathsf{U}\, q)))$.

5.2) Montrez que ces formules CTL ne sont généralement pas équivalentes:

a) $\forall\mathsf{F}(\Phi_1 \vee \Phi_2) \not\equiv (\forall\mathsf{F}\Phi_1) \vee (\forall\mathsf{F}\Phi_2)$,

b) $\exists\mathsf{G}(\Phi_1 \wedge \Phi_2) \not\equiv (\exists\mathsf{G}\Phi_1) \wedge (\exists\mathsf{G}\Phi_2)$,

c) $\neg\exists(\Phi_1 \,\mathsf{U}\, \Phi_2) \not\equiv \forall((\neg\Phi_1) \,\mathsf{U}\, (\neg\Phi_2))$.

5.3) Répondez à l'exemple 6.6 de [BK08] (sera fait en séance synchrone).

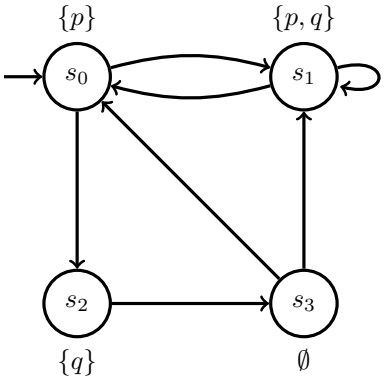
5.4) Nous disons qu'une formule LTL φ et une formule CTL Φ sont *équivalentes*, dénoté $\varphi \equiv \Phi$, lorsque $\mathcal{T} \models \varphi \iff \mathcal{T} \models \Phi$ pour toute structure de Kripke $\mathcal{T}$. Montrez que ces formules ne sont *pas* équivalentes:

a) $\varphi \coloneqq \mathsf{FG}p$ et $\Phi \coloneqq \forall\mathsf{F}\forall\mathsf{G}p$

b) $\varphi \coloneqq \mathsf{FX}p$ et $\Phi \coloneqq \forall\mathsf{F}\forall\mathsf{X}p$

5.5) Montrez maintenant que $\mathsf{GF}p \equiv \forall\mathsf{G}\forall\mathsf{F}p$.

5.6) Dites si cette structure de Kripke satisfait les formules CTL ci-dessous:



a) $\forall\mathsf{F}\exists\mathsf{G}p$

b) $\forall\mathsf{G}\exists\mathsf{X}\forall\mathsf{F}(p \wedge q)$

c) $\exists[(\neg(p \wedge q)) \,\mathsf{U}\, (\forall\mathsf{X}p)]$

6

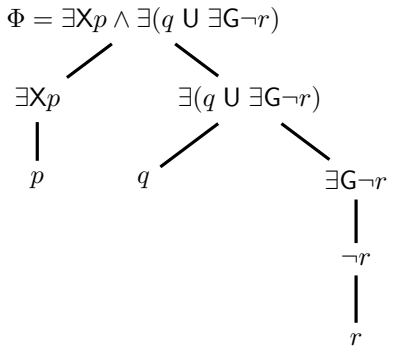
Vérification algorithmique de formules CTL

Dans ce chapitre, nous présentons une approche afin d'automatiser la vérification de formules CTL. Celle-ci s'appuie sur des manipulations d'ensembles et de calculs de points fixes. D'un point de vue calculatoire, la vérification sera plus simple que pour LTL: elle est polynomiale plutôt qu'exponentielle.

L'approche générale consiste à déterminer si $\mathcal{T} \models \Phi$, où $\mathcal{T} = (S, \to, I, AP, L)$ est une structure de Kripke et Φ une formule CTL, en calculant $[\![\Phi']\!]$ récursivement pour chaque sous-formule Φ' de Φ, jusqu'à l'obtention de $[\![\Phi]\!]$. Le calcul de ce dernier permet finalement de tester $I \subseteq [\![\Phi]\!]$. Par exemple, pour la formule $\Phi = \exists \mathsf{X} p \wedge \exists(q \ \mathsf{U} \ \exists \mathsf{G} \neg r)$, nous calculerons récursivement les ensembles suivants:

1. $[\![p]\!]$,
2. $[\![q]\!]$,
3. $[\![r]\!]$,
4. $[\![\neg r]\!]$,
5. $[\![\exists \mathsf{G} \neg r]\!]$,
6. $[\![\exists \mathsf{X} p]\!]$,
7. $[\![\exists(q \ \mathsf{U} \ (\exists G \neg r))]\!]$,
8. $[\![\Phi]\!]$.

Autrement dit, nous procéderons par induction structurelle sur l'arbre syntaxique de Φ, c.-à-d. en calculant l'ensemble d'états de chacun de ces sommets récursivement (du bas vers le haut):



6.1 Forme normale existentielle

Nous décrivons d'abord une certaine forme normale qui facilite le calcul récursif des ensembles d'états. Afin d'effectuer la vérification $\mathcal{T} \models \Phi$, nous supposerons que Φ est d'abord mise dans cette forme normale.

Une formule CTL est dite en *forme normale existentielle* si elle s'exprime avec la grammaire suivante:

$$\Phi ::= \mathit{vrai} \mid p \mid \Phi \wedge \Phi \mid \neg\Phi \mid \exists \mathsf{X}\Phi \mid \exists \mathsf{G}\Phi \mid \exists(\Phi \ \mathsf{U} \ \Phi)$$

La sémantique d'une telle formule demeure la même que celle introduite au chapitre précédent; il ne s'agit que d'une restriction syntaxique.

Toute formule CTL peut être mise en forme normale en appliquant récursivement ces équivalences:

$$\begin{aligned}
\exists \mathsf{F}\Phi &\equiv \quad \exists(\mathit{vrai} \ \mathsf{U} \ \Phi) \\
\forall \mathsf{X}\Phi &\equiv \neg\exists \mathsf{X}\neg\Phi \\
\forall \mathsf{F}\Phi &\equiv \neg\exists \mathsf{G}\neg\Phi \\
\forall \mathsf{G}\Phi &\equiv \neg\exists(\mathit{vrai} \ \mathsf{U} \ \neg\Phi) \\
\forall(\Phi_1 \ \mathsf{U} \ \Phi_2) &\equiv (\neg\exists \mathsf{G}\neg\Phi_2) \wedge (\neg\exists(\neg\Phi_2 \ \mathsf{U} \ (\neg\Phi_1 \wedge \neg\Phi_2)))
\end{aligned}$$

Exemple.

Considérons la formule $\Phi = \forall \mathsf{F}\exists[(\neg(p \wedge \forall \mathsf{G}q)) \ \mathsf{U} \ (\forall \mathsf{X}p)]$. La mise en forme normale existentielle de Φ mène à:

$$\begin{aligned}
\Phi &= \forall \mathsf{F}\exists[(\neg(p \wedge \forall \mathsf{G}q)) \ \mathsf{U} \ (\forall \mathsf{X}p)] \\
&\equiv \forall \mathsf{F}\exists[(\neg(p \wedge \forall \mathsf{G}q)) \ \mathsf{U} \ (\neg\exists \mathsf{X}\neg p)] \\
&\equiv \forall \mathsf{F}\exists[(\neg(p \wedge \neg\exists[\mathit{vrai} \ \mathsf{U} \ \neg q])) \ \mathsf{U} \ (\neg\exists \mathsf{X}\neg p)] \\
&\equiv \neg\exists \mathsf{G}\neg\exists[(\neg(p \wedge \neg\exists[\mathit{vrai} \ \mathsf{U} \ \neg q])) \ \mathsf{U} \ (\neg\exists \mathsf{X}\neg p)].
\end{aligned}$$

6.2 Calcul des états

Cherchons à calculer $[\![\Phi]\!] \subseteq S$, où Φ est une formule CTL et S est l'ensemble des états de la structure de Kripke sous considération.

6.2.1 Logique propositionnelle et opérateur temporel X

Les règles suivantes découlent directement de la sémantique de la logique:

$$\begin{aligned}
[\![\mathit{vrai}]\!] &= S, \\
[\![p]\!] &= \{s \in S : p \in L(s)\}, \\
[\![\Phi_1 \wedge \Phi_2]\!] &= [\![\Phi_1]\!] \cap [\![\Phi_2]\!], \\
[\![\neg\Phi]\!] &= S \setminus [\![\Phi]\!].
\end{aligned}$$

Remarquons que $s_0 \models \exists \mathsf{X}\Phi$ ssi il existe un chemin infini $s_0 s_1 \cdots$ tel que $s_1 \models \Phi$. Ainsi, $s_0 \models \exists \mathsf{X}\Phi$ ssi s_0 possède au moins un successeur s_1 qui satisfait Φ (en supposant que tous les chemins soient infinis). Ainsi, nous avons la règle:

$$[\![\exists \mathsf{X}\Phi]\!] = \{s \in S : \mathrm{Post}(s) \cap [\![\Phi]\!] \neq \emptyset\}.$$

6.2.2 Opérateur temporel G

Les formules de la forme « $\exists \mathsf{G}\Phi$ » et « $\exists(\Phi_1 \,\mathsf{U}\, \Phi_2)$ » sont plus complexes car elles ne dépendent pas de critères locaux. Considérons d'abord la première forme. Posons $T \coloneqq [\![\exists \mathsf{G}\Phi]\!]$. Nous avons $s_0 \in T$ ssi il existe un chemin infini $s_0 s_1 \cdots$ tel que $s_i \models \Phi$ pour tout $i \in \mathbb{N}$. Ainsi, si $s_0 \in T$, alors il existe $s_1 \in \mathrm{Post}(s_0)$ tel que $s_1 \in T$, puis un état $s_2 \in \mathrm{Post}(s_1)$ tel que $s_2 \in T$ et ainsi de suite. Ainsi, *tous* les états de T doivent satisfaire la propriété $\mathrm{Post}(s) \cap T \neq \emptyset$. Plus formellement:

Proposition 7. *L'ensemble* $[\![\exists \mathsf{G}\Phi]\!]$ *est le plus grand ensemble* $T \subseteq S$ *tel que* $T \subseteq [\![\Phi]\!]$ *et* $s \in T \implies \mathrm{Post}(s) \cap T \neq \emptyset$.

Démonstration. Montrons d'abord que T est bien défini, c.-à-d. qu'il existe bien *un plus grand* ensemble. Pour ce faire, il suffit de démontrer que les ensembles qui satisfont la propriété sont clos sous union. Soient $T_1, T_2 \subseteq [\![\Phi]\!]$ tels que

$$\begin{aligned}
s \in T_1 &\implies \mathrm{Post}(s) \cap T_1 \neq \emptyset, \\
s \in T_2 &\implies \mathrm{Post}(s) \cap T_2 \neq \emptyset.
\end{aligned}$$

Posons $T' \coloneqq T_1 \cup T_2$. Soit $s \in T'$. Nous avons $s \in T_i$ pour un certain $i \in \{1, 2\}$. Ainsi, $\mathrm{Post}(s) \cap T_i \neq \emptyset$ et par conséquent $\mathrm{Post}(s) \cap T' \neq \emptyset$.

Montrons maintenant que $[\![\exists \mathsf{G}\Phi]\!] = T$.

$[\![\exists \mathsf{G}\Phi]\!] \subseteq T$. Soit $s_0 \in [\![\exists \mathsf{G}\Phi]\!]$. Par définition, il existe un chemin infini $s_0 s_1 \cdots$ tel que $s_i \models \Phi$ pour tout $i \in \mathbb{N}$. Posons $S' \coloneqq \{s_i : i \in \mathbb{N}\}$. Si $S' \subseteq T$, alors nous avons terminé puisqu'en particulier $s_0 \in T$. Afin d'obtenir une contradiction, supposons que $S' \not\subseteq T$. Posons $T' \coloneqq T \cup S'$. Remarquons que, par définition de T et S', l'ensemble T' satisfait:

$$T' \subseteq [\![\Phi]\!] \text{ et } s \in T' \implies \mathrm{Post}(s) \cap T' \neq \emptyset.$$

Cela contredit la maximalité de T puisque $T' \supset T$. Ainsi, $S' \subseteq T$.

$T \subseteq [\![\exists \mathsf{G}\Phi]\!]$. Soit $s_0 \in T$. Par définition de T, nous avons $s_0 \models \Phi$ et il existe $s_1 \in \text{Post}(s_0)$ tel que $s_1 \in T$. Par le même raisonnement, nous avons $s_1 \models \Phi$ et il existe $s_2 \in \text{Post}(s_1)$ tel que $s_2 \in T$. En continuant ce processus, nous obtenons un chemin infini $s_0 s_1 s_2 \cdots$ tel que $s_i \in T$ pour tout $i \in \mathbb{N}$, ce qui implique $s_0 \in [\![\exists \mathsf{G}\Phi]\!]$ car $T \subseteq [\![\Phi]\!]$. □

L'algorithme 4 calcule l'ensemble $[\![\exists \mathsf{G}\Phi]\!]$ en exploitant la caractérisation de plus grand point fixe de la proposition 7.

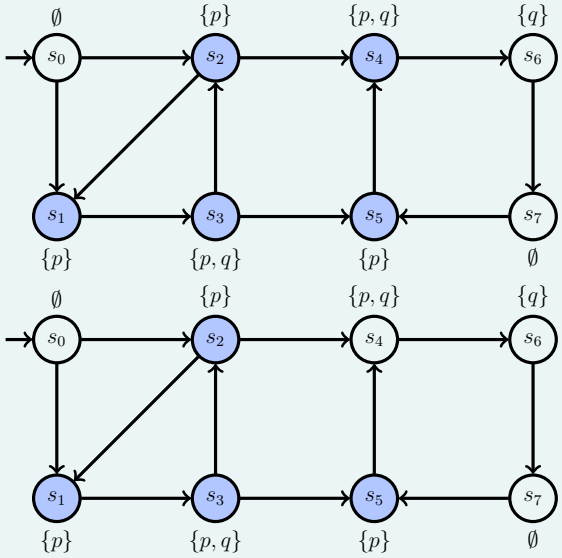
Algorithme 4 : Calcule $[\![\exists \mathsf{G}\Phi]\!]$ à partir de $[\![\Phi]\!]$.

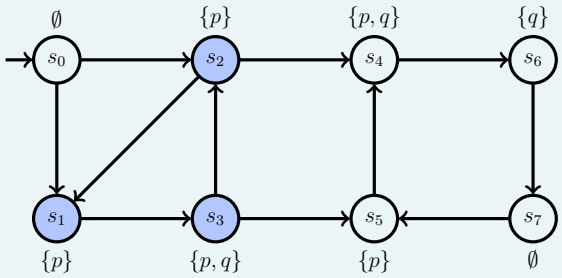
Entrées : ensemble $[\![\Phi]\!]$
Sorties : ensemble $[\![\exists \mathsf{G}\Phi]\!]$

$T \leftarrow [\![\Phi]\!]$ `// États pouvant satisfaire ∃GΦ`
`// Raffiner T`
tant que $\exists s \in T : \text{Post}(s) \cap T = \emptyset$
  retirer s **de** T
retourner T

Exemple.

Exécutons l'algorithme 4 sur la formule $\Phi = \exists \mathsf{G} p$ et la structure de Kripke ci-dessous. Nous marquons les sommets qui appartiennent à T au début de chaque itération:





Ainsi, $[\![\Phi]\!] = \{s_1, s_2, s_3\}$. Le système ne satisfait donc pas Φ car $s_0 \notin [\![\Phi]\!]$.

6.2.3 Opérateur temporel U

Considérons maintenant les formules de la forme « $\Phi = \exists(\Phi_1 \mathsf{U} \Phi_2)$ ». Posons $T \coloneqq [\![\Phi]\!]$. Par définition, $s_0 \in T$ ssi il existe un chemin infini σ tel que

$$\exists j \in \mathbb{N} : \forall 0 \leq i < j : \sigma(i) \models \Phi_1 \text{ et } \sigma(j) \models \Phi_2.$$

Ainsi, $s_0 \in T$ ssi il y a un chemin infini $s_0 s_1 \cdots s_j$ tel que $s_j \in [\![\Phi_2]\!]$ et $s_i \in [\![\Phi_1]\!]$ pour tout $0 \leq i < j$. Il suffit donc d'identifier d'abord les états de $[\![\Phi_2]\!]$, puis tous leurs prédécesseurs (immédiats ou non) qui peuvent les atteindre sans quitter $[\![\Phi_1]\!]$. Plus formellement:

Proposition 8. *L'ensemble* $[\![\exists(\Phi_1 \mathsf{U} \Phi_2)]\!]$ *est le plus petit ensemble* $T \subseteq S$ *tel que* $[\![\Phi_2]\!] \subseteq T$ *et* $(s \in [\![\Phi_1]\!] \wedge \text{Post}(s) \cap T \neq \emptyset) \implies s \in T$.

L'algorithme 5 calcule l'ensemble $[\![\exists(\Phi_1 \mathsf{U} \Phi_2)]\!]$ en exploitant la caractérisation de plus petit point fixe de la proposition 8.

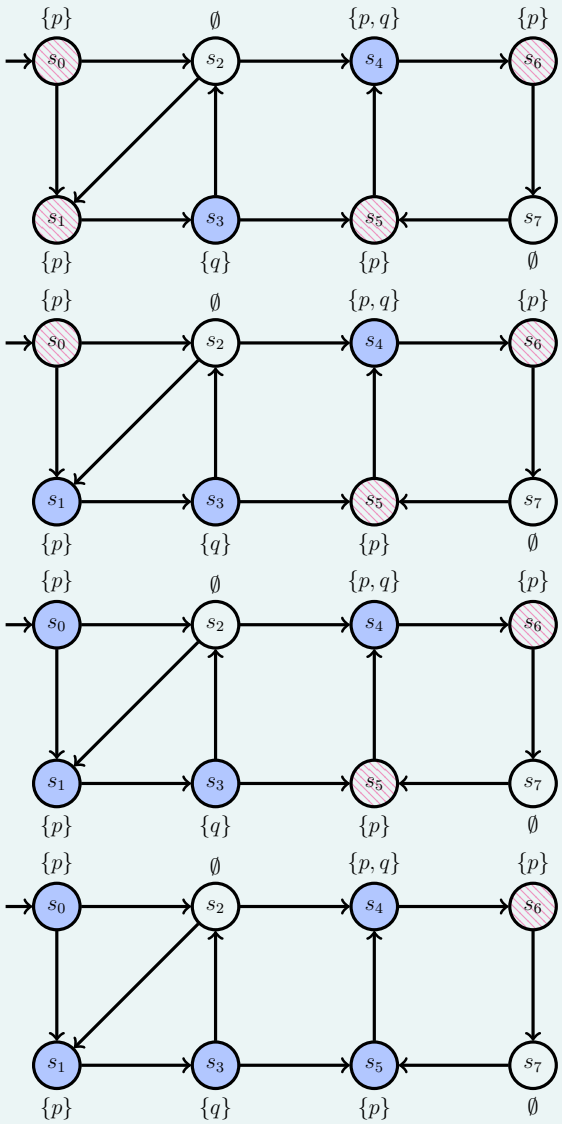
Algorithme 5 : Calcule $[\![\exists(\Phi_1 \mathsf{U} \Phi_2)]\!]$ à partir de $[\![\Phi_1]\!]$ et $[\![\Phi_2]\!]$.

Entrées : ensembles $[\![\Phi_1]\!]$ et $[\![\Phi_2]\!]$
Sorties : ensemble $[\![\exists(\Phi_1 \mathsf{U} \Phi_2)]\!]$

$T \leftarrow [\![\Phi_2]\!]$ `// États qui satisfont immédiatement` $\exists(\Phi_1 \mathsf{U} \Phi_2)$
$C \leftarrow [\![\Phi_1]\!] \setminus T$ `// États pouvant aussi satisfaire` $\exists(\Phi_1 \mathsf{U} \Phi_2)$
`// Élargir` T `à partir de` C
tant que $\exists s \in C : \text{Post}(s) \cap T \neq \emptyset$
  ajouter s **à** T
  retirer s **de** C
retourner T

Exemple.

Exécutons l'algorithme 5 sur la formule $\Phi = \exists(p \ \mathsf{U} \ q)$ et la structure de Kripke ci-dessous. Nous marquons les sommets qui font partie de T (plein) et C (rayé) au début de chaque itération:



Ainsi, $[\![\Phi]\!] = \{s_0, s_1, s_3, s_4, s_5\}$. Le système satisfait donc Φ car $s_0 \in [\![\Phi]\!]$.

6.2.4 Algorithme complet

L'algorithme 6 regroupe toutes les caractérisations décrites précédemment afin de vérifier automatiquement si une structure de Kripke satisfait une certaine spécification CTL. Comme une implémentation naïve de chaque étape prend un temps quadratique, et que le nombre de sous-formules est linéaire par rapport à la taille de la formule initiale, l'algorithme fonctionne en temps cubique dans le pire cas.

Algorithme 6 : Algorithme de vérification.

Entrées : structure de Kripke $\mathcal{T} = (S, \to, I, AP, L)$ et formule CTL Φ
Sorties : $T \models \Phi$?

```
états(Φ):
    si Φ = vrai alors retourner S
    sinon si Φ = p ∈ AP alors retourner {s ∈ S : p ∈ L(s)}
    sinon si Φ = Φ1 ∧ Φ2 alors retourner états(Φ1) ∩ états(Φ2)
    sinon si Φ = ¬Φ' alors retourner S \ états(Φ')
    sinon si Φ = ∃XΦ' alors
        T ← états(Φ')
        retourner {s ∈ S : Post(s) ∩ T ≠ ∅}
    sinon si Φ = ∃GΦ' alors
        T' ← états(Φ')
        retourner T obtenu par l'algorithme 4 sur entrée T'
    sinon si Φ = ∃(Φ1 U Φ2) alors
        T1 ← états(Φ1)
        T2 ← états(Φ2)
        retourner T obtenu par l'algorithme 5 sur entrée (T1, T2)
normaliser Φ
T ← états(Φ)
retourner I ⊆ T
```

6.2.5 Optimisations

Il est possible d'adpter l'algorithme 6 afin d'opérer directement sur une formule CTL générale, plutôt que de la mettre en forme normale existentielle. De plus, chaque étape s'implémente en temps linéaire plutôt que quadratique. Ainsi, la vérification d'une formule CTL Φ requiert un temps de $\mathcal{O}((|S| + |\to|) \cdot |\Phi|)$ dans le pire cas. En particulier, pour une formule de taille constante, la vérification s'effectue en temps linéaire.

Nous présentons deux telles optimisations afin d'illustrer les idées générales. Voyons d'abord comment calculer $[\![\exists \mathsf{G}\Phi]\!]$ en temps linéaire à partir de $[\![\Phi]\!]$. Soit

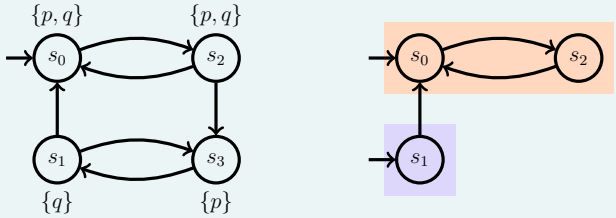
G_Φ le graphe induit par la structure de Kripke restreinte à ses états qui appartiennent à $[\![\Phi]\!]$. Remarquons que

$$s \models \exists \mathsf{G}\Phi \iff s \in G_\Phi \text{ et } s \text{ peut atteindre une composante fortement connexe non triviale de } G_\Phi, \quad (6.1)$$

où « non triviale » signifie que la composante contient au moins une transition. Ainsi, on peut calculer $[\![\exists \mathsf{G}\Phi]\!]$ en identifiant les composantes fortement connexes de G_Φ (ordonnées topologiquement). Cette opération se réalise en temps linéaire, par ex. à l'aide de l'algorithme de Tarjan ou de Kosaraju.

Exemple.

Considérons la structure de Kripke ci-dessous (gauche) et la formule $\Phi = \exists \mathsf{G}q$. Le sous-graphe G_q illustré ci-dessous (droite) est constitué des composantes fortement connexes $\{s_0, s_2\}$ et $\{s_1\}$:



Par l'équivalence (6.1), $[\![\Phi]\!] = \{s_0, s_1, s_2\}$. En effet, la composante $\{s_0, s_2\}$ est non triviale, et la composante triviale $\{s_1\}$ peut atteindre une composante non triviale. La structure satisfait donc Φ.

Montrons que l'équivalence décrite est correcte:

Proposition 9. *L'équivalence* (6.1) *est valide.*

Démonstration. $\Rightarrow$) Soit $s_0 \in [\![\exists \mathsf{G}\Phi]\!]$. Il existe un chemin infini $s_0 s_1 \cdots$ tel que $s_i \models \Phi$ pour tout $i \in \mathbb{N}$. Tous les états de ce chemin appartiennent au sous-graphe G_Φ. De plus, comme le nombre d'états est fini et que le chemin est infini, il existe des indices $i < j$ tels que $s_i = s_j$. Par conséquent,

$$s_0 \xrightarrow{*} s_i \xrightarrow{+} s_i,$$

ce qui signifie que s_0 peut atteindre la composante fortement connexe de s_i, qui n'est pas triviale puisqu'elle possède au moins une transition (car $i < j$).

$\Leftarrow$) Soit $s_0 \in G_\Phi$ un état qui peut atteindre une composante fortement connexe non triviale de G_Φ. Soit $s_0 s_1 \cdots s_i$ un tel chemin fini. Comme la composante de s_i est non triviale, s_i peut s'atteindre via un chemin non vide. Autrement dit, il existe un chemin non vide $s_i s_{i+1} \cdots s_j$ où $s_j = s_i$. Ainsi, le chemin infini $s_0 s_1 \cdots s_{i-1} (s_i s_{i+1} \cdots s_j)^\omega$ montre que $s_0 \models \exists \mathsf{G}\Phi$. □

Voyons maintenant comment gérer une formule de la forme $\forall \mathsf{G}\Phi$ sans passer par la forme normale existentielle. Rappelons que $s \models \forall \mathsf{G}\Phi$ ssi tous les chemins qui débutent dans l'état s satisfont Φ. Ainsi, on peut initialement considérer tous les états de $[\![\Phi]\!]$, puis itérativement raffiner l'ensemble en retirant les états qui possèdent au moins un successeur qui n'en fait pas partie. Cette procédure est décrite à l'algorithme 7.

Algorithme 7 : Calcule $[\![\forall \mathsf{G}\Phi]\!]$ à partir de $[\![\Phi]\!]$.

Entrées : ensemble $[\![\Phi]\!]$
Sorties : ensemble $[\![\forall \mathsf{G}\Phi]\!]$

$T \leftarrow [\![\Phi]\!]$ `// États pouvant satisfaire ∀GΦ`

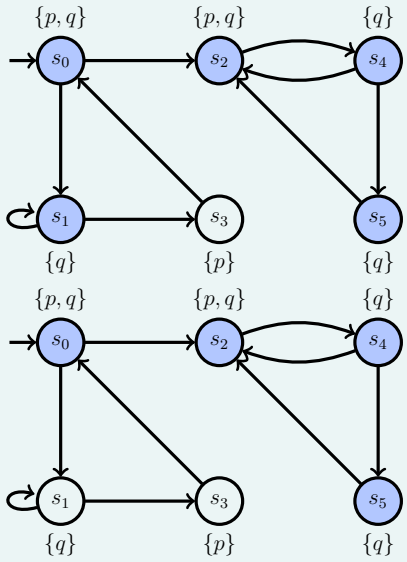
`// Raffiner T`

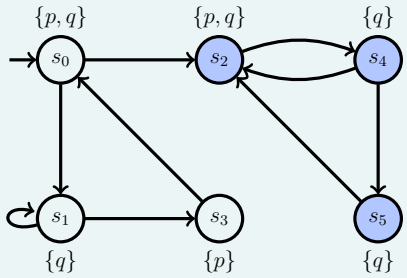
tant que $\exists s \in T : \mathrm{Post}(s) \not\subseteq T$
  retirer s **de** T

retourner T

Exemple.

Exécutons l'algorithme 7 sur la formule $\Phi = \forall \mathsf{G} q$ et la structure de Kripke ci-dessous. Nous marquons les sommets qui appartiennent à T au début de chaque itération:





Ainsi, $[\![\Phi]\!] = \{s_2, s_4, s_5\}$. Le système ne satisfait donc pas Φ car $s_0 \notin [\![\Phi]\!]$.

6.2.6 Outils

Il existe un certain nombre d'outils de vérifications pour CTL. En particulier, le logiciel de vérification NuSMV, développé conjointement à l'Université Carnegie-Mellon, de Gênes et de Trente, permet de vérifier des spécifications CTL (et LTL) de systèmes finis dans un langage de description de haut niveau. NuSMV est un descendant de SMV, un outil de vérification historique qui a notamment permis de vérifier formellement des composantes matérielles de grande taille et d'identifier des bogues dans le standard IEEE Future+.

Voici un exemple de modélisation d'une structure de Kripke et de la propriété $\forall \mathsf{X}(p \wedge \exists(p \,\mathsf{U}\, q))$:

```
MODULE main
VAR
  etat : {s0, s1, s2, s3};
ASSIGN
  init(etat) := s0;
  next(etat) :=
    case
      etat = s0 : {s1, s2};
      etat = s1 : s3;
      etat = s2 : {s0, s1, s2};
      etat = s3 : s2;
    esac;
DEFINE
  p := etat = s0 | etat = s1 | etat = s2;
  q := etat = s1;

SPEC
  AX (p & E [p U q])
```

Cet exemple utilise le langage de NuSMV afin de représenter une structure de Kripke abstraite. En pratique, il est plus utile d'utiliser le langage de haut

niveau afin de décrire des structures plus complexes, qui sont ensuite converties en structures de Kripke à l'interne (comme pour Spin et Promela).

Notons qu'un module de l'analyseur statique Infer de Facebook permet de raisonner sur les arbres de syntaxe abstraite de différents langages de programmation à l'aide d'une variante de CTL.

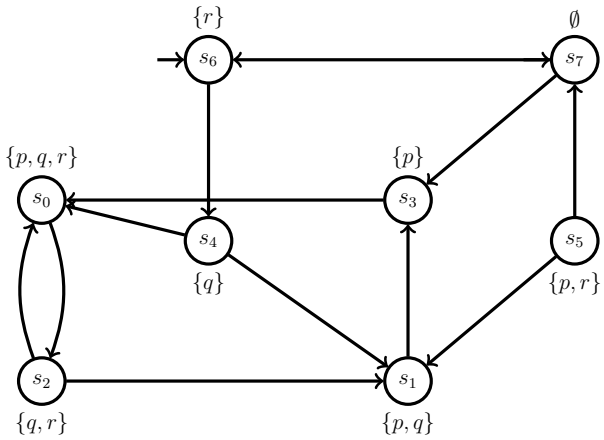
6.3 Exercices

6.1) Calculer algorithmiquement les états de la structure de Kripke ci-dessous qui satisfont ces propriétés:

a) $\Phi_1 \coloneqq \exists \mathsf{F}(p \leftrightarrow r \wedge p \oplus q)$

b) $\Phi_2 \coloneqq \exists \mathsf{G} q$

c) $\forall \mathsf{X}(\neg\Phi_1 \vee \neg\Phi_2)$



(*adapté de [BK08, ex. 6.26]*)

6.2) Expliquez pourquoi la mise en forme normale existentielle peut mener à une explosion exponentielle de la taille d'une formule.

6.3) Décrivez des algorithmes afin de vérifier directement des formules de la forme $\exists \mathsf{F}\Phi$, $\forall \mathsf{F}\Phi$, $\forall \mathsf{X}\Phi$ et $\forall(\Phi_1 \mathsf{U} \Phi_2)$.

6.4) Vérifiez les propriétés de la structure Kripke décrite par `kripke.smv` et confirmez votre résultat avec NuSMV.

6.5) Spécifiez les propriétés de la pile modélisée par `pile_sans_solutions.smv` et vérifiez-les avec NuSMV.

6.6) ★ Démontrez la proposition 8.

6.7) Comme pour LTL, certaines spécifications ne font du sens en pratique que sous hypothèse d'une notion d'équité. Contrairement à LTL, il est généralement impossible de spécifier directement l'équité en CTL. Une vérification qui tient compte de l'équité requiert donc de nouveaux algorithmes.
Soient $S_1, \ldots, S_n \subseteq S$ des ensembles d'états. Nous disons qu'un chemin infini $s_0 s_1 \cdots$ est *équitable* s'il visite chaque ensemble S_i infiniment souvent. De plus, nous disons que:

— $s \models \exists_{\text{équit}} \mathsf{G}\Phi$ ssi il existe un chemin infini équitable qui débute en s et qui satisfait $\mathsf{G}\Phi$.

— $s \models \exists_{\text{équit}}(\Phi_1 \mathsf{U} \Phi_2)$ ssi il existe un chemin infini équitable qui débute en s et qui satisfait $\Phi_1 \mathsf{U} \Phi_2$.

a) Donnez un algorithme qui calcule $[\![\exists_{\text{équit}} \mathsf{G}\Phi]\!]$ étant donné $[\![\Phi]\!]$. Pensez d'abord au cas où $n = 1$, puis généralisez votre approche à $n > 1$.

b) Exprimez $\exists_{\text{équit}}(\Phi_1 \mathsf{U} \Phi_2)$ à l'aide de $\exists\mathsf{U}$ et $\exists_{\text{équit}}\mathsf{G}$. Déduisez-en un algorithme qui calcule $[\![\exists_{\text{équit}}(\Phi_1 \mathsf{U} \Phi_2)]\!]$.

6.8) Expliquez comment identifier algorithmiquement, en temps linéaire, tous les sommets d'un graphe qui peuvent atteindre une composante fortement connexe non triviale.

7

Vérification symbolique : diagrammes de décision binaire

Au chapitre 6, nous avons vu qu'il est possible de vérifier qu'un système $\mathcal{T}$ satisfait une spécification CTL Φ en temps polynomial, plus précisément $\mathcal{O}((\#\text{états}+\#\text{transitions})\cdot|\Phi|)$. Toutefois, les algorithmes de vérification manipulent des ensembles d'états souvent gigantesques dû à l'explosion combinatoire causée par le passage d'un modèle de haut niveau vers une structure de Kripke. Ainsi, une représentation « énumérative », où *tous* les éléments d'un ensemble sont explicitement stockés, s'avère peu efficace en pratique. Dans ce chapitre, nous voyons une structure de données qui représente des ensembles d'états de façon *symbolique*. Une telle représentation permet de réduire la quantité de données tout en offrant généralement de bonnes propriétés algorithmiques. Elle est utilisée à l'interne, par exemple, par l'outil de vérification NuSMV. Ce chapitre se base fortement sur l'excellente présentation de [And98]. En particulier, nous suivons donc sa terminologie anglophone afin d'en faciliter sa lecture.

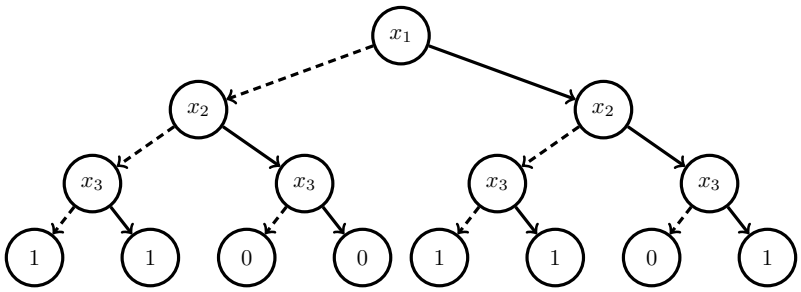


FIGURE 7.1 – Arbre de décision de l'expression booléenne $(x_1 \vee \neg x_2) \wedge (\neg x_2 \vee x_3)$. Une arête tiretée (resp. pleine) sortant d'un sommet x_i indique la décision où la variable x_i prend la valeur *faux* (resp. *vrai*). Les sommets 0 et 1 correspondent respectivement aux valeurs booléennes *faux* et *vrai*.

À titre d'exemple, considérons une structure de Kripke dont l'ensemble des états est $S = \{s_0, s_1, \ldots, s_7\}$, et telle que $[\![p]\!] = \{s_0, s_1, s_4, s_5, s_7\}$. Chaque état de S peut être représenté par la représentation binaire de son indice. Ainsi,

$$\begin{aligned}[\![p]\!] &= \{000, 001, 100, 101, 111\} \\ &= \{x_1x_2x_3 \in \{0,1\}^3 : (x_1 \vee \neg x_2) \wedge (\neg x_2 \vee x_3)\}.\end{aligned}$$

L'ensemble $[\![p]\!]$ se représente donc symboliquement par l'expression $(x_1 \vee \neg x_2) \wedge (\neg x_2 \vee x_3)$, ou alternativement par l'arbre de décision illustré à la figure 7.1.

Ces deux représentations ne sont pas pratiques parce que d'une part les expressions booléennes souffrent d'une complexité calculatoire élevée, et d'autre part de tels arbres de décision sont de taille 2^n où n est le nombre de variables. Cependant, un arbre de décision contient généralement une grande quantité de redondance. Par exemple, l'arbre de la figure 7.1 possède plusieurs occurrences des sommets 0 et 1, deux sous-arbres isomorphes étiquetés par x_3, en plus de choix inutiles: par ex. lorsque $x_1 = x_2 =$ *faux*, le résultat est *vrai* indépendamment de la valeur de x_3.

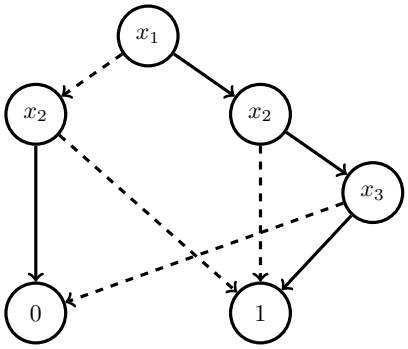


FIGURE 7.2 – Diagramme de décision binaire de l'expression $(x_1 \vee \neg x_2) \wedge (\neg x_2 \vee x_3)$ obtenu à partir de l'arbre de décision de la figure 7.1.

En éliminant cette redonance de l'arbre de décision, nous obtenons un graphe dirigé acyclique équivalent tel qu'illustré à la figure 7.2. Ce graphe résultant est un diagramme de décision binaire. Plus formellement:

Définition 1. Un *diagramme de décision binaire (réduit et ordonné)*, abrégé par le terme anglophone *BDD*, est un graphe dirigé acyclique B tel que:

- B possède des variables ordonnées $x_1 < x_2 < \cdots < x_n$;
- B contient deux sommets spéciaux, nommés 0 et 1, qui ne possèdent pas de succeseurs;
- chaque autre sommet u de B est associé à une variable $var(u)$, et possède deux successeurs $lo(u)$ et $hi(u)$;

— chaque chemin de B respecte l'ordre des variables, autrement dit:

$$u \to v \implies \mathit{var}(u) < \mathit{var}(v);$$

— chaque sommet est unique, autrement dit:

$$(\mathit{var}(u) = \mathit{var}(v) \wedge \mathit{lo}(u) = \mathit{lo}(v) \wedge \mathit{hi}(u) = \mathit{hi}(v)) \implies u = v;$$

— les sommets ne sont pas redondants, autrement dit: $\mathit{lo}(u) \neq \mathit{hi}(u)$.

Chaque sommet u d'un BDD à n variables représente naturellement une expression booléenne $f_u \colon \{0,1\}^n \to \{0,1\}$, où la valeur de $f_u(x_1, \ldots, x_n)$ est obtenue en suivant le chemin associé aux variables de u vers le sommet 0 ou 1. Ainsi, un BDD représente *plusieurs* expressions booléennes à la fois.

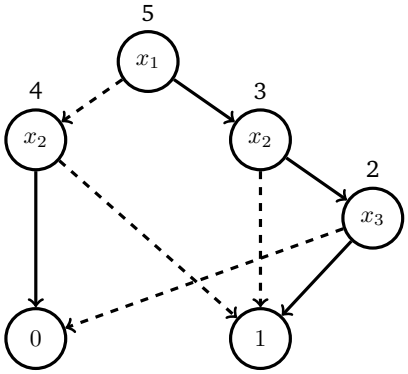


FIGURE 7.3 – Exemple d'un BDD sur trois variables $x_1 < x_2 < x_3$.

Exemple.

Pour le BDD illustré à la figure 7.3, nous avons:

$$\begin{aligned}
f_0 &= \mathit{faux} && \equiv \emptyset,\\
f_1 &= \mathit{vrai} && \equiv \{000, 001, 010, 011, 100, 101, 110, 111\},\\
f_2 &= x_3 && \equiv \{001, 011, 101, 111\},\\
f_3 &= \neg x_2 \vee x_3 && \equiv \{000, 001, 100, 101, 011, 111\},\\
f_4 &= \neg x_2 && \equiv \{000, 001, 100, 101\},\\
f_5 &= (x_1 \vee \neg x_2) \wedge (\neg x_2 \vee x_3) && \equiv \{000, 001, 100, 101, 111\}.
\end{aligned}$$

Il est possible de démontrer que les BDDs offrent une représentation canonique des fonctions booléennes. Autrement dit, chaque fonction booléenne est représentée par un unique BDD (pour un ordre fixe des variables). Ainsi:

Lemme 1. *Soit t une fonction booléenne sur les variables ordonnées $x_1 < x_2 < \cdots < x_n$. Il existe un unique sommet de BDD tel que $f_u = t$.*

Remarque.

Certains ouvrages utilisent plutôt le terme *multi-BDD* pour référer au fait qu'il n'y a pas de sommet de départ particulier et que chaque sommet calcule sa propre fonction booléenne.

7.1 Représentation de BDD

Chaque sommet d'un BDD est uniquement déterminé par sa variable et ses deux successeurs. Ainsi, un BDD peut être stocké sous forme de tableau associatif où chaque entrée indique cette information.

Par exemple, le BDD de la figure 7.3 se représente par ce tableau:

u	$var(u)$	$lo(u)$	$hi(u)$
5	x_1	4	3
4	x_2	1	0
3	x_2	1	2
2	x_3	0	1
1	x_∞	—	—
0	x_∞	—	—

Notons que les sommets spéciaux 0 et 1 ne possèdent pas de variable. Nous pouvons tout de même considérer qu'ils sont étiquetés par une variable artificielle x_∞ plus grande que toutes les autres variables. En pratique, il n'est pas nécessaire de représenter ces deux entrées explicitement car elles seront toujours les mêmes.

Algorithme 8 : Création d'un sommet de BDD.

Entrées : indice $i \in [1..n]$ et sommets ℓ et h
Sorties : sommet u t.q. $\neg x_i$ mène à ℓ et x_i mène à h

```
make(i, ℓ, h):
    si ℓ = h alors retourner ℓ        // Ne pas créer de redondance
    sinon si le BDD contient déjà un sommet (x_i, ℓ, h) alors
        retourner u où u est le sommet associé à (x_i, ℓ, h)
    sinon
        u ← prochain identifiant numérique
        ajouter u : (x_i, ℓ, h) au BDD
        retourner u
```

L'ajout d'un nouveau sommet u tel que $\mathit{var}(u) = x_i$, $\mathit{lo}(u) = \ell$ et $\mathit{hi}(u) = h$ s'effectue à l'aide de l'algorithme 8. Nous supposons qu'il est possible d'obtenir $(\mathit{var}(u), \mathit{lo}(u), \mathit{hi}(u))$ à partir de u en temps constant, et inversement qu'il est possible de déterminer l'identifiant d'un sommet étiqueté par (x_i, ℓ, h) en temps constant. En pratique, cette hypothèse se traduit par l'usage de tables de hachage (bien qu'elles fournissent techniquement un temps constant *amorti*).

7.2 Construction de BDD

Voyons maintenant comment construire un BDD à partir d'une expression booléenne. Soient t une expression booléenne, $b \in \{0, 1\}$ et x_i une variable. Nous dénotons par $t[b/x_i]$ l'expression booléenne obtenue en remplaçant chaque occurrence de x_i par b dans l'expression t. Par exemple, $(x_1 \vee x_2)[0/x_1] = x_2$ et $(x_1 \wedge x_2)[0/x_1] = \mathit{faux}$.

Algorithme 9 : Conversion d'une expression booléenne vers un BDD.

Entrées : une expression booléenne t
Sorties : sommet u tel que $f_u = t$

```
build(t):
    build'(t, i):
        si t = faux alors retourner 0
        sinon si t = vrai alors retourner 1
        sinon
            v0 ← build'(t[0/x_i], i + 1)
            v1 ← build'(t[1/x_i], i + 1)
            retourner make(x_i, v0, v1)
    retourner build'(t, 1)
```

Étant donné une expression booléenne t, l'unique sommet représentant t se calcule à l'aide de l'algorithme 9. La figure 7.4 montre les sommets obtenus lors de l'appel de `build`$(x_1 \vee \neg x_2)$ (en cyan plein) suivi de l'appel de `build`$(\neg x_2 \vee x_3)$ (en magenta rayé). Les sommets 3 et 5 représentent respectivement les expressions booléennes $x_1 \vee \neg x_2$ et $\neg x_2 \vee x_3$. Notons que des appels subséquents à `build` pourraient réutiliser certains sommets existants.

7.3 Manipulation de BDD

7.3.1 Opérations logiques binaires

À partir de sommets u_1 et u_2, on peut construire un sommet qui représente la fonction $f_{u_1} \bigcirc f_{u_2}$ où $\bigcirc$ est une opération logique binaire telle que $\wedge$, $\vee$ ou $\oplus$. Un algorithme qui accomplit cette tâche est décrit à l'algorithme 10. Informellement, cet algorithme calcule $u_1 \bigcirc u_2$ en calculant récursivement $\mathit{lo}(u_1) \bigcirc \mathit{lo}(u_2)$

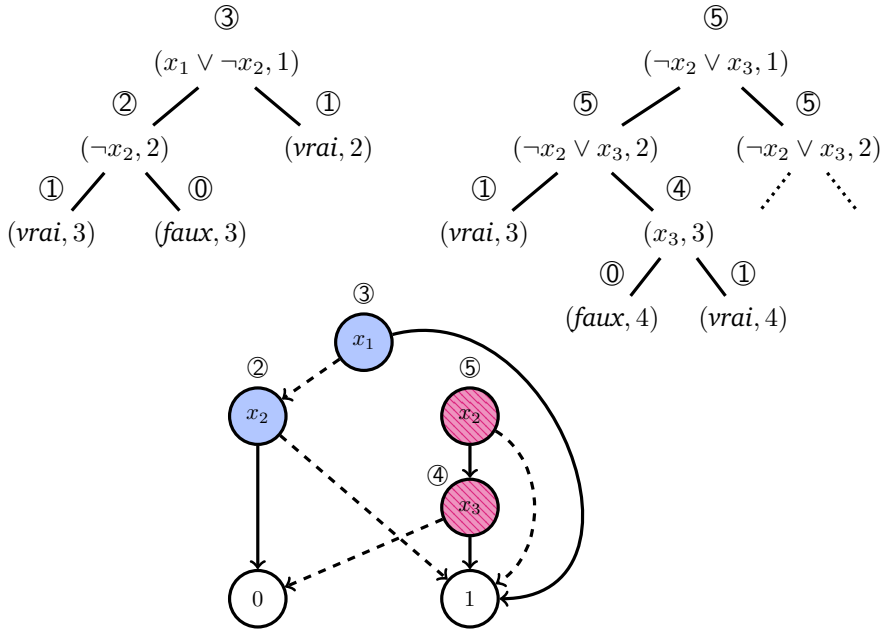


FIGURE 7.4 – *Haut gauche*: arbre d'appel de `build`$(x_1 \vee \neg x_2)$; *haut droite*: arbre d'appel de `build`$(\neg x_2 \vee x_3)$; où $x_1 < x_2 < x_3$, et chaque nombre encerclé indique la valeur retournée par l'appel. *Bas*: BDD résultant des deux appels.

et $hi(u_1) \bigcirc hi(u_2)$. Comme deux chemins ne font pas nécessairement des requêtes simultanément aux mêmes variables, certains appels doivent « mettre un côté en attente » pour que l'autre côté « le ratrappe ».

Par exemple, supposons que nous désirons obtenir un sommet représentant la fonction $f_3 \wedge f_5$, où 3 et 5 sont les sommets du BDD de la figure 7.4. Un appel à `apply`$_\wedge(3, 5)$ génère le nouveau sommet 6, tel que $f_6 = f_3 \wedge f_5$, illustré à la figure 7.5.

7.3.2 Restriction et quantification existentielle

L'algorithme 11 calcule un sommet u' tel que $f_{u'} = f_u[b/x_i]$. Cet algorithme s'avère utile pour quantifier des variables existentiellement. En effet, pour toute expression booléenne t, la formule $\exists x_i \in \{0, 1\} : t(x_1, x_2, \ldots, x_n)$ est équivalente à $t[0/x_i] \vee t[1/x_i]$. Ainsi, comme illustré à l'algorithme 12, deux appels à `restrict` et un appel à `apply` implémentent la quantification existentielle.

Algorithme 10 : Calcul d'une opération binaire $\bigcirc$.

Entrées : sommets u_1 et u_2
Sorties : sommet u qui représente $f_u \coloneqq f_{u_1} \bigcirc f_{u_2}$

$\texttt{apply}_{\bigcirc}(u_1, u_2)$*:*
  $v_1, \ell_1, h_1 \leftarrow \mathit{var}(u_1), \mathit{lo}(u_1), \mathit{hi}(u_1)$
  $v_2, \ell_2, h_2 \leftarrow \mathit{var}(u_2), \mathit{lo}(u_2), \mathit{hi}(u_2)$
  si $u_1 \in \{0,1\}$ *et* $u_2 \in \{0,1\}$ **alors**
    retourner $u_1 \bigcirc u_2$
  sinon si $v_1 < v_2$ **alors** `// Creuser à gauche`
    retourner $\texttt{make}(v_1, \texttt{apply}_{\bigcirc}(\ell_1, u_2), \texttt{apply}_{\bigcirc}(h_1, u_2))$
  sinon si $v_1 > v_2$ **alors** `// Creuser à droite`
    retourner $\texttt{make}(v_2, \texttt{apply}_{\bigcirc}(u_1, \ell_2), \texttt{apply}_{\bigcirc}(u_1, h_2))$
  sinon `// Creuser des deux côtés`
    retourner $\texttt{make}(v_1, \texttt{apply}_{\bigcirc}(\ell_1, \ell_2), \texttt{apply}_{\bigcirc}(h_1, h_2))$

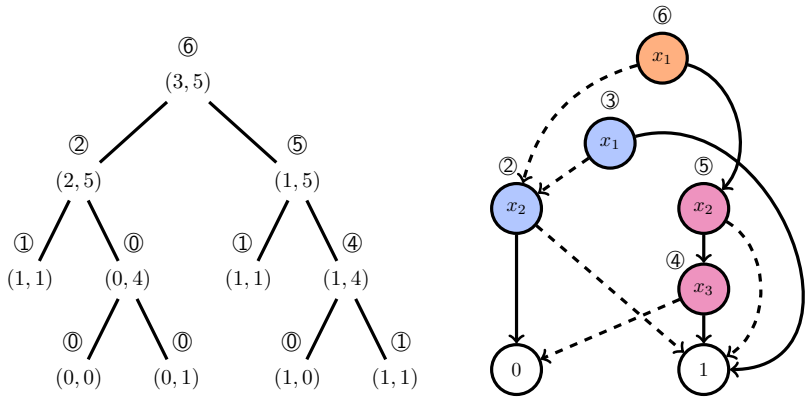


FIGURE 7.5 – *Gauche*: arbre d'appel de $\texttt{apply}_{\wedge}(3, 5)$ sur le BDD de la figure 7.4; *droite*: BDD résultant où $f_6 = f_3 \wedge f_5$.

7.4 Vérification CTL à l'aide de BDD

Les opérations décrites jusqu'ici permettent de représenter et de manipuler des ensembles d'états satisfaisant des propositions atomiques comme $[\![p \wedge q]\!]$. Cependant, les opérateurs temporels quantifiés comme $\exists \mathsf{X}$ nécessitent également de raisonner à propos des transitions d'une structure de Kripke.

Considérons la structure de Kripke $\mathcal{T}$ illustrée à la figure 7.6. Ses états sont représentés par $s_0 = 00$, $s_1 = 01$, $s_2 = 10$ et $s_3 = 11$. La relation de transition $\rightarrow$ peut donc être représentée sur les variables $x_1 < x_2 < x_3 < x_4$ par:

$$\{0001, 0011, 0101, 0110, 1000, 1100\}.$$

Algorithme 11 : Algorithme de restriction d'une variable.

Entrées : un sommet u, un indice $i \in [1..n]$ et une valeur $b \in \{0, 1\}$
Sorties : un sommet u' qui représente $f_{u'} \coloneqq f_u[b/x_i]$

```
restrict(u, i, b):
    restrict'(u):
        v ← var(u)
        si v > x_i alors   // Rien faire, x_i n'apparaît pas sous u
            retourner u
        sinon si v < x_i alors                    // Creuser jusqu'à x_i
            retourner make(v, restrict'(lo(u)), restrict'(hi(u)))
        sinon si b = 0 alors                      // Élaguer branche pleine
            retourner lo(u)
        sinon                                     // Élaguer branche tiretée
            retourner hi(u)
    retourner restrict'(u)
```

Algorithme 12 : Algorithme de quantification existentielle.

Entrées : un sommet u et un indice $i \in [1..n]$
Sorties : un sommet u' qui représente $f_{u'} \coloneqq \exists x_i : f_u$

```
exists(u, i):
    retourner apply_∨(restrict(u, i, 0), restrict(u, i, 1))
```

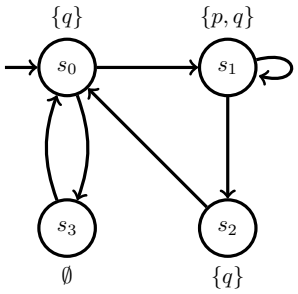


FIGURE 7.6 – Une structure de Kripke.

Par exemple, 0110 dénote la transition $s_1 \to s_2$.

Cherchons à déterminer si $\mathcal{T} \models \exists \mathsf{X} p$. Soit $M \subseteq S$. Définissons $\text{Post}(M) \coloneqq \bigcup_{s \in M} \text{Post}(s)$ et $\text{Pre}(M) \coloneqq \bigcup_{s \in M} \text{Pre}(s)$. Observons que:

$$\text{Post}(M) = \{t : \exists s \ (s, t) \in (\to \cap (M \times S))\},$$
$$\text{Pre}(M) = \{s : \exists t \ (s, t) \in (\to \cap (S \times M))\}.$$

Notons que $[\![\exists \mathsf{X} p]\!] = \text{Pre}([\![p]\!])$. Ainsi, nous devons calculer l'ensemble:

$$\{s : \exists t \ (s,t) \in (\rightarrow \cap \ (S \times [\![p]\!]))\}.$$

La représentation ensembliste de $\rightarrow$ ainsi que l'ensemble $S \times [\![p]\!] = \{0001, 0101, 1001, 1101\}$ sont représentés respectivement par les sommets 8 (en magenta rayé) et 3 (en cyan plein) du BDD de la figure 7.7. Ces sommets s'obtiennent avec deux appels à `build`. L'ensemble $\rightarrow \cap \ (S \times [\![p]\!])$ est représenté par le sommet 9 de la figure 7.7. Ce sommet est obtenu par un appel à $\texttt{apply}_{\wedge}(3, 8)$.

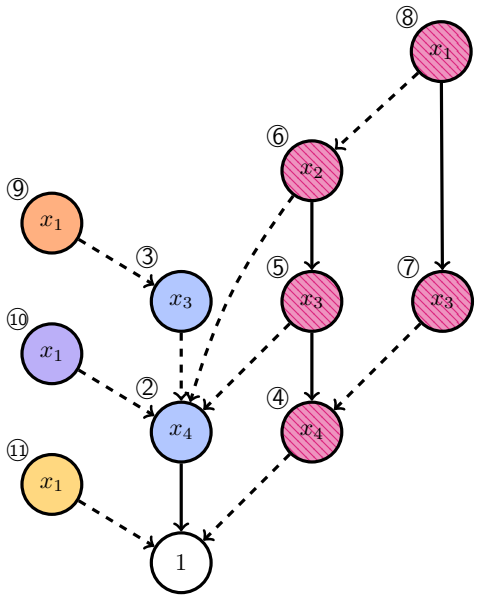


FIGURE 7.7 – BDD partiel obtenu lors de la vérification de $\mathcal{T} \models \exists \mathsf{X} p$. Les arêtes vers le sommet 0 sont omises par souci de lisibilité.

Nous devons maintenant calculer le résultat de la quantification existentielle « $\exists t$ ». Puisque t se situe du côté droit de la paire (s,t), cela correspond à une quantification existentielle de x_3 et x_4. Autrement dit, nous devons calculer $\exists x_4 \ (\exists x_3 \ f_9)$. Ce calcul se fait en deux étapes à l'aide de $\texttt{exists}(\texttt{exists}(9,3),4)$. Ces appels retournent le nouveau sommet 11 illustré à la figure 7.7.

Le sommet 11 représente donc l'ensemble $\text{Pre}([\![p]\!])$ sur les variables x_1 et x_2. Rappelons le critère afin de satisfaire $\exists \mathsf{X} p$:

$$\begin{aligned} \mathcal{T} \models \exists \mathsf{X} p &\iff I \subseteq \text{Pre}[\![p]\!] \\ &\iff I \cap \overline{\text{Pre}[\![p]\!]} = \emptyset. \end{aligned}$$

Afin de compléter la vérification, il demeure donc de:

— calculer un sommet u représentant l'ensemble I;

— calculer un sommet v représentant $\neg f_{11}$;

— calculer un sommet w représentant $f_u \wedge f_v$;

— tester si $f_w \equiv \textit{faux}$.

Par le lemme 1, le test $f_w \equiv \emptyset$ correspond à vérifier si $w = 0$. Le calcul de $\neg f_{11}$ s'effectue quant à lui par un algorithme récursif simple laissé en exercice.

Alternativement, nous pouvons procéder sans implémentation de la négation. En effet, tester une inclusion $A \subseteq B$ correspond à tester si la fonction booléenne $f_A \to f_B$ est une tautotologie, où f_A et f_B sont les fonctions booléennes qui correspondent aux ensembles A et B. L'implication peut être calculée par l'algorithme 10 (`apply`), et le test de tautotologie correspond à vérifier si le sommet obtenu est 1.

7.5 Complexité calculatoire

Pour tout sommet u d'un BDD, nous dénotons par $|u|$ le nombre de sommets accessibles à partir de u. Tel que détaillé dans [And98], toutes les opérations, à l'exception de `build`, s'implémentent de façon polynomiale:

Opération	**Complexité dans le pire cas**
$\texttt{make}(i, \ell, h)$	$\mathcal{O}(1)$
$\texttt{build}(t)$	$\mathcal{O}(2^n)$
$\texttt{apply}_{\bigcirc}(u_1, u_2)$	$\mathcal{O}(\lvert u_1\rvert \cdot \lvert u_2\rvert)$
$\texttt{restrict}(u, i, b)$	$\mathcal{O}(\lvert u\rvert)$
$\texttt{exists}(u, i)$	$\mathcal{O}(\lvert u\rvert^2)$
$f_u \equiv \textit{vrai}$?	$\mathcal{O}(1)$
$f_u \equiv \textit{faux}$?	$\mathcal{O}(1)$

Notons que cette complexité polynomiale nécessite un usage de la mémoïsation, typique à la programmation dynamique, afin d'éviter de recalculer des valeurs déjà calculées (mènerait sinon à des complexités exponentielles). Par exemple, si dans un certain sous-appel récursif à $\texttt{apply}_{\wedge}(u_1, u_2)$ on doit identifier $\texttt{apply}_{\wedge}(a, b)$ et qu'on a déjà calculé sa valeur ailleurs dans la récursion, alors on ne la recalcule pas; on la récupère plutôt d'un tableau associatif.

7.6 Exercices

7.1) Montrez que l'ordre des variables d'une expression booléenne φ peut avoir un impact sur le nombre de sommets du BDD obtenu pour φ.

7.2) Montrez maintenant que l'ordre des variables peut avoir un impact *exponentiel* sur le nombre de sommets.

7.3) Construisez un sommet de BDD qui représente...

a) $f \coloneqq (x_1 \wedge x_3) \vee x_2$;
b) $g \coloneqq x_1 \oplus x_2$;
c) $f \wedge g$;
d) $\exists x_3 : f$.

7.4) Expliquez comment implémenter ces opérations pour des sommets u et v:

a) tester si $f_u \leftrightarrow f_v$;
b) calculer $\{x \in \{0,1\}^n : f_u(x_1, \ldots, x_n) = \textit{vrai}\}$;
c) calculer $x \in \{0,1\}^n$ tel que $f_u(x_1, \ldots, x_n) = \textit{vrai}$;
d) calculer $|\{x \in \{0,1\}^n : f_u(x_1, \ldots, x_n) = \textit{vrai}\}|$.

7.5) Expliquez comment gérer d'autres quantificateurs et opérateurs temporels, par ex. $\forall$X et $\exists$G, à l'aide de BDD.

7.6) ★★ (*Requiert des connaissances en théorie du calcul*) Si la complexité de `build` était polynomiale (pas le cas), alors on aurait P = NP. Pourquoi?

8

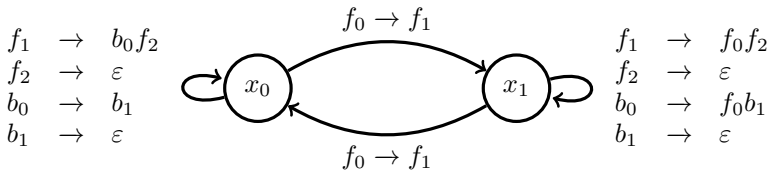
Systèmes avec récursion

Dans ce chapitre, nous voyons comment vérifier des systèmes qui font des appels récursifs ou possèdent une forme de pile. Il n'est pas possible de modéliser de tels systèmes par une structure de Kripke finie puisqu'une pile peut généralement avoir une taille arbitraire. Néanmoins, il s'avère possible de les représenter symboliquement à l'aide d'une variante des automates à pile.

Par exemple, considérons le programme suivant constitué de deux fonctions et d'une variable booléenne globale x:

```
        bool x ∈ {faux,vrai}

        foo():                          bar():
f0:       x = ¬x:                 b0:     si x: foo()
f1:       si x: foo()             b1:     retourner
          sinon: bar()
f2:       retourner
```

Il n'est pas immédiatement clair si la pile d'appel peut devenir infinie ou non à partir d'un appel initial à `foo`. Ainsi, sans en avoir la certitude, nous ne pouvons pas utiliser les méthodes de vérification présentées jusqu'ici. Le programme peut plutôt être modélisé à l'aide de ce diagramme:



Les états x_0 et x_1 indiquent que x $=$ *faux* et x $=$ *vrai* respectivement. Les transitions représentent le flot du programme. Par exemple, la transition « $f_1 \to b_0 f_2$ » indique qu'à partir de la ligne étiquetée par f_1, on se déplace à celle étiquetée

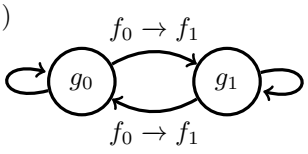
par b_0, et on effectue un retour vers f_2 (plus tard). Le mot vide ε indique que l'exécution locale se complète et que rien n'est mis sur la pile d'appel, par ex. « $b_1 \to \varepsilon$ » représente un retour de la fonction `bar`.

Ce type de modélisation s'étend également aux variables locales. Par exemple, considérons ce programme:

```
        bool g ∈ {faux,vrai}

        main(l):                    foo():
m0:       si l: foo()       f0:     g = ¬g
m1:       assert(g == l)    f1:     si ¬g:
m2:       retourner                   foo()
                            f2:       foo()
                            f3:     retourner
```

Lorsque le programme se situe à une instruction de la fonction `main`, l'état du programme est entièrement déterminé par l'étiquette de l'instruction courante ainsi que la valeur du paramètre `l`; autrement dit, par une paire de la forme (m_i, ℓ_j). De plus, nous pouvons introduire un symbole « $\bot$ » afin d'indiquer que l'assertion est enfreinte. Nous obtenons ainsi cette modélisation, où chaque paire (m_i, ℓ_j) est vue comme un unique symbole:

$$
\begin{array}{lcl}
(m_0,\ell_0) & \to & (m_1,\ell_0) \\
(m_0,\ell_1) & \to & f_0(m_1,\ell_1) \\
(m_1,\ell_0) & \to & (m_2,\ell_0) \\
(m_1,\ell_1) & \to & \bot \\
(m_2,\ell_0) & \to & \varepsilon \\
(m_2,\ell_1) & \to & \varepsilon \\
f_1 & \to & f_0 f_2 \\
f_2 & \to & f_0 f_3 \\
f_3 & \to & \varepsilon
\end{array}
$$



$$
\begin{array}{lcl}
(m_0,\ell_0) & \to & (m_1,\ell_0) \\
(m_0,\ell_1) & \to & f_0(m_1,\ell_1) \\
(m_1,\ell_0) & \to & \bot \\
(m_1,\ell_1) & \to & (m_2,\ell_1) \\
(m_2,\ell_0) & \to & \varepsilon \\
(m_2,\ell_1) & \to & \varepsilon \\
f_1 & \to & f_3 \\
f_2 & \to & f_0 f_3 \\
f_3 & \to & \varepsilon
\end{array}
$$

8.1 Systèmes à pile

Il est naturel de chercher à vérifier des propriétés de systèmes tels que ceux présentés ci-dessus. Par exemple: est-ce que toutes les exécutions du deuxième programme satisfont l'assertion à la ligne m_1? Afin d'effectuer une telle vérification, nous introduisons les systèmes à pile; un formalisme qui représente les modélisations que nous avons données pour les programmes ci-dessus.

Définition 2. Un *système à pile* est un triplet $\mathcal{P} = (P, \Gamma, \Delta)$ où

— P est un ensemble fini dont les éléments sont appelés *états*;

— Γ est l'*alphabet* fini de la pile;

— $\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$ est un ensemble fini de *transitions*.

Une *configuration* de $\mathcal{P}$ est une paire $\langle p, w\rangle \in P \times \Gamma^*$ où w décrit le contenu d'une pile. Par exemple, $\langle p, abc\rangle$ représente la configuration où $\mathcal{P}$ est dans l'état p et le contenu de sa pile est:

a
b
c

Une transition $((p, a), (q, u)) \in \Delta$, que nous dénotons $p \xrightarrow{a \to u} q$, indique que dans l'état p, si la lettre a est au sommet de la pile, alors on peut dépiler a, se déplacer à l'état q, et empiler u (qui peut être le mot vide). Dans ce cas, nous écrivons $\langle p, av\rangle \to \langle q, uv\rangle$, où v décrit le contenu du reste de la pile. Plus généralement, nous écrivons

$$\langle p, w\rangle \xrightarrow{i} \langle p', w'\rangle$$

s'il est possible de passer de la configuration $\langle p, w\rangle$ à la configuration $\langle p', w'\rangle$ en i transitions.

Soit C un ensemble de configurations. Nous définissons l'ensemble des *prédecesseurs* et des *successeurs* de C respectivement par

$$\text{Pre}^*(C) := \bigcup_{i \geq 0} \text{Pre}^i(C), \qquad \text{Post}^*(C) := \bigcup_{i \geq 0} \text{Post}^i(C),$$

où

$$\text{Pre}^i(C) := \{\langle p, w\rangle : \text{il existe } \langle p', w'\rangle \in C \text{ telle que } \langle p, w\rangle \xrightarrow{i} \langle p', w'\rangle\},$$
$$\text{Post}^i(C) := \{\langle p', w'\rangle : \text{il existe } \langle p, w\rangle \in C \text{ telle que } \langle p, w\rangle \xrightarrow{i} \langle p', w'\rangle\}.$$

8.2 Calcul des prédécesseurs

L'ensemble $\text{Pre}^*(C)$ permet de déterminer certaines propriétés d'un système à pile. Par exemple, posons $C := \{\langle g_i, \bot w\rangle : i \in \{0, 1\}, w \in \Gamma^*\}$. L'ensemble C représente les configurations qui enfreignent l'assertion du programme précédent. Ainsi, à partir du point d'entrée `main`, le programme contient un bogue ssi $\text{Pre}^*(C) \cap \{\langle g_i, (m_0, \ell_j)\rangle : i, j \in \{0, 1\}\} \neq \emptyset$. Nous décrivons une procédure qui calcule $\text{Pre}^*(C)$. Puisque C et $\text{Pre}^*(C)$ peuvent être infinis, nous devons représenter ces ensembles *symboliquement*. Pour ce faire, nous utilisons les $\mathcal{P}$-automates définis comme suit:

Définition 3. Soit $\mathcal{P} = (P, \Gamma, \Delta)$ un système à pile. Un $\mathcal{P}$-automate $\mathcal{A}$ est un automate fini tel que $\mathcal{A} = (Q, \Gamma, \delta, P, F)$, où les composantes sont respectivement les *états*, *alphabet*, *transitions*, *états initiaux* et *états finaux* de $\mathcal{A}$.

Autrement dit, un $\mathcal{P}$-automate $\mathcal{A}$ est un automate fini classique dont les états initiaux sont ceux de $\mathcal{P}$ et dont l'alphabet est celui de la pile de $\mathcal{P}$. Le langage accepté par $\mathcal{A}$ représente le contenu possible de la pile de $\mathcal{P}$. Afin d'éviter toute ambiguïté, nous ajoutons parfois un indice à « $\rightarrow$ » afin de mettre l'emphase sur le système dont nous faisons référence, par ex. $\rightarrow_{\mathcal{A}}$ pour $\mathcal{A}$.

Formellement, nous disons que le $\mathcal{P}$-automate $\mathcal{A}$ *accepte* une configuration $\langle p, w\rangle \in P \times \Gamma^*$ ssi $p \in P$ et il existe $q \in F$ tel que

$$p \xrightarrow{w}_{\mathcal{A}} q.$$

L'ensemble des configurations acceptées par $\mathcal{A}$ est dénoté par *Conf*$(\mathcal{A})$. Par exemple, le haut de la figure 8.1 illustre un système à pile $\mathcal{P}$ (gauche) et un $\mathcal{P}$-automate $\mathcal{A}$ (droite) où *Conf*$(\mathcal{A}) = \{\langle p, aa\rangle\}$.

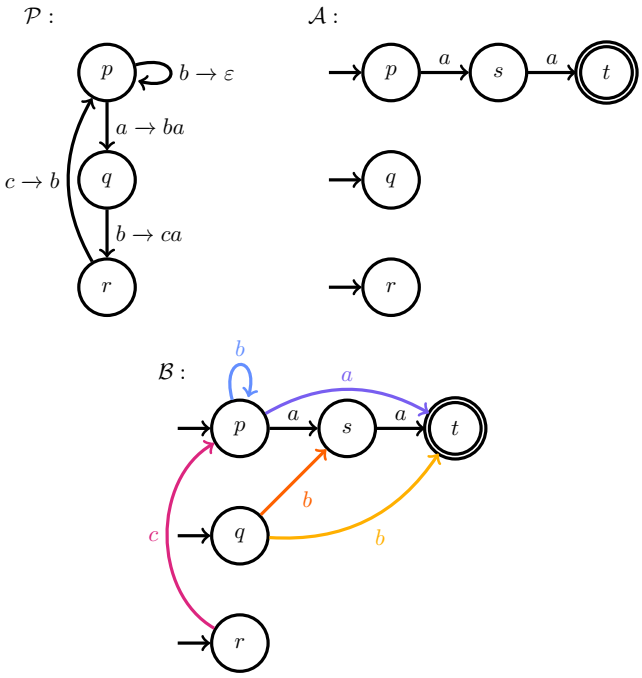


FIGURE 8.1 – Exemple de $\mathcal{P}$-automate $\mathcal{B}$ tel que *Conf*$(\mathcal{B})$ = Pre$^*(\mathcal{A})$ calculé à partir d'un système à pile $\mathcal{P}$ et d'un $\mathcal{P}$-automate $\mathcal{A}$.

Une représentation des prédécesseurs de *Conf*$(\mathcal{A})$ se calcule algorithmiquement:

Théorème 2 ([EHRS00, BEM97]). *Soient $\mathcal{P}$ un système à pile et $\mathcal{A}$ un $\mathcal{P}$-automate. Il est possible de construire, en temps polynomial, un $\mathcal{P}$-automate $\mathcal{B}$ tel que Conf$(\mathcal{B})$ =* Pre*(*Conf*$(\mathcal{A})$).

Le $\mathcal{P}$-automate $\mathcal{B}$ de la proposition 2 se calcule comme suit. Par souci technique, nous supposons, sans perte de généralité, qu'aucune transition n'entre

dans un état initial de $\mathcal{A}$. Nous initialisons $\mathcal{B} \coloneqq \mathcal{A}$, puis nous enrichissons $\mathcal{B}$ en lui ajoutant des transitiitions avec cette règle:

> Si $p \xrightarrow{a \to w}_{\mathcal{P}} p'$ et $p' \xrightarrow{w}_{\mathcal{B}} q$, alors (p, a, q) peut être ajoutée à $\mathcal{B}$.

Expliquons le raisonnement derrière cette règle. Soit $y \in \Gamma^*$ un mot qui mène de q vers un état final q_f. Nous avons

$$p' \xrightarrow{w}_{\mathcal{B}} q \xrightarrow{y}_{\mathcal{B}} q_f$$

et ainsi $\langle p', wy \rangle \in \mathit{Conf}(\mathcal{B})$. Puisque $p \xrightarrow{a \to w}_{\mathcal{P}} p'$, nous avons $\langle p, ay \rangle \to \langle p', wy \rangle$ et ainsi $\langle p, ay \rangle \in \mathrm{Pre}(\mathit{Conf}(\mathcal{B}))$. Comme nous cherchons à obtenir *tous* les prédécesseurs, nous désirons enrichir $\mathcal{B}$ en ajoutant $\langle p, ay \rangle$ à $\mathit{Conf}(\mathcal{B})$. L'ajout de la transition (p, a, q) accomplit exactement cette tâche (pour tous les y possibles).

Ainsi, la règle est appliquée jusqu'à saturation de $\mathcal{B}$, c.-à-d. jusqu'à ce que la règle ne soit plus applicable et qu'on obtienne tous les prédécesseurs. Une telle procédure est décrite sous forme de pseudocode à l'algorithme 13.

Algorithme 13 : Calcul de prédécesseurs.

Entrées : Un système à pile $\mathcal{P} = (P, \Gamma, \Delta)$ et un $\mathcal{P}$-automate $\mathcal{A} = (Q, \Gamma, \delta, P, F)$

Sorties : Un $\mathcal{P}$-automate $\mathcal{B}$ tel que $\mathit{Conf}(\mathcal{B}) = \mathrm{Pre}^*(\mathit{Conf}(\mathcal{A}))$

```
B ← A
changement ← vrai
tant que changement
    changement ← faux
    pour ((p, a), (p', w)) ∈ Δ
        pour q ∈ Q
            si p' -w-> q dans B alors
                ajouter (p, a, q) à B
                changement ← vrai
retourner B
```

Exemple.

Appliquons l'algorithme 13 au système à pile $\mathcal{P}$ et au $\mathcal{P}$-automate $\mathcal{A}$ illustrés au haut de la figure 8.1. Cinq transitions sont ajoutées à $\mathcal{A}$:

	Règle	Nouvelle transition
①	$p \xrightarrow{b\to\varepsilon} p$ $p \xrightarrow{\varepsilon} p$	(p, b, p)
②	$r \xrightarrow{c\to b} p$ $p \xrightarrow{b} p$	(r, c, p)
③	$q \xrightarrow{b\to ca} r$ $r \xrightarrow{ca} s$	(q, b, s)
④	$p \xrightarrow{a\to ba} q$ $q \xrightarrow{ba} t$	(p, a, t)
⑤	$b \xrightarrow{b\to ca} r$ $r \xrightarrow{ca} t$	(q, b, t)

Nous donnons un aperçu du bon fonctionnement de l'algorithme, les détails techniques sont décrits en annexe à l'exercice 8.8).

Esquisse de preuve du théorème 2. Soit $\mathcal{B}_0 := \mathcal{A}$ et soit $\mathcal{B}_i$ le $\mathcal{P}$-automate obtenu à partir de $\mathcal{B}_{i-1}$ après avoir ajouté tout triplet (p, a, q) permis par la règle. Par définition, nous avons $\mathit{Conf}(\mathcal{B}_{i+1}) \supseteq \mathit{Conf}(\mathcal{B}_i)$ pour tout $i \in \mathbb{N}$. Comme nous ajoutons des transitions, mais jamais d'état, nous avons forcément

$$\mathit{Conf}(\mathcal{B}_k) = \mathit{Conf}(\mathcal{B}_{k+1}) = \mathit{Conf}(\mathcal{B}_{k+2}) = \cdots \tag{8.1}$$

où $k := |P|^2 \cdot |\Gamma|$, c.-à-d. le nombre maximal de transitions.

De plus, par définition de la règle, il est possible de montrer que

$$\mathrm{Pre}^*(\mathit{Conf}(\mathcal{A})) \supseteq \mathit{Conf}(\mathcal{B}_i) \supseteq \mathrm{Pre}^i(\mathit{Conf}(\mathcal{A})) \text{ pour tout } i \in \mathbb{N}. \tag{8.2}$$

En particulier, $\mathrm{Pre}^*(\mathit{Conf}(\mathcal{A})) \supseteq \mathit{Conf}(B_k)$. De plus,

$$\begin{aligned}
\mathrm{Pre}^*(\mathit{Conf}(\mathcal{A})) &= \bigcup_{i=0}^{\infty} \mathrm{Pre}^i(\mathit{Conf}(\mathcal{A})) && \text{(par déf. de Pre}^*\text{)} \\
&\subseteq \bigcup_{i=0}^{\infty} \mathit{Conf}(\mathcal{B}_i) && \text{(par (8.2))} \\
&= \mathit{Conf}(\mathcal{B}_k) && \text{(par (8.1))}.
\end{aligned}$$

Nous concluons donc que $\mathit{Conf}(\mathcal{B}_k) = \mathrm{Pre}^*(\mathit{Conf}(\mathcal{A}))$. □

8.3 Vérification à l'aide de système à pile

Afin d'illustrer la vérification d'un programme récursif dont le domaine des variables est fini, considérons ce programme:

```
        bool x, y ∈ {faux,vrai}

        main():                          foo():
m0:       y = ¬y:                f0:       x = ¬y
m1:       si y:                  f1:       si x: foo()
            foo()
          sinon:
            main()
m2:       assert(x ≠ y)
```

Celui-ci se modélise par le système à pile $\mathcal{P}$ illustré à la figure 8.2.

Afin de déterminer si l'assertion n'est jamais enfreinte, il suffit de:

— construire un $\mathcal{P}$-automate $\mathcal{A}$ qui accepte l'ensemble des configurations où les variables prennent une valeur arbitraire et le dessus de la pile correspond à $\perp$ (voir la figure 8.3);

— calculer un $\mathcal{P}$-automate $\mathcal{B}$ qui accepte $\text{Pre}^*(\mathit{Conf}(\mathcal{A}))$ à l'aide de l'algorithme 13;

— vérifier s'il existe une configuration initiale, c.-à-d. de la forme $\langle (x_i, y_i), m_0 \rangle$, acceptée par $\mathcal{B}$, auquel cas il y a un bogue (et sinon aucun).

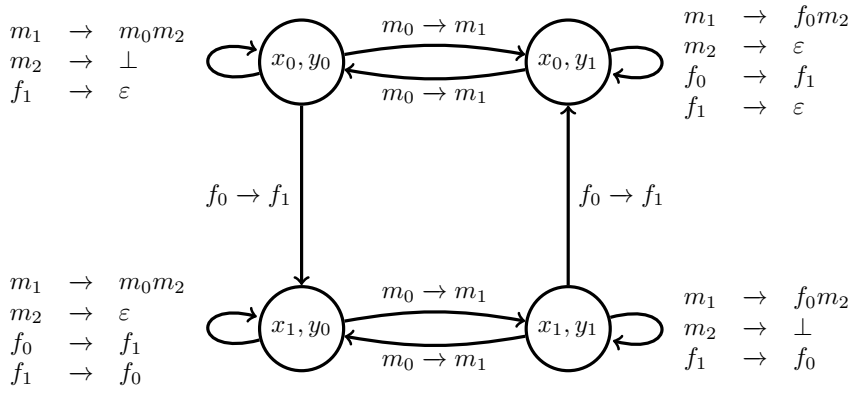


FIGURE 8.2 – Système à pile qui modélise un programme récursif.

Mentionnons qu'il existe également un algorithme un peu plus complexe qui calcule une représentation symbolique des *successeurs* des configurations d'un $\mathcal{P}$-automate. Celui-ci enrichit également un $\mathcal{P}$-automate $\mathcal{B}$ initialisé à $\mathcal{A}$ mais en lui ajoutant possiblement de nouveaux états, ce qui complique son analyse.

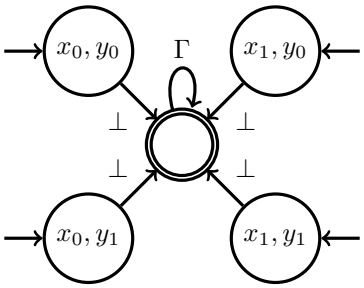


FIGURE 8.3 – Un $\mathcal{P}$-automate $\mathcal{A}$ qui représente les configurations de $\mathcal{P}$ où l'assertion est enfreinte, c.-à-d. $\{\langle (x_i, y_j), \bot w\rangle : i, j \in \{0, 1\}, w \in \Gamma^*\}$.

Théorème 3 ([EHRS00]). *Soient $\mathcal{P}$ un système à pile et $\mathcal{A}$ un $\mathcal{P}$-automate. Il est possible de construire, en temps polynomial, un $\mathcal{P}$-automate $\mathcal{B}$ tel que Conf$(\mathcal{B}) =$ Post*(Conf$(\mathcal{A})$).*

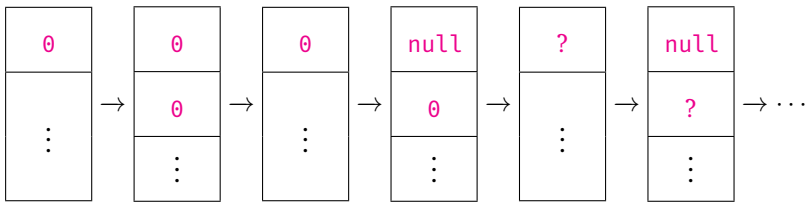
De plus, il existe un algorithme de complexité polynomiale [EHRS00] qui vérifie si un système à pile satisfait une formule LTL dont la négation est spécifiée par un automate de Büchi et dont les propositions atomiques raisonnent sur les lettres au dessus de la pile. La vérification CTL est également possible, mais en temps exponentiel dans le pire cas.

8.4 Autre exemple: analyse de code intermédiaire

Afin d'illustrer une application supplémentaire, utilisons les systèmes à pile afin de raisonner sur le typage d'un programme dérit sous forme de code intermédiaire (« *bytecode* ») Java comme celui-ci (tiré de [Fé19, p. 65]):

```
sconst_0        // mettre 0 de type « short » sur la pile
sconst_0        // mettre 0 de type « short » sur la pile
                /* dépiler deux nombres de type « short »,
                   les multiplier et mettre le résultat
smul               de type « short » sur la pile        */
aconst_null     // empiler la référence nulle
goto -2         // brancher deux instructions en arrière
```

Si l'on ignore les erreurs de typage, la pile de ce programme évoluerait de cette façon, où « ? » indique une valeur non définie:



À la seconde exécution de l'instruction `smul`, le programme tente de multiplier l'entier 0 par la référence nulle, ce qui ne devrait pas être permis.

Nous pourrions modéliser ce programme à l'aide d'un système à pile, mais la représentation des entiers de type « short » (16 bits) ajouterait 2^{16} lettres à l'alphabet Γ. De façon générale, il faudrait aussi représenter la valeur des registres (pas utilisés dans ce programme) dans les états du système à pile.

Ce genre d'erreur de typage se détecte à l'aide d'une modélisation moins fine où les valeurs correspondent aux types. Posons $\Gamma \coloneqq \{\mathtt{s}, \mathtt{a}, \ldots\}$ où $\mathtt{s}$ représente le type « short » et $\mathtt{a}$ représente le type « reference ». Le typage du programme précédent se modélise par le système à pile $\mathcal{P}$ de la figure 8.4.

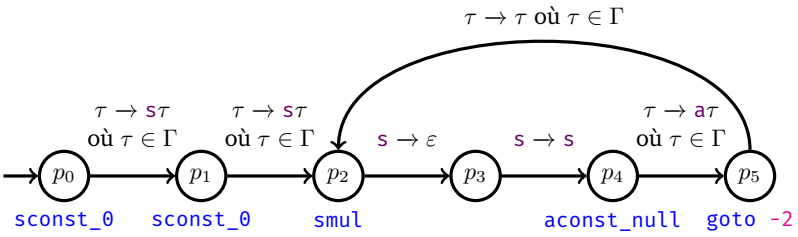


FIGURE 8.4 – Modélisation du typage d'un programme en code intermédiaire.

L'instruction `smul` cause une erreur (au sens du typage) si l'on peut atteindre une configuration de cet ensemble:

$$\mathsf{Err} \coloneqq \{\langle p_i, \tau w\rangle : i \in \{2,3\}, \tau \in \Gamma \setminus \{\mathtt{s}\}, w \in \Gamma^*\} \cup \{\langle p_i, \varepsilon\rangle : i \in \{2,3\}\}.$$

Le $\mathcal{P}$-automate $\mathcal{A}$ illustré à la figure 8.5 accepte l'ensemble Err. Nous pourrions donc utiliser l'algorithme 13 pour effectuer la vérification automatiquement.

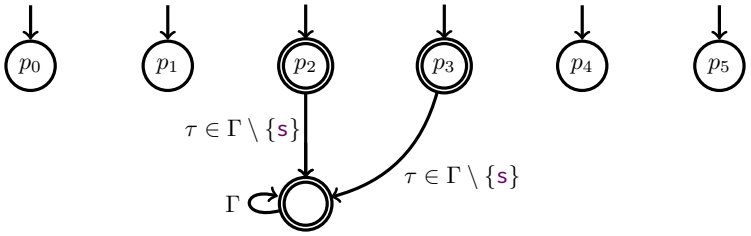


FIGURE 8.5 – Un $\mathcal{P}$-automate $\mathcal{A}$ tel que $\mathit{Conf}(\mathcal{A}) = \mathsf{Err}$.

Remarque.

La définition de système à pile force une transition à dépiler. Les algorithmes s'adaptent pour gérer le dépilement d'aucune lettre (mot vide) ou de plusieurs lettres. Cela simplifierait la modélisation de la figure 8.4.

8.5 Exercices

8.1) Reconsidérons ce programme non déterministe du chapitre 1:

```
  main():          foo():           bar():
m0: foo()      f0: si ?: bar()  b0: foo()
m1: retourner  f1: retourner    b1: si ?: retourner
                                b2: bar()
```

a) Modélisez le programme à l'aide d'un système à pile.

b) Expliquez comment nous pourrions déterminer si le programme *peut* terminer à partir du point d'entrée `main`.

8.2) Soit $\mathcal{A}$ un $\mathcal{P}$-automate. Comment peut-on déterminer si $\text{Pre}^*(\mathit{Conf}(\mathcal{A}))$ (ou $\text{Post}^*(\mathit{Conf}(\mathcal{A}))$) est infini?

8.3) Nous pouvons associer une structure de Kripke *infinie* à un système à pile: les états sont les configurations et les transitions dictent l'évolution entre les configurations. Considérons une variante où la fonction d'étiquetage L est de la forme $L\colon (P \times \Gamma) \to 2^{AP}$, plutôt que de $(P \times \Gamma^*) \to 2^{AP}$. En mots, cela signifie qu'une proposition atomique attachée à une configuration $\langle q, aw\rangle$ ne peut raisonner que sur son état q et la lettre a au dessus de sa pile. Donnez une formule LTL, et la façon dont vous assignez les propositions atomiques, pour ces propriétés du programme de la section 8.3:

a) une trace exécute `foo` infiniment souvent;

b) le programme termine;

c) chaque fois que `foo` est exécutée, `main` l'est éventuellement aussi;

d) à partir d'un certain point, $\mathtt{x} = \mathtt{y}$.

8.4) Expliquez pourquoi nous pouvons supposer qu'aucune transition ne mène vers un état initial d'un $\mathcal{P}$-automate.

8.5) Calculez $\text{Pre}^*(\mathit{Conf}(\mathcal{A}))$ avec l'algorithme 13 où $\mathcal{A}$ est le $\mathcal{P}$-automate de la figure 8.5 par rapport au système à pile $\mathcal{P}$ de la figure 8.4. Supposez que $\Gamma = \{\mathtt{a}, \mathtt{i}, \mathtt{s}\}$.

8.6) Notre modélisation à la section 8.4 suppose que seuls deux entiers de type « short » (16 bits) peuvent être additionnés par l'instruction `smul`. Supposons qu'elle supporte aussi des entiers de type « byte » (8 bits) qui sont convertis à l'interne sur 16 bits. Adaptez $\mathcal{P}$ et $\mathcal{A}$ pour en tenir compte.

8.7) Donnez un système à pile pour ces programmes (tirés de [Fé19, p. 65]):

— `sconst_0; goto 3; goto -2; sconst_0; smul`

— `aconst_null; pop; goto -2`

8.8) ★★ (*Cet « exercice » est ardu et sert d'abord à présenter une preuve, omise dans le chapitre, aux lectrices et lecteurs intéressé·e·s; tentez d'abord l'inclusion de droite qui est la plus simple*)

Complétez cette partie de l'esquisse de preuve du théorème 2:

$$\mathrm{Pre}^*(\mathcal{A}) \supseteq \mathit{Conf}(\mathcal{B}_i) \supseteq \mathrm{Pre}^i(\mathit{Conf}(\mathcal{A})) \text{ pour tout } i \in \mathbb{N}.$$

9

Systèmes infinis

À l'exception des systèmes à pile, nous avons considéré des systèmes finis. En général, par le théorème de Rice, il est impossible de vérifier le bon fonctionnement d'un système infini arbitraire. En particulier, il n'existe aucun algorithme qui résout le problème d'arrêt qui consiste à déterminer si un programme termine sur une entrée donnée. Néanmoins, il existe des sous-classes de systèmes infinis qui sont vérifiables algorithmiquement. Nous faisons une excursion dans ce domaine en considérant les réseaux de Petri, un formalisme qui peut être vu comme une extension des graphes (ou des automates) avec une forme de compteurs. Les réseaux de Petri permettent de modéliser des systèmes variés allant des programmes concurrents aux systèmes biologiques, chimiques ou d'affaires.

9.1 Réseaux de Petri

Définition 4. Un *réseau de Petri* est un triplet $\mathcal{N} = (P, T, F)$ où

- P est un ensemble fini (*places*);
- T est un ensemble fini disjoint de P (*transitions*);
- $F \colon ((P \times T) \cup (T \times P)) \to \mathbb{N}$ (*fonction de flot*).

Exemple.

Le réseau de Petri suivant est illustré à la figure 9.1:

$$P = \{p, q\}, \qquad T = \{s, t\},$$

$$\begin{aligned} F(p,s) &= 1, & F(p,t) &= 0,\\ F(q,s) &= 0, & F(q,t) &= 3,\\ F(s,p) &= 0, & F(t,p) &= 4,\\ F(s,q) &= 2, & F(t,q) &= 1. \end{aligned}$$

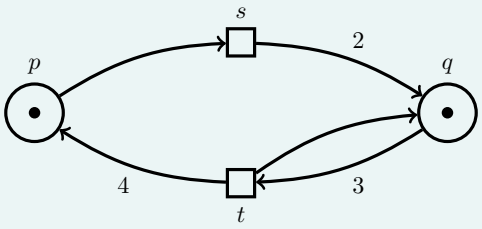


FIGURE 9.1 – Un exemple de réseau de Petri marqué par $(1, 1)$. Les places et les transitions sont représentées respectivement par des cercles et des carrés. Les arcs représentent la fonction de flot.

Un *marquage* est un vecteur $\boldsymbol{m} \in \mathbb{N}^P$ qui associe un nombre de jetons $\boldsymbol{m}(p)$ à chaque place p. Une transition $t \in T$ est *déclenchable* dans $\boldsymbol{m}$ si $\boldsymbol{m}(p) \geq F(p, t)$ pour toute place $p \in P$. Si t est déclenchable, alors

$$\boldsymbol{m} \xrightarrow{t} \boldsymbol{m}' \text{ où } \boldsymbol{m}'(p) := \boldsymbol{m}(p) - F(p, t) + F(t, p) \text{ pour tout } p \in P.$$

Nous écrivons $\boldsymbol{m} \to \boldsymbol{m}'$ s'il existe $t \in T$ telle que $\boldsymbol{m} \xrightarrow{t} \boldsymbol{m}'$, et $\boldsymbol{m} \xrightarrow{*} \boldsymbol{m}'$ si $\boldsymbol{m} = \boldsymbol{m}'$ ou s'il existe une séquence de marquages et de transitions tels que:

$$\boldsymbol{m} = \boldsymbol{m}_0 \xrightarrow{t_1} \boldsymbol{m}_1 \xrightarrow{t_2} \cdots \xrightarrow{t_k} \boldsymbol{m}_k = \boldsymbol{m}'.$$

Exemple.

Dans le réseau de la figure 9.1, nous avons $(1, 1) \xrightarrow{*} (3, 3)$ puisque

$$(1, 1) \xrightarrow{s} (0, 3) \xrightarrow{t} (4, 1) \xrightarrow{s} (3, 3).$$

L'ensemble des *successeurs* et *prédécesseurs* d'un marquage $\boldsymbol{m}$ sont respectivement définis par:

$$\text{Post}^*(\boldsymbol{m}) = \{\boldsymbol{m}' \in \mathbb{N}^P : \boldsymbol{m} \xrightarrow{*} \boldsymbol{m}'\},$$
$$\text{Pre}^*(\boldsymbol{m}) = \{\boldsymbol{m}' \in \mathbb{N}^P : \boldsymbol{m}' \xrightarrow{*} \boldsymbol{m}\}.$$

Nous écrivons $\boldsymbol{m} \geq \boldsymbol{m}'$ si $\boldsymbol{m}(p) \geq \boldsymbol{m}'(p)$ pour toute place $p \in P$. Par exemple, $(1, 2, 3) \geq (1, 1, 0)$, mais $(1, 2, 3) \not\geq (0, 1, 4)$.

9.2 Modélisation de systèmes concurrents

Considérons le programme suivant, constitué d'une variable booléenne globale x, qui peut exécuter un nombre arbitraire de processus:

```
      x = faux

      tant que ?:
        lancer proc()

      proc():
p0:     si ¬x: x = vrai  sinon: goto p0
p1:     tant que ¬x: pass
p2:     // code
```

Ce programme ne peut pas être modélisé à l'aide d'une structure de Kripke finie puisque le nombre de processus n'est pas borné. Nous pouvons cependant le modéliser à l'aide d'un réseau de Petri tel qu'illustré à la figure 9.2.

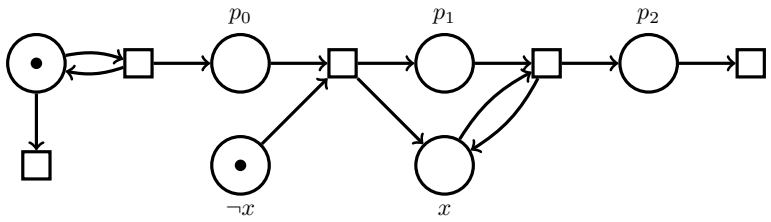


FIGURE 9.2 – Réseau de Petri qui modélise l'exécution du processus `proc`. La place tout à gauche permet de lancer un nombre arbitraire de processus et de décider de cesser d'en lancer.

Supposons qu'on cherche à déterminer si plusieurs processus peuvent atteindre la ligne p_2. Cela est équivalent à déterminer s'il existe un marquage $\boldsymbol{m}'$ tel que:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ & 1 & 0 & \end{pmatrix} \xrightarrow{*} \boldsymbol{m}' \text{ et } \boldsymbol{m}' \geq \begin{pmatrix} 0 & 0 & 0 & 2 \\ & 0 & 0 & \end{pmatrix}.$$

Considérons un programme similaire, aussi constitué d'une variable booléenne globale x, et qui peut aussi exécuter un nombre arbitraire de processus:

```
      x = faux

      tant que ?:
        lancer proc2()

      proc2():
p0:     si ¬x: x = vrai  sinon: goto p0
p1:     tant que ¬x: pass
p2:     x = ¬x
```

Ce programme est modélisé par un réseau de Petri à la figure 9.3. Supposons qu'on cherche à déterminer si le programme peut se terminer avec $x =$ `vrai`.

Cela est équivalent à déterminer si:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ & 1 & 0 & \end{pmatrix} \xrightarrow{*} \begin{pmatrix} 0 & 0 & 0 & 0 \\ & 0 & 1 & \end{pmatrix}.$$

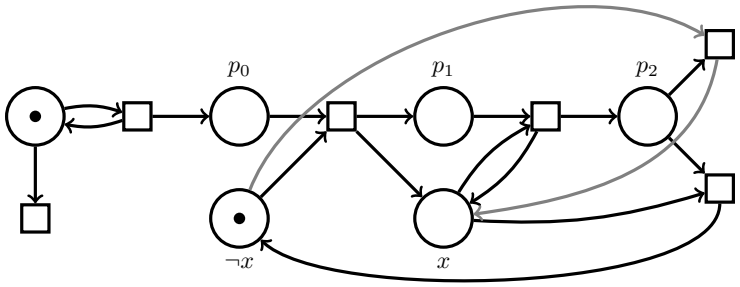


FIGURE 9.3 – Réseau de Petri quie modélise l'exécution du processus `proc2`.

9.3 Vérification

Nous formalisons les problèmes de vérification sous-jacents aux deux exemples:

PROBLÈME D'ACCESSIBILITÉ

ENTRÉE: réseau de Petri $\mathcal{N} = (P, T, F)$ et marquages $\boldsymbol{m}, \boldsymbol{m}' \in \mathbb{N}^P$

QUESTION: $\boldsymbol{m} \xrightarrow{*} \boldsymbol{m}'$?

PROBLÈME DE COUVERTURE

ENTRÉE: réseau de Petri $\mathcal{N} = (P, T, F)$ et marquages $\boldsymbol{m}, \boldsymbol{m}' \in \mathbb{N}^P$

QUESTION: existe-t-il $\boldsymbol{m}'' \in \mathbb{N}^P$ tel que $\boldsymbol{m} \xrightarrow{*} \boldsymbol{m}''$ et $\boldsymbol{m}'' \geq \boldsymbol{m}'$?

Ces deux problèmes sont décidables; autrement dit, ils sont tous deux solubles à l'aide d'un algorithme.

Remarque.

Bien que ces deux problèmes soient décidables, leur complexité théorique est colossale: non élémentaire [ST77, May81, Kos82, Lam92, Ler12, CLL+19] et EXPSPACE-complet [Lip76, Rac78], respectivement. Cela n'empêche toutefois pas de les résoudre en pratique.

Nous mettons l'emphase sur le problème de couverture et présentons deux algorithmes permettant de résoudre ce problème.

Nous disons que $\boldsymbol{m}'$ est *couvrable* à partir de $\boldsymbol{m}$ s'il existe un marquage $\boldsymbol{m}'' \in \mathbb{N}^P$ tel que $\boldsymbol{m} \xrightarrow{*} \boldsymbol{m}''$ et $\boldsymbol{m}'' \geq \boldsymbol{m}'$. Nous disons que $\boldsymbol{m}$ peut *couvrir* $\boldsymbol{m}'$ si $\boldsymbol{m}'$ est couvrable à partir de $\boldsymbol{m}$.

9.3.1 Graphes de couverture

Considérons le réseau de Petri illustré à la figure 9.4. Cherchons à déterminer si $\boldsymbol{m} = (1, 1)$ peut couvrir un marquage $\boldsymbol{m}'$. Nous pourrions tenter de construire $\text{Post}^*(1, 1)$ sous forme de graphe d'accessibilité tel qu'illustré du côté gauche de la figure 9.5. Cependant, ce graphe est infini et il serait à priori impossible de déterminer lorsqu'il faut arrêter de construire le graphe.

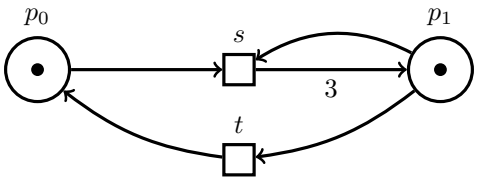


FIGURE 9.4 – Autre exemple de réseau de Petri.

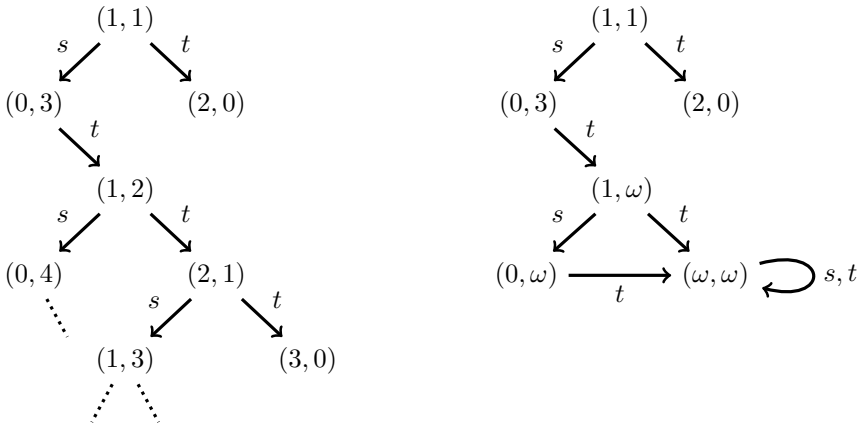


FIGURE 9.5 – Graphe d'accessibilité (gauche) et graphe de couverture (droite) du réseau de Petri de la figure 9.4 à partir du marquage $(1, 1)$.

Afin de pallier ce problème, nous introduisons la notion de graphe de couverture. Nous étendons $\mathbb{N}$ avec un élément maximal ω. Plus formellement, nous

définissons l'ensemble $\mathbb{N}_\omega \coloneqq \mathbb{N} \cup \{\omega\}$ où

$$\begin{aligned} n + \omega &= \omega && \text{pour tout } n \in \mathbb{N}_\omega, \\ \omega - n &= \omega && \text{pour tout } n \in \mathbb{N}, \\ \omega &> n && \text{pour tout } n \in \mathbb{N}. \end{aligned}$$

Un *marquage étendu* est un vecteur $\boldsymbol{m} \in \mathbb{N}_\omega^P$. Le concept de déclenchement de transitions est étendu naturellement à ces marquages.

Reconsidérons le graphe d'accessibilité à la gauche de la figure 9.5. Lorsque nous atteignons le marquage $(1, 2)$, nous observons que $(1, 1) \xrightarrow{st} (1, 2)$ et $(1, 2) \geq (1, 1)$. En itérant la séquence st, nous pouvons donc faire croître arbitrairement la deuxième place. Nous remplaçons donc ce marquage par le marquage étendu $(1, \omega)$, que nous appelons son *accélération*. Le symbole « ω » n'indique pas que toutes les valeurs sont atteignables, mais bien que la place peut contenir autant de jetons que désiré. Ainsi, en inspectant les ancêtres d'un marquage lors de son ajout, nous obtenons un graphe *fini* qui capture la forme des marquages couvrables, comme celui du côté droite de la figure 9.5.

La procédure de calcul d'un graphe de couverture est décrite à l'algorithme 14.

Algorithme 14 : Algorithme de calcul de graphe de couverture.

Entrées : Réseau de Petri $\mathcal{N} = (P, T, F)$ et $\boldsymbol{m} \in \mathbb{N}^P$
Sorties : Graphe de couverture à partir de $\boldsymbol{m}$

```
cover(𝒩, m):
    V ← ∅                                              // Sommets
    E ← ∅                                              // Arcs
    W ← {m}                                            // À traiter
    tant que W ≠ ∅
        m′ ← retirer de W
        ajouter m′ à V
        pour t ∈ T déclenchable en m′
            m″ ← marquage tel que m′ —t→ m″
            accel(m″)
            si m″ ∉ V alors                            // Déjà traité?
                ajouter m″ à W
            ajouter (m′, m″) à E
    retourner (V, E)

    accel(x):
        pour tout ancêtre x′ de x dans le graphe (V, E)
            si x′ ≤ x alors
                pour p ∈ P
                    si x′(p) < x(p) alors
                        x(p) ← ω
```

La proposition suivante explique comment déterminer si un marquage $\boldsymbol{m}'$ est couvrable à l'aide d'un graphe de couverture:

Proposition 10. *Soit $\mathcal{N} = (P, T, F)$ un réseau de Petri, et soient deux marquages $\boldsymbol{m}, \boldsymbol{m}' \in \mathbb{N}^P$. Soit G un graphe de couverture calculé par* `cover`$(\mathcal{N}, \boldsymbol{m})$*. Le marquage $\boldsymbol{m}'$ est couvrable à partir de $\boldsymbol{m}$ si et seulement si G possède un marquage $\boldsymbol{m}''$ tel que $\boldsymbol{m}'' \geq \boldsymbol{m}'$.*

Notons que l'algorithme termine bel et bien sur toute entrée et qu'ainsi un graphe de couverture est nécessairement fini:

Proposition 11. *L'algorithme 14 termine sur toute entrée.*

Démonstration. Afin d'obtenir une contradiction, supposons qu'il existe une entrée sur laquelle l'algorithme ne termine pas. L'algorithme calcule donc un graphe infini G. Observons que chaque sommet de G possède au plus $|T|$ successeurs immédiats; donc un nombre *fini* de successeurs immédiats. Par le lemme de König, G contient donc un chemin simple infini $\boldsymbol{m}_0 \to \boldsymbol{m}_1 \to \cdots$.

Pour tout $i \in \mathbb{N}$, définissons

$$[\![\boldsymbol{m}_i]\!] \coloneqq \{p \in P : \boldsymbol{m}_i(p) = \omega\}.$$

Comme le chemin est infini et que P est fini, il existe des indices $i_0 < i_1 < \cdots$ tels que $[\![\boldsymbol{m}_{i_0}]\!] = [\![\boldsymbol{m}_{i_1}]\!] = \cdots$. Ainsi, par le lemme de Dickson, il existe $j, k \in \mathbb{N}$ tels que $j < k$ et $\boldsymbol{m}_{i_j} \leq \boldsymbol{m}_{i_k}$. Notons que $\boldsymbol{m}_{i_j} \neq \boldsymbol{m}_{i_k}$, puisqu'autrement le chemin ne serait pas simple. Ainsi, $\boldsymbol{m}_{i_j}(p) < \boldsymbol{m}_{i_k}(p)$ pour au moins une place $p \in P$. Si $\boldsymbol{m}_{i_k}(p) = \omega$, alors $[\![\boldsymbol{m}_{i_j}]\!] \neq [\![\boldsymbol{m}_{i_k}]\!]$, ce qui est une contradiction. Nous avons donc $\boldsymbol{m}_{i_k}(p) \in \mathbb{N}$. Cela implique que $\boldsymbol{m}_{i_k}$ aurait dû être accéléré par son ancêtre $\boldsymbol{m}_{i_j}$, ce qui n'est pas le cas puisque $[\![\boldsymbol{m}_{i_j}]\!] = [\![\boldsymbol{m}_{i_k}]\!]$. □

9.3.2 Algorithme arrière

Nous étudions un autre algorithme qui permet de résoudre le problème de couverture: l'*algorithme arrière*. Plutôt que d'identifier les marquages couvrables, cet algorithme identifie plutôt les marquages qui peuvent couvrir un marquage donné. L'algorithme arrière repose notamment sur la représentation et la manipulation d'ensembles dits clos par le haut.

Soit un marquage $\boldsymbol{m} \in \mathbb{N}^P$. La *clôture par le haut* de $\boldsymbol{m}$ est l'ensemble de marquages $\uparrow\boldsymbol{m} \coloneqq \{\boldsymbol{m}' \in \mathbb{N}^P : \boldsymbol{m}' \geq \boldsymbol{m}\}$. La *clôture par le haut* d'un ensemble $X \subseteq \mathbb{N}^P$ est l'ensemble:

$$\uparrow X \coloneqq \bigcup_{\boldsymbol{m} \in X} \uparrow\boldsymbol{m}.$$

Nous disons que $X \subseteq \mathbb{N}^P$ est *clos par le haut* si $\uparrow X = X$. Une *base* d'un ensemble clos par le haut X est un ensemble B tel que $\uparrow B = X$. Une base B est *minimale* si ce n'est *pas* le cas qu'il existe $\boldsymbol{x}, \boldsymbol{y} \in B$ tels que $\boldsymbol{x} \geq \boldsymbol{y}$ et $\boldsymbol{x} \neq \boldsymbol{y}$. Autrement dit, B est minimale si tous ses éléments sont incomparables. La figure 9.6 donne un exemple d'ensemble clos par le haut et de base minimale.

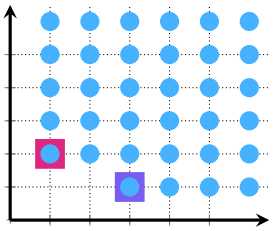


FIGURE 9.6 – Illustration d'un ensemble clos par le haut (cercles). Sa base minimale $\{(1,2),(3,1)\}$ est représentée par des carrés.

Observons que l'ensemble des marquages qui peuvent couvrir un certain marquage est clos par le haut:

Proposition 12. *Soit $\boldsymbol{m}' \in \mathbb{N}^P$. L'ensemble des marquages qui peuvent couvrir $\boldsymbol{m}'$ est égal à $\uparrow\mathrm{Pre}^*(\uparrow\boldsymbol{m}')$.*

Démonstration. $\subseteq$) Soit $\boldsymbol{m}$ un marquage qui peut couvrir $\boldsymbol{m}'$. Par définition de couverture, il existe un marquage $\boldsymbol{m}''$ tel que $\boldsymbol{m} \xrightarrow{*} \boldsymbol{m}''$ et $\boldsymbol{m}'' \geq \boldsymbol{m}'$. Ainsi,

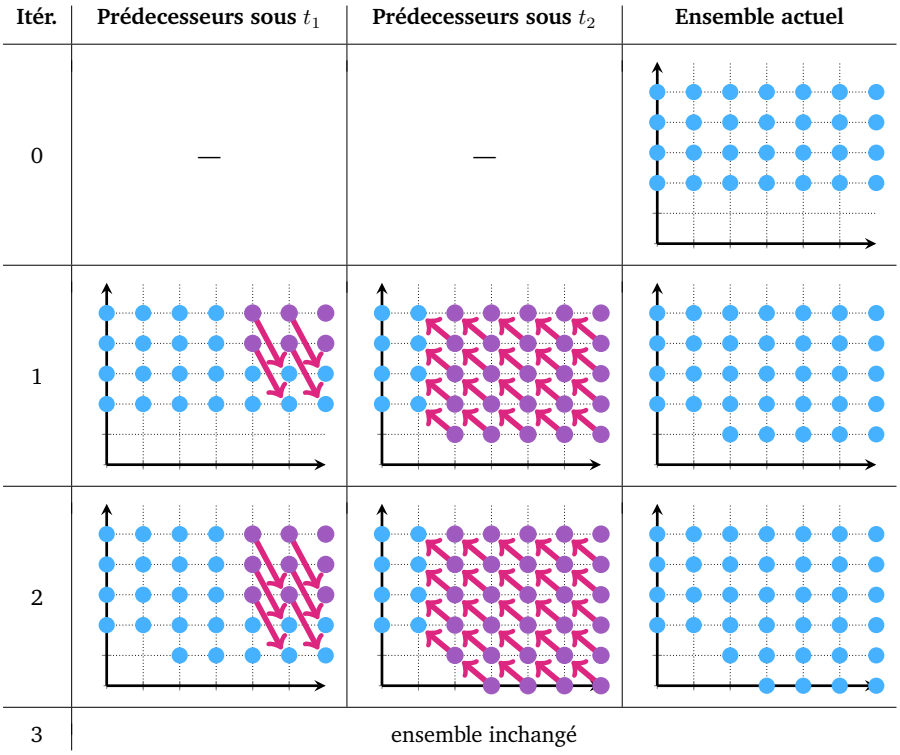
$$\begin{aligned}
\boldsymbol{m} &\in \mathrm{Pre}^*(\boldsymbol{m}'') && \text{(par définition de Pre}^*\text{)}\\
&\subseteq \mathrm{Pre}^*(\uparrow\boldsymbol{m}') && \text{(car } \boldsymbol{m}'' \in \uparrow\boldsymbol{m}'\text{)}\\
&\subseteq \uparrow\mathrm{Pre}^*(\uparrow\boldsymbol{m}') && \text{(par l'identité } X \subseteq \uparrow X\text{)}.
\end{aligned}$$

$\supseteq$) Soit $\boldsymbol{m} \in \uparrow\mathrm{Pre}^*(\uparrow\boldsymbol{m}')$. Il existe des marquages $\boldsymbol{k}$ et $\boldsymbol{m}'''$ tels que

$$\begin{array}{ccc}
\boldsymbol{m} & & \\
\vee| & & \\
\boldsymbol{k} & \xrightarrow{*} & \boldsymbol{m}'' \\
 & & \vee| \\
 & & \boldsymbol{m}'
\end{array}$$

Ainsi, par monotonicité, il existe un marquage $\boldsymbol{m}'''$ tel que $\boldsymbol{m} \xrightarrow{*} \boldsymbol{m}'''$ et $\boldsymbol{m}''' \geq \boldsymbol{m}'' \geq \boldsymbol{m}'$. Informellement, la « monotonicité » signifie qu'un plus grand budget de jetons permet de déclencher (au moins) autant de transitions. Nous concluons donc que $\boldsymbol{m}$ peut couvrir $\boldsymbol{m}'$. □

L'algorithme arrière cherche à calculer une représentation de $\uparrow\mathrm{Pre}^*(\uparrow\boldsymbol{m}')$. Par exemple, considérons le réseau de Petri illustré à la figure 9.7 avec $\boldsymbol{m}' = (0,2)$. L'algorithme arrière débute avec l'ensemble $\uparrow\boldsymbol{m}'$, puis calcule itérativement les prédécesseurs immédiats de chacun de ses marquages jusqu'à ce que l'ensemble se stabilise:



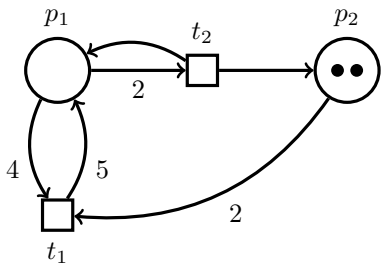


FIGURE 9.7 – Un réseau de Petri.

Puisque les ensembles clos par le haut sont infinis, cette procédure n'est pas, à priori, effective. Pour qu'elle le soit, nous devons être en mesure de:

— représenter les ensembles clos par le haut symboliquement;

— calculer les prédécesseurs immédiats de tous les marquages d'un ensemble clos par le haut.

Le premier point est rendu possible par cette observation:

Proposition 13. *Tout ensemble $X \subseteq \mathbb{N}^P$ clos par le haut a une base finie.*

Ainsi, nous manipulons des bases finies comme représentants d'ensembles clos par le haut. En particulier, il est possible de tester l'appartenance à un ensemble clos par le haut représenté par une base B puisque:

$$\boldsymbol{m} \in {\uparrow}B \iff \text{il existe } \boldsymbol{k} \in B \text{ tel que } \boldsymbol{m} \geq \boldsymbol{k}.$$

Pour le second point, nous calculons pour chaque élément de la base et pour chaque transition, le plus petit marquage pouvant le couvrir sous cette transition. Formellement, soit $t \in T$ une transition et $\boldsymbol{m} \in \mathbb{N}^P$ un marquage. Le *plus petit marquage pouvant couvrir $\boldsymbol{m}$ en déclechant t* est le marquage $\boldsymbol{m}_t$ tel que

$$\boldsymbol{m}_t(p) \coloneqq \max(\underbrace{F(p,t)}_{\text{« budget nécessaire »}}, \underbrace{\boldsymbol{m}(p) + F(p,t) - F(t,p)}_{\text{« déclenchement arrière »}}) \qquad \text{pour tout } p \in P.$$

L'algorithme complet est décrit à l'algorithme 15. Reconsidérons le réseau de Petri illustré à la figure 9.7 avec $\boldsymbol{m}' = (0,2)$. Nous exécutons cette fois l'algorithme arrière en manipulant directement une base:

Itér.	**Base B**	**Prédécesseurs**
0	$\{(0,2)\}$	$(0,2)_{t_1} = (4,4)$ $(0,2)_{t_2} = (2,1)$
1	$\{(0,2),(2,1)\}$	$(2,1)_{t_1} = (4,3)$ $(2,1)_{t_2} = (3,0)$
2	$\{(0,2),(2,1),(3,0)\}$	$(3,0)_{t_1} = (4,2)$ $(3,0)_{t_2} = (4,0)$
3	$\{(0,2),(2,1),(3,0)\}$	base inchangée

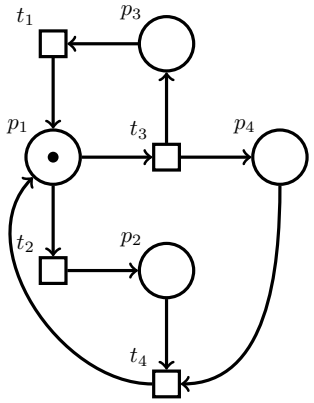
Algorithme 15 : Algorithme arrière.

Entrées : Réseau de Petri $\mathcal{N} = (P,T,F)$ et $\boldsymbol{m}' \in \mathbb{N}^P$
Sorties : Une base minimale de ${\uparrow}\mathrm{Pre}^*({\uparrow}\boldsymbol{m}')$

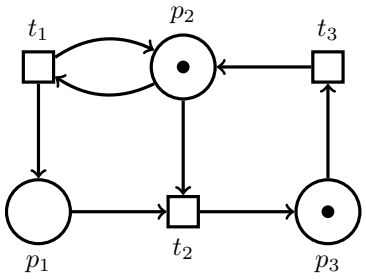
```
arriere(N, m'):
    B' ← {m'}
    faire
        B ← B'                                   // Base actuelle
        pour m ∈ B
            pour t ∈ T
                ajouter m_t à B'
        minimiser(B')        // Retirer les marquages redondants
    tant que B' ≠ B
    retourner B
minimiser(X):
    pour x ∈ X
        pour y ∈ X
            si x ≤ y ∧ x ≠ y alors retirer y de X
```

9.4 Exercices

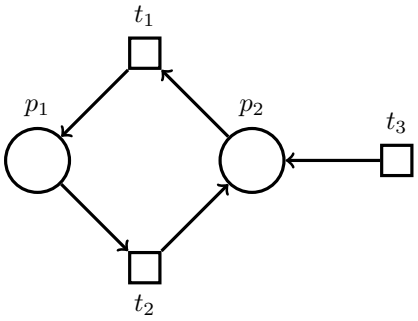
9.1) Calculez un graphe de couverture de ce réseau de Petri à partir de $(1, 0, 0, 0)$:



9.2) Exécutez l'algorithme arrière sur ce réseau de Petri à partir de $(0, 1, 1)$:



9.3) Montrez que l'ordre dans lequel on construit un graphe de couverture peut avoir une incidence sur sa taille. Considérez ce réseau de Petri:



Remarque.

L'ancienne version de l'exercice débutait par le marquage $(1, 1)$, mais on obtient un contre-exemple plus simple à partir de $(0, 0)$.

9.4) Montrez la propriété de monotonicité, c.-à-d. que $\boldsymbol{m} \xrightarrow{t} \boldsymbol{m}'$ et $\boldsymbol{k} \geq \boldsymbol{m}$ implique l'existence de $\boldsymbol{k}' \geq \boldsymbol{m}'$ tel que $\boldsymbol{k} \xrightarrow{t} \boldsymbol{k}'$.

9.5) Considérons une variable entière `x` de n bits qui peut être incrémentée.

a) Modélisez `x` à l'aide de 8 places pour $n = 3$, puis généralisez.

b) Modélisez `x` à l'aide de 6 places pour $n = 3$, puis généralisez.

c) Modélisez le test « `x == 0` ».

d) En supposant que `x` est signée et représentée sous complément à deux, modélisez le test « `x > 0` ».

9.6) Considérons cette (sous) interface Java:

```
                                java.awt.geom
new AffineTransformation()
Shape Shape.createTransformedShape(AffineTransformation)
String Point2D.ToString()
double Point2D.getX()
double Point2D.getY()
void AffineTransformation.setToRotation(double, double, double)
void AffineTransformation.invert()
Area Area.createTransformedArea(AffineTransformation)
```

a) Modélisez l'interface au niveau des types, c.-à-d. en assignant une place par type et une transition par méthode.

b) Dans cette modélisation, quel est le marquage $\boldsymbol{m}$ associé à cette nouvelle méthode qu'on voudrait implémenter:

```
Area rotate(Area object, Point2D point, double angle)
```

c) Donnez une séquence de transitions qui mène à $\boldsymbol{m}$.

(*adapté de [BHO20] basé sur [FMW+17]*)

10

Systèmes probabilistes

Le non-déterminisme utilisé jusqu'ici s'avère pratique afin de modéliser des comportements à priori incertains. Toutefois, il ne permet ni de distinguer entre comportements fortement et faiblement probables, ni d'évaluer la probabilité qu'une propriété soit satisfaite. Parfois, une distribution de probabilité (ou une estimation) est connue. Ainsi, il devient intéressant de distinguer, par ex., entre « la spécification peut être enfreinte » et « la spécification peut être enfreinte, mais avec faible probabilité ». Nous introduisons un formalisme qui remplace le non-déterminisme des structures de Kripke par des probabilités, et qui permet l'automatisation de la vérification de telles questions.

10.1 Chaînes de Markov

Définition 5. Une *chaîne de Markov* [1] est un tuple $\mathcal{M} = (S, \mathbf{P}, \mathbf{init}, AP, L)$ où

- $(S, \to, I)$ est un système de transition où $\to \coloneqq \{(s, s') : \mathbf{P}(p, p') > 0\}$,
- $\mathbf{P}\colon S \times S \to \mathbb{Q}_{[0,1]}$ est une matrice qui associe des probabilités aux transitions, et qui satisfait $\sum_{s' \in S} \mathbf{P}(s, s') = 1$ pour tout état $s \in S$,
- $\mathbf{init}\colon S \to \mathbb{Q}_{[0,1]}$ est un vecteur qui indique la probabilité $\mathbf{init}(s)$ de débuter dans l'état s, et qui satisfait $\sum_{s \in S} \mathbf{init}(s) = 1$,
- AP est un ensemble de *propositions atomiques*,
- $L\colon S \to 2^{AP}$ est une fonction d'*étiquetage*.

Nous supposons pour le reste du chapitre qu'une chaîne de Markov est finie. Les notions de chemins, d'exécutions, de successeurs et de prédécesseurs s'appliquent directement à une chaîne de Markov: nous considérons le système de transition qu'il induit en ignorant la valeur précise des probabilités. Pour faciliter la présentation, nous supposerons qu'une chaîne de Markov ne possède pas d'état terminal (c.-à-d. sans successeur immédiat).

1. Plus précisément, une *chaîne de Markov à temps discret*.

Exemple.

Considérons la figure 10.1 qui illustre une chaîne de Markov, sans propositions atomiques. Celle-ci modélise un système de communication simple qui tente d'envoyer un message à répétition sur un canal non fiable.

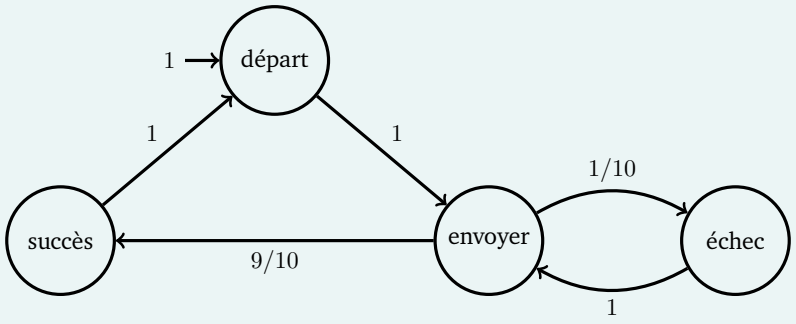


FIGURE 10.1 – Chaîne de Markov, sans propositions atomiques, qui modélise un système de communication (tirée de [BK08]).

En ordonnant les états $S = \{\text{départ}, \text{envoyer}, \text{succès}, \text{échec}\}$, nous avons:

$$\mathbf{P} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 9/10 & 1/10 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \quad \text{et} \quad \mathbf{init} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}.$$

Afin de raisonner sur les propriétés d'une chaîne de Markov $\mathcal{M}$, nous devons lui associer un espace de probabilité. À cette fin, nous associons à chaque chemin fini $\rho = s_0 s_1 \cdots s_n$ de $\mathcal{M}$, ce *cylindre* et cette valeur:

$$\begin{aligned} \text{cyl}(\rho) &\coloneqq \{\rho' : \rho' \text{ est un chemin infini qui débute par } \rho\}, \\ \mathbf{P}(s_0 \cdots s_n) &\coloneqq \prod_{0 \le i < n} \mathbf{P}(s_i, s_{i+1}). \end{aligned}$$

L'espace de probabilité de $\mathcal{M}$ est défini comme suit:

- son univers est l'ensemble des exécutions infinies de $\mathcal{M}$;
- ses événements est l'ensemble qui contient le cylindre de chaque chemin fini de $\mathcal{M}$, et qui est clos par complémentation et par union dénombrable;
- sa mesure de probabilité $\mathbb{P}$ satisfait $\mathbb{P}(\text{cyl}(s_0 \cdots s_n)) \coloneqq \mathbf{init}(s_0) \cdot \mathbf{P}(s_0 \cdots s_1)$.

Afin de spécifier des ensembles de chemins infinis plus aisément, nous utilisons la notation LTL, où un état est vu comme une proposition atomique et un

ensemble $A = \{s_1, \ldots, s_k\}$ comme l'expression $s_1 \vee \cdots \vee s_k$. Par exemple, $\mathsf{F}A$ est l'ensemble des chemins infinis qui passent au moins une fois par un état de A, et $\mathsf{G}A$ est l'ensemble des chemins infinis qui ne visitent que des états de A. La probabilité de ces ensembles existe car ils sont des événements, par la proposition ci-dessous. En particulier, nous écrirons $\mathbb{P}(s \models \varphi)$ pour dénoter $\mathbb{P}(\varphi)$ où s serait l'unique état initial de $\mathcal{M}$. Par exemple, $\mathbb{P}(s \models \mathsf{F}s')$ est la probabilité d'atteindre l'état s' à partir de l'état s. De façon générale, nous avons donc

$$\mathbb{P}(\varphi) = \sum_{s \in S} \mathbf{init}(s) \cdot \mathbb{P}(s \models \varphi).$$

Proposition 14. *Les ensembles engendrés par la notation LTL sont mesurables.*

Démonstration. Nous montrons seulement le cas de $\mathsf{F}A$. Nous devons montrer que cet ensemble est un événement en l'exprimant par rapport à des cylindres. Un chemin infini visite A au moins une fois ssi il possède un préfixe fini (possiblement vide) dans $S \setminus A$ suivi d'un état de A et de n'importe quoi. Ainsi:

$$\mathsf{F}A = \bigcup_{\rho \in (S \setminus A)^*} \bigcup_{s \in A} \mathrm{cyl}(\rho \cdot s). \qquad \square$$

En particulier, la probabilité de l'événement $\mathsf{F}A$ est:

$$\begin{aligned}
\mathbb{P}(\mathsf{F}A) &= \mathbb{P}\Big(\bigcup_{s_0 \cdots s_n \in (S \setminus A)^* A} \mathrm{cyl}(s_0 \cdots s_n)\Big) \\
&= \sum_{s_0 \cdots s_n \in (S \setminus A)^* A} \mathbb{P}(\mathrm{cyl}(s_0 \cdots s_n)) && \text{(car les cylindres sont indépendants)} \\
&= \sum_{s_0 \cdots s_n \in (S \setminus A)^* A} \mathbf{init}(s_0) \cdot \mathbf{P}(s_0 \cdots s_n) && \text{(par déf. de } \mathbb{P}\text{)}.
\end{aligned}$$

Exemple.

Reconsidérons la chaîne de Markov $\mathcal{M}$ de la figure 10.1. Calculons la probabilité que $\mathcal{M}$ atteigne l'état de succès. Remarquons que l'unique

état initial est l'état de départ. Ainsi, nous avons:

$$\begin{aligned}\mathbb{P}(\mathsf{F}\ \text{succès}) &= \sum_{i=0}^{\infty} \mathbf{P}(\text{départ envoyer (échec envoyer)}^i \text{ succès}) \\ &= \sum_{i=0}^{\infty} 1 \cdot (1/10)^i \cdot (9/10) \\ &= (9/10) \cdot \sum_{i=0}^{\infty} (1/10)^i \\ &= (9/10) \cdot (1/(1-1/10)) \qquad (10.1) \\ &= 1,\end{aligned}$$

où (10.1) découle du fait qu'il s'agit d'une série géométrique.

10.2 Probabilités d'accessibilité

Cherchons à exprimer la probabilité qu'un certain état s atteigne un ensemble d'états B. Si $s \in B$, alors celle-ci est de 1. Si s peut atteindre un état de B, alors la probabilité est de 0. Autrement dit:

$$\mathbb{P}(s \models \mathsf{F}B) = \begin{cases} 0 & \text{si } \mathrm{Post}^*(s) \cap B = \emptyset, \\ 1 & \text{si } s \in B, \\ ? & \text{sinon.} \end{cases}$$

Dans le cas général indiqué par « ? », un état s atteint l'ensemble B en plus d'une étape, ou en exactement une étape. Ainsi:

$$\mathbb{P}(s \models \mathsf{F}B) = \sum_{s' \in S \setminus B} \mathbf{P}(s, s') \cdot \mathbb{P}(s' \models \mathsf{F}B) + \sum_{s' \in B} \mathbf{P}(s, s').$$

La première somme se simplifie légèrement. En effet, il est inutile de considérer les états s' qui ne peuvent pas atteindre B car leur probabilité est de zéro.

Réécrivons le tout de façon plus concise. Posons:

$$\begin{aligned} S_0 &:= \{s \in S : \mathrm{Post}^*(s) \cap B = \emptyset\}, \\ S_1 &:= B, \\ S_? &:= S \setminus (S_0 \cup S_1). \end{aligned}$$

Soit $\mathbf{A}$ la matrice $\mathbf{P}$ restreinte à $S_?$, et soit $\boldsymbol{b}$ le vecteur tel que $\boldsymbol{b}(s) = \sum_{s' \in S_1} \mathbf{P}(s, s')$. Le vecteur $\boldsymbol{x} \colon S_? \to \mathbb{Q}_{[0,1]}$ tel que $\boldsymbol{x}(s) = \mathbb{P}(s \models \mathsf{F}B)$ est une solution de:

$$\boldsymbol{x} = \mathbf{A} \cdot \boldsymbol{x} + \boldsymbol{b}, \text{ qui s'écrit aussi } (\mathbf{I} - \mathbf{A}) \cdot \boldsymbol{x} = \boldsymbol{b}.$$

En fait, on peut démontrer que ce système possède une *unique* solution.

Exemple.

Reconsidérons l'événement « F succès » de la chaîne de Markov $\mathcal{M}$ illustrée à la figure 10.1. Nous avons $S_0 = \emptyset$, $S_1 = \{\text{succès}\}$ et $S_? = \{\text{départ}, \text{envoyer}, \text{échec}\}$. Nous obtenons ce système:

$$\left[\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} - \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1/10 \\ 0 & 1 & 0 \end{pmatrix}\right] \cdot \boldsymbol{x} = \begin{pmatrix} 0 \\ 9/10 \\ 0 \end{pmatrix}.$$

L'unique solution du système est $(1, 1, 1)$.

Ce calcul de probabilités se généralise à un événement de la forme $A \cup B$:

Théorème 4 ([BK08, Thm. 10.15])**.** *Soit $\mathcal{M} = (S, \mathbf{P}, \mathbf{init}, AP, L)$ une chaîne de Markov. Soient $A, B \subseteq S$. Posons*

$$\begin{aligned} S_0 &\coloneqq [\![\neg\exists(A \cup B)]\!], \\ S_1 &\coloneqq B, \\ S_? &\coloneqq S \setminus (S_0 \cup S_1). \end{aligned}$$

Soient $\mathbf{A}$ la matrice $\mathbf{P}$ restreinte à $S_?$, et $\boldsymbol{b}$ le vecteur sur $S_?$ où $\boldsymbol{b}(s) \coloneqq \sum_{s' \in S_1} \mathbf{P}(s, s')$. Le vecteur $\boldsymbol{x} \colon S_? \to \mathbb{Q}_{[0,1]}$ tel que $\boldsymbol{x}(s) = \mathbb{P}(s \models (A \cup B))$ est l'unique solution de:

$$\boldsymbol{x} = \mathbf{A} \cdot \boldsymbol{x} + \boldsymbol{b},$$

ou, de façon équivalente: $\boldsymbol{x} = \lim_{n\to\infty} f^n(\mathbf{0})$ où $f(\boldsymbol{y}) \coloneqq \mathbf{A} \cdot \boldsymbol{y} + \boldsymbol{b}$. De plus, $f^n(\mathbf{0})$ est le vecteur de probabilités du cas où B doit être atteint en au plus n étapes.

Notons que la seconde formulation du théorème (en terme de limite) permet d'approximer la solution *numériquement* en calculant itérativement les vecteurs $\mathbf{0}, f(\mathbf{0}), f(f(\mathbf{0})), \ldots$ jusqu'à une marge d'erreur jugée acceptable.

Exemple.

Considérons l'événement $\{\text{envoyer}, \text{échec}\} \cup \{\text{départ}, \text{succès}\}$ de la chaîne de Markov $\mathcal{M}$ de la figure 10.1. Nous avons $S_0 = \emptyset$, $S_1 = \{\text{départ}, \text{succès}\}$ et $S_? = \{\text{envoyer}, \text{échec}\}$. Nous obtenons ce système:

$$\left[\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} - \begin{pmatrix} 0 & 1/10 \\ 1 & 0 \end{pmatrix}\right] \cdot \boldsymbol{x} = \begin{pmatrix} 9/10 \\ 0 \end{pmatrix}.$$

L'unique solution du système est $(1, 1)$. Alternativement, en évaluant f à répétition, nous convergeons vers $(1, 1)$:

$$\begin{pmatrix} 0 \\ 0 \end{pmatrix}; \begin{pmatrix} 0{,}9 \\ 0{,}0 \end{pmatrix}; \begin{pmatrix} 0{,}99 \\ 0{,}90 \end{pmatrix}; \begin{pmatrix} 0{,}999 \\ 0{,}990 \end{pmatrix}; \ldots$$

10.3 Probabilités de comportements limites

Les exemples précédents mènent tous à des probabilités de 1, ce qui n'est pas une coïncidence. Cela découle du fait que le graphe induit par la chaîne de Markov est fortement connexe [2]. Nous disons qu'une composante fortement connexe (*CFC*) est *terminale* si elle ne peut pas en atteindre d'autre. En général, toute exécution infinie ρ atteint une composante fortement connexe terminale C avec probabilité 1. De plus, tous les états de C sont visités par ρ avec probabilité 1. Plus formellement, nous dénotons par $\inf(\rho)$ l'ensemble des états visités infiniment souvent par ρ. Nous avons:

Proposition 15. *Soient un état s et $E \coloneqq \{\rho : \inf(\rho)$ est une CFC terminale$\}$. Nous avons $\mathbb{P}(s \models E) = 1$.*

Ainsi, peu importe le point de départ de la chaîne de Markov de la figure 10.1, l'état « succès » est visité infiniment souvent avec probabilité 1, ce qui explique le résultat des exemples.

De plus, la proposition 15 permet d'évaluer certaines « propriétés limites » en termes d'accessibilité. Par exemple, $\mathbb{P}(s \models \mathsf{GF}A) = \mathbb{P}(s \models \mathsf{F}A')$ où

$$A' \coloneqq \{s \in S : s \text{ appartient à une CFC terminale } C \text{ t.q. } C \cap A \neq \emptyset\}.$$

Similairement, $\mathbb{P}(s \models \mathsf{FG}A) = \mathbb{P}(s \models \mathsf{F}A')$ où

$$A' \coloneqq \{s \in S : s \text{ appartient à une CFC terminale } C \text{ t.q. } C \subseteq A\}.$$

Comme on peut identifier les composantes fortement connexes terminales d'un graphe en temps linéaire, ces probabilités se calculent en temps polynomial.

Exemple.

Considérons la chaîne de Markov suivante. Cherchons à évaluer $\mathbb{P}(\mathsf{FG}p)$.

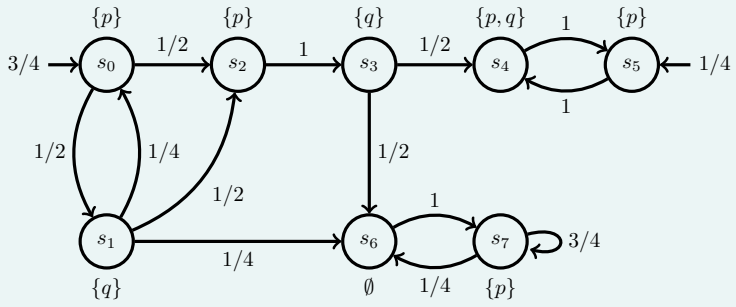


FIGURE 10.2 – Exemple de chaîne de Markov sur $AP = \{p, q\}$.

2. Une telle chaîne est dite *ergodique* ou *irréductible* dans le jargon mathématique.

Posons $A \coloneqq [\![p]\!] = \{s_0, s_2, s_4, s_5, s_7\}$. Nous avons:

$$\begin{aligned}
\mathbb{P}(\mathsf{FG}p) &= \mathbb{P}(\mathsf{FG}A) \\
&= \mathbb{P}(\mathsf{F}A') && \text{(où } A' \coloneqq \{s_4, s_5\}\text{)} \\
&= 3/4 \cdot \mathbb{P}(s_0 \models \mathsf{F}A') + 1/4 \cdot \mathbb{P}(s_5 \models \mathsf{F}A') && \text{(selon \textbf{init})} \\
&= 3/4 \cdot \mathbb{P}(s_0 \models \mathsf{F}A') + 1/4 && \text{(car } s_5 \in A'\text{)} \\
&= 3/4 \cdot \boldsymbol{x}(s_0) + 1/4 && \text{(où } (\mathbf{I} - \mathbf{A}) \cdot \boldsymbol{x} = \boldsymbol{b}\text{).}
\end{aligned}$$

Il nous reste à déterminer $\mathbf{A}$ et $\boldsymbol{b}$ afin d'identifier $\boldsymbol{x}$. Nous avons $S_0 = \{s_6, s_7\}$, $S_1 = \{s_4, s_5\}$ et $S_? = \{s_0, s_1, s_2, s_3\}$. Ainsi:

$$\mathbf{A} \coloneqq \begin{pmatrix} 0 & 1/2 & 1/2 & 0 \\ 1/4 & 0 & 1/2 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \text{ et } \boldsymbol{b} \coloneqq \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1/2 \end{pmatrix}.$$

En résolvant $(\mathbf{I} - \mathbf{A}) \cdot \boldsymbol{x} = \boldsymbol{b}$, nous obtenons $\boldsymbol{x} = (3/7, 5/14, 1/2, 1/2)$. En particulier, $\boldsymbol{x}(s_0) = 3/7$ et ainsi $\mathbb{P}(\mathsf{FG}p) = (3/4) \cdot (3/7) + (1/4) = 4/7$.

10.4 CTL probabiliste

10.4.1 Syntaxe et sémantique

Nous introduisons une variante probabiliste de CTL qui permet de vérifier plusieurs propriétés. La syntaxe de la *logique temporelle arborescente probabiliste (PCTL)* sur propositions atomiques *AP* est définie à partir de cette grammaire:

$$\begin{aligned}
\Phi &::= \mathit{vrai} \mid p \mid (\Phi \wedge \Phi) \mid \neg\Phi \mid \mathcal{P}_I(\varphi) && \text{(formule d'état)} \\
\varphi &::= \mathsf{X}\Phi \mid (\Phi \,\mathsf{U}\, \Phi) \mid (\Phi \,\mathsf{U}^{\leq n}\, \Phi) && \text{(formule de chemin)}
\end{aligned}$$

où $p \in \mathit{AP}$, $n \in \mathbb{N}$ et I dénote un intervalle de $\mathbb{Q}_{[0,1]}$.

La syntaxe diffère de celle de CTL en deux points. Premièrement, les quantificateurs de chemins $\exists$ et $\forall$ sont remplacés par l'opérateur $\mathcal{P}_I$ qui affirme que la probabilité de l'événement $[\![\varphi]\!]$ appartient à l'intervalle I, Par exemple, $\mathcal{P}_{[1/2,1]}(\mathsf{F}p)$, que nous abrégons par $\mathcal{P}_{\geq 1/2}(\mathsf{F}p)$, spécifie que la probabilité d'atteindre un état qui satisfait p est d'au moins $1/2$. Deuxièmement, le nouvel opérateur temporel $\mathsf{U}^{\leq n}$ borne le moment maximal auquel son terme de droite doit devenir vrai.

Formellement, la sémantique est définie comme suit. Pour tout état $s \in S$:

$$\begin{aligned} &s \models \mathit{vrai}, \\ &s \models p &&\overset{\text{déf}}{\iff} p \in L(s), \\ &s \models \neg\Phi &&\overset{\text{déf}}{\iff} \neg(s \models \Phi), \\ &s \models \Phi_1 \wedge \Phi_2 &&\overset{\text{déf}}{\iff} (s \models \Phi_1) \wedge (s \models \Phi_2), \\ &s \models \mathcal{P}_I(\varphi) &&\overset{\text{déf}}{\iff} \mathbb{P}(s \models \varphi) \in I. \end{aligned}$$

Pour tout chemin infini σ de la chaîne de Markov:

$$\begin{aligned} &\sigma \models \mathsf{X}\Phi &&\overset{\text{déf}}{\iff} \sigma(1) \models \Phi, \\ &\sigma \models \Phi_1 \,\mathsf{U}\, \Phi_2 &&\overset{\text{déf}}{\iff} \exists j \geq 0 &&: [(\forall i \in [0..j-1] : \sigma(i) \models \Phi_1) \wedge (\sigma(j) \models \Phi_2)], \\ &\sigma \models \Phi_1 \,\mathsf{U}^{\leq n}\, \Phi_2 &&\overset{\text{déf}}{\iff} \exists j \in [0..n] &&: [(\forall i \in [0..j-1] : \sigma(i) \models \Phi_1) \wedge (\sigma(j) \models \Phi_2)]. \end{aligned}$$

Notons que la sémantique de l'opérateur $\mathcal{P}_I$ est bien définie car toute formule de chemin donne lieu à un événement:

Proposition 16. *Pour toute formule de chemin φ et tout état s, $\mathbb{P}(s \models \varphi)$ est bien définie, c.-à-d. que cet ensemble est mesurable:*

$$\{\sigma \textit{ est un chemin infini} : \sigma \textit{ débute en } s \textit{ et } \sigma \models \varphi\}.$$

Nous introduisons les opérateurs temporels F et G à la manière de CTL en tenant compte des probabilités. Le *dual* d'un intervalle de probabilités[3] $I = [p, q]$ est $\text{dual}(I) := [1-q, 1-p]$. Nous définissons:

$$\begin{aligned} \mathsf{F}\Phi &:= \mathit{vrai}\ \mathsf{U}\ \Phi, & \mathsf{F}^{\leq n}\Phi &:= \mathit{vrai}\ \mathsf{U}^{\leq n}\ \Phi, \\ \mathcal{P}_I(\mathsf{G}\Phi) &:= \neg\mathcal{P}_{\text{dual}(I)}(\mathsf{F}\neg\Phi), & \mathcal{P}_I(\mathsf{G}^{\leq n}\Phi) &:= \neg\mathcal{P}_{\text{dual}(I)}(\mathsf{F}^{\leq n}\neg\Phi). \end{aligned}$$

Exemple.

La formule PCTL suivante spécifie que, presque sûrement, un message est reçu avec succès, et qu'une tentative d'envoi mène à un succès en au plus trois envois avec probabilité au moins $99/100$:

$$\mathcal{P}_{=1}(\mathsf{F}\,\text{succès}) \wedge \mathcal{P}_{=1}(\mathsf{G}(\text{envoyer} \rightarrow \mathcal{P}_{\geq 0,99}(\mathsf{F}^{\leq 3}\,\text{succès}))).$$

10.4.2 Vérification

Le problème de vérification PCTL consiste à déterminer si un état s satisfait une formule d'état Φ. Cela se détermine en temps $\mathcal{O}(\text{poly}(|\mathcal{M}|) \cdot |\Phi| \cdot m)$, où m est le plus grand exposant qui apparaît sur un opérateur $\mathsf{U}^{\leq n}$ (ou 1 s'il n'y en a pas).

3. Nous traitons les intervalles ouverts de façon analogue, par ex. $\text{dual}([p, q)) := (1-q, 1-p]$.

Nous savons déjà vérifier la portion propositionnelle de PCTL: nous calculons récursivement l'ensemble des états qui satisfont les sous-formules à l'aide d'opérations sur les ensembles. Il demeure donc d'identifier des algorithmes pour obtenir les états qui satisfont une formule de la forme $\mathcal{P}_I(\varphi)$.

Opérateur X**.** Nous avons $[\![\mathcal{P}_I(\mathsf{X}\Phi)]\!] = \{s \in S : \mathbb{P}(s \models \mathsf{X}\Phi) \in I\}$. De plus:

$$\mathbb{P}(s \models \mathsf{X}\Phi) = \sum_{s' \in [\![\Phi]\!]} \mathbf{P}(s, s'). \tag{10.2}$$

Ainsi, $[\![\mathcal{P}_I(\mathsf{X}\Phi)]\!]$ s'obtient en évaluant chaque probabilité avec (10.2) et en conservant les états dont la valeur appartenant à I.

Opérateur U**.** Nous avons

$$[\![\mathcal{P}_I(\Phi_1 \mathsf{U} \Phi_2)]\!] = \{s \in S : \mathbb{P}(s \models [\![\Phi_1]\!] \mathsf{U} [\![\Phi_2]\!]) \in I\}.$$

Nous savons comment calculer $\mathbb{P}(s \models [\![\Phi_1]\!] \mathsf{U} [\![\Phi_2]\!])$ par le théorème 4. Ainsi, comme pour l'opérateur X, il suffit d'évaluer les probabilités et de préserver les états dont la valeur appartient à l'intervalle I.

Opérateur $\mathsf{U}^{\leq n}$**.** Nous avons

$$[\![\mathcal{P}_I(\Phi_1 \mathsf{U} \Phi_2)]\!] = \{s \in S : \mathbb{P}(s \models [\![\Phi_1]\!] \mathsf{U}^{\leq n} [\![\Phi_2]\!]) \in I\}.$$

Pour évaluer les probabilités, nous exploitons le théorème 4 et évaluons $f^n(\mathbf{0})$.

Exemple.

Reconsidérons la chaîne de Markov $\mathcal{M}$ de la figure 10.1 et cette formule:

$$\Phi := \underbrace{\mathcal{P}_{=1}(\mathsf{F}\,\text{succès})}_{\Phi_1} \wedge \underbrace{\mathcal{P}_{=1}(\mathsf{G}(\text{envoyer} \to \mathcal{P}_{\geq 0{,}99}(\mathsf{F}^{\leq 3}\,\text{succès})))}_{\Phi_2}.$$

Par la proposition 15, $[\![\Phi_1]\!] = S$ car $\mathcal{M}$ est fortement connexe. Évaluons $\mathbb{P}(s \models \mathsf{F}^{\leq 3}\,\text{succès})$ pour chaque état s à l'aide du théorème 4. Nous avons $S_0 = \emptyset$, $S_1 = \{\text{succès}\}$ et $S_? = \{\text{départ}, \text{envoyer}, \text{échec}\}$,

$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1/10 \\ 0 & 1 & 0 \end{pmatrix} \text{ et } \boldsymbol{b} = \begin{pmatrix} 0 \\ 9/10 \\ 0 \end{pmatrix}.$$

En évaluant $f^3(\mathbf{0})$, où $f(\boldsymbol{y}) = \mathbf{A} \cdot \boldsymbol{y} + \boldsymbol{b}$, nous obtenons le vecteur

$$\boldsymbol{x} = (9/10, 99/100, 9/10).$$

Ainsi,

$$[\![\mathcal{P}_{\geq 0{,}99}(\mathsf{F}^{\leq 3}\,\text{succès})]\!] = \{s : \boldsymbol{x}(s) \geq 99/100\} = \{\text{envoyer}, \text{succès}\}.$$

Nous obtenons donc

$$\begin{aligned}
[\![\Phi_2]\!] &= [\![\mathcal{P}_{=1}(\mathsf{G}(\text{envoyer} \to \mathcal{P}_{\geq 0{,}99}(\mathsf{F}^{\leq 3}\,\text{succès})))]\!] \\
&= [\![\neg\mathcal{P}_{<1}(\mathsf{F}\neg(\text{envoyer} \to \mathcal{P}_{\geq 0{,}99}(\mathsf{F}^{\leq 3}\,\text{succès})))]\!] \\
&= [\![\neg\mathcal{P}_{<1}(\mathsf{F}(\text{envoyer} \wedge \neg\mathcal{P}_{\geq 0{,}99}(\mathsf{F}^{\leq 3}\,\text{succès})))]\!] \\
&= [\![\neg\mathcal{P}_{<1}(\mathsf{F}(\text{envoyer} \wedge (\text{départ} \vee \text{échec})))]\!] \\
&= [\![\neg\mathcal{P}_{<1}(\mathsf{F}\,\mathit{faux})]\!] \\
&= \overline{[\![\mathcal{P}_{<1}(\mathsf{F}\,\mathit{faux})]\!]} \\
&= \overline{\emptyset} \\
&= S.
\end{aligned}$$

Ainsi $[\![\Phi]\!] = [\![\Phi_1]\!] \cap [\![\Phi_2]\!] = S$ et, en particulier, l'état initial satisfait Φ.

10.5 Outils

La vérification formelle de chaînes de Markov (et de processus de décision markoviens) est supportée par des outils comme PRISM [KNP11] et Storm [HJK+20]. Par exemple, la chaîne de Markov de la figure 10.1 se modélise avec PRISM:

```
dtmc

const double p = 0.9;

module systeme
        // 0 = départ, 1 = envoyer, 2 = succès, 3 = échec
        etat : [0..3] init 0;

        [] (etat = 0) -> (etat' = 1);
        [] (etat = 1) -> p : (etat' = 2) + (1 - p) : (etat' = 3);
        [] (etat = 2) -> (etat' = 0);
        [] (etat = 3) -> (etat' = 1);
endmodule

// Propositions atomiques
label "envoyer" = (etat = 1);
label "succes"  = (etat = 2);
```

PRISM vérifie automatiquement les propriétés de l'exemple précédent:

```
P>=1 [ F "succes" ]
P>=1 [ G "envoyer" => P>=0.99 [ F<=3 "succes" ] ]
```

Il sait également calculer des probabilités. Par exemple, la requête suivante calcule $\mathbb{P}(\text{départ} \models \mathsf{F}^{\leq 3}\,\text{succès}) = 0{,}9$:

```
P=? [ F<=3 "succes" ]
```

PRISM peut aussi performer des analyses quantitatives. Par exemple, nous pouvons calculer l'espérance du nombre d'envois effectués avant un succès, en ajoutant ce bloc de code:

```
// Incrémente le nombre d'envois chaque fois que «envoyer» est atteint
rewards "nombre_envois"
        (etat = 0 | etat = 3) : 1;
        (etat = 1 | etat = 2) : 0;
endrewards
```

puis en effectuant cette requête qui retourne (une approximation de) $10/9$:

```
R{"nombre_envois"}=? [ F "succes" ]
```

10.6 Exercices

10.1) Soit $\mathcal{M}$ la chaîne de Markov de la figure 10.2.

a) Calculez $\mathbb{P}(s_0 \models \mathsf{F} s_2)$ à l'aide d'une somme exhaustive des chemins.

b) Calculez $\mathbb{P}(\mathsf{GF} p)$.

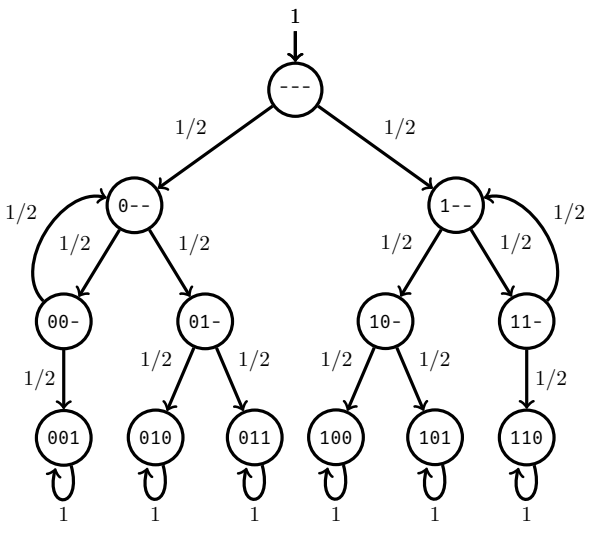
c) Calculez $\mathbb{P}(p \,\mathsf{U}\, q)$.

10.2) Cette fonction de Knuth et Yao cherche à simuler un dé à six faces à l'aide d'une pièce non biaisée:

```python
from random import randint

def lancer_de():
    x = randint(0, 1)

    while True:
        y = randint(0, 1)
        z = randint(0, 1)

        if not(x == y == z):
            break

    return 4*x + 2*y + z
```

a) Spécifiez en PCTL que la fonction « `lancer_de` » simule bien un dé, en raisonnant sur sa modélisation ci-dessous.

b) Vérifiez que l'état initial satisfait votre formule.



(*adapté de [BK08]*)

10.3) Nous disons qu'une formule CTL Φ et qu'une formule PCTL Φ' sont *équivalentes* lorsque $[\![\Phi]\!] = [\![\Phi']\!]$ pour toute structure de Kripke. Montrez que ces formules ne sont pas équivalentes à l'aide de contre-exemples:

a) $\Phi \coloneqq \forall \mathsf{F} p$ et $\Phi' \coloneqq \mathcal{P}_{=1}(\mathsf{F} p)$

b) $\Phi \coloneqq \exists \mathsf{G} p$ et $\Phi' \coloneqq \mathcal{P}_{>0}(\mathsf{G} p)$

10.4) Exprimez (10.2) en terme de matrice et de vecteur.

10.5) ★ Montrez que les ensembles $\mathsf{X} A$, $\mathsf{G} A$ et $A \mathsf{U} B$ sont mesurables.

10.6) ★ Dites pourquoi l'ensemble $A \coloneqq \{\rho : \inf(\rho) \text{ est une CFC terminale}\}$ défini à la proposition 15 est un événement.

10.7) Considérons la modélisation ci-dessous du protocole *IPv4 zeroconf* paramétré par $n = 3$, $a = 1/4$ et $b = 1/10$. Les propositions atomiques p_{IP}, $p_?$ et $p_\perp$ indiquent respectivement que l'hôte s'est fait attribuer une adresse IP; que l'hôte est en attente d'une réponse à une requête; et que l'hôte crée une collision sur le réseau. Spécifiez ces propriétés en PCTL:

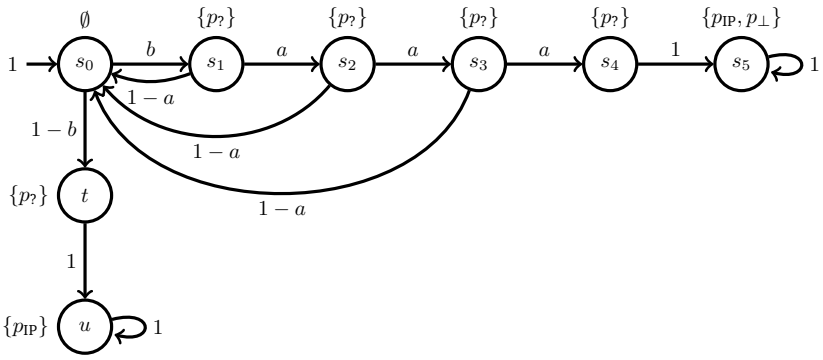
a) Une adresse IP est attribuée presque sûrement.

b) Avec probabilité au moins $1/2$, au plus deux requêtes sont effectuées avant qu'une adresse IP soit attribuée.

c) L'hôte crée une collision presque jamais.

d) Il y a une infinité de requêtes avec probabilité non nulle.

e) Presque sûrement, une adresse IP attribuée demeure attribuée.



(*adapté de [BK08]*)

Solutions des exercices

Cette section présente des solutions à certains des exercices du document. Dans certains cas, il ne s'agit que d'ébauches de solutions.

Chapitre 1

1.1)

a) Par ex. pour $s = s_2$:

$$\begin{aligned} \text{Post}(s_2) &= \{s_1, s_2, s_3\}, \\ \text{Pre}(s_2) &= \{s_1, s_2\}, \\ \text{Post}^*(s_2) &= \{s_1, s_2, \ldots, s_6\}, \\ \text{Pre}^*(s_2) &= \{s_0, s_1, s_2\}. \end{aligned}$$

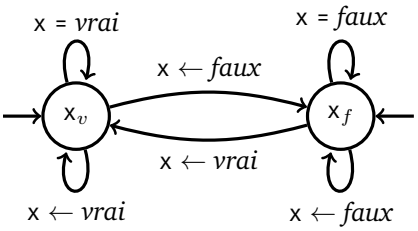
b) $s_0 \to s_1 \to s_2 \to s_3 \to s_4$.

c) Par ex. $s_0 \to s_0 \to s_0 \to \cdots$.

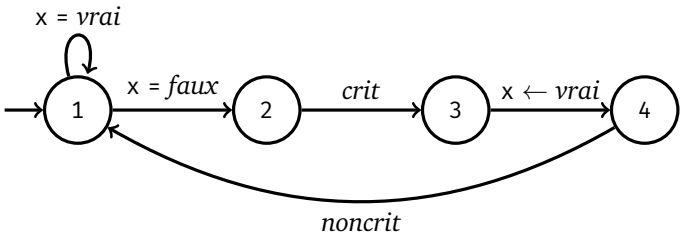
d) Par ex. $s_0 \to s_1 \to s_2 \to s_1 \to \cdots$ ou $s_0 \to s_5 \to s_6 \to s_6 \to \cdots$

1.2)

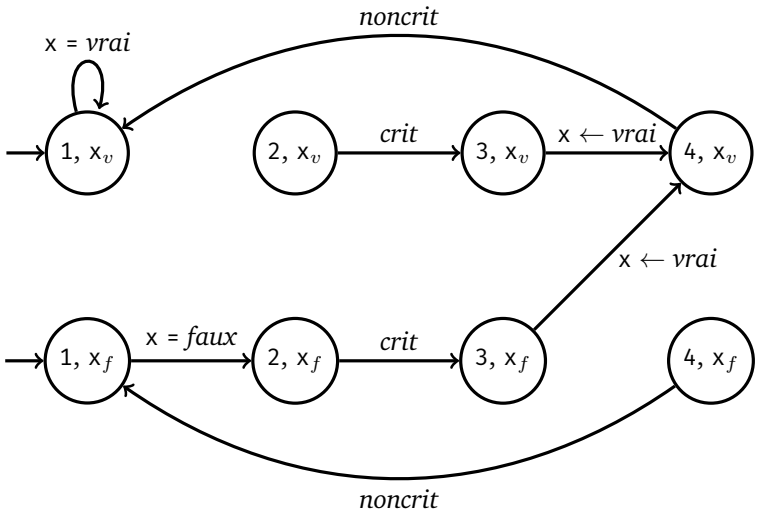
a)



b)



c)



d) Si les étiquettes sont les mêmes, alors on obtient une composition complètement synchrone où les systèmes doivent évoluer en même temps. Si elles sont disjointes, alors on obtient une composition où les deux systèmes évoluent indépendamment.

1.3) $\text{Post}^*(s) \cap \text{Pre}^*(s)$

1.4) Supposons que ce ne soit pas le cas. Puisque $\mathcal{T}$ est fini et que le chemin est infini, il existe une composante fortement connexe $C \subseteq S$ visitée infiniment souvent. En particulier, il existe donc un état $s_i \in C$ visité infiniment souvent. Par hypothèse, $\{s_i, s_{i+1} \ldots\} \not\subseteq C$. Ainsi, il existe un indice $j > i$ tel que $s_j \notin C$. Puisque s_i est visité infiniment souvent, il existe un indice $k > j$ tel que $s_k = s_i$.

Ainsi, $s_i \xrightarrow{*} s_j \xrightarrow{*} s_i$, ce qui implique la contradiction $s_j \in C$. □

Chapitre 2

2.1)

a) À un certain point, seul le feu vert est allumé.

b) On ne peut pas immédiatement passer du rouge au vert.

c) Lorsque le feu rouge est allumé, c'est aussi éventuellement le cas du jaune, puis du vert.

2.2) Il existe bien des solutions (une infinité), en voici des exemples:

a) — satisfait: $\emptyset\{vert\}\emptyset^\omega$
   — ne satisfait pas: $(\{vert, jaune\}\{rouge\})^\omega$

b) — satisfait: $(\{rouge\}\{rouge\}\{vert\})^\omega$
   — ne satisfait pas: $\{rouge\}(\{rouge\}\{vert\})^\omega$

c) — satisfait: $\{rouge\}\{jaune\}\{vert\}\emptyset^\omega$
   — ne satisfait pas: $\{rouge\}\{jaune, vert\}\emptyset^\omega$

2.3)

a)

$$\begin{aligned}\mathsf{G}((&rouge \wedge \neg jaune \wedge \neg vert) \vee \\ &(\neg rouge \wedge jaune \wedge \neg vert) \vee \\ &(\neg rouge \wedge \neg jaune \wedge vert))\end{aligned}$$

b) $\mathsf{GF}\, rouge$

c) $\neg\mathsf{F}\,((rouge \wedge \neg vert) \wedge \mathsf{X}\,((\neg jaune)\ \mathsf{U}\ (\neg rouge \wedge vert)))$

2.4)

a) $\mathsf{G}(d \rightarrow \mathsf{F}r)$ ou $\mathsf{G}(d \rightarrow \mathsf{XF}r)$ (selon l'interpétation de « suivie »)

b) $\mathsf{FG}\, d$

c) $\mathsf{F}(d \wedge r)$

d) $\mathsf{GF}\, r$

2.5)

a)

$$\begin{aligned}
&\sigma \models \mathsf{F}(\varphi_1 \vee \varphi_2) \\
\iff &\exists j \geq 0 : \sigma[j..] \models (\varphi_1 \vee \varphi_2) && \text{(par déf. de F)} \\
\iff &\exists j \geq 0 : (\sigma[j..] \models \varphi_1) \vee (\sigma[j..] \models \varphi_2) && \text{(par déf. de } \vee\text{)} \\
\iff &(\exists j \geq 0 : \sigma[j..] \models \varphi_1) \vee (\exists j \geq 0 : \sigma[j..] \models \varphi_2) && \text{(par distributivité)} \\
\iff &(\sigma \models \mathsf{F}\varphi_1) \vee (\sigma \models \mathsf{F}\varphi_2) && \text{(par déf. de F)}
\end{aligned}$$

□

b)

$$\begin{aligned}
\mathsf{G}(\varphi_1 \wedge \varphi_2) &\equiv \neg\mathsf{F}\neg(\varphi_1 \wedge \varphi_2) && \text{(par déf. de G)}\\
&\equiv \neg\mathsf{F}(\neg\varphi_1 \vee \neg\varphi_2) && \text{(loi de De Morgan)}\\
&\equiv \neg((\mathsf{F}\neg\varphi_1) \vee (\mathsf{F}\neg\varphi_2)) && \text{(par exercice préc.)}\\
&\equiv (\neg\mathsf{F}\neg\varphi_1) \wedge (\neg\mathsf{F}\neg\varphi_2) && \text{(loi de De Morgan)}\\
&\equiv (\mathsf{G}\neg\neg\varphi_1) \wedge (\mathsf{G}\neg\neg\varphi_2) && \text{(par dualité)}\\
&\equiv (\mathsf{G}\varphi_1) \wedge (\mathsf{G}\varphi_2) && \text{(double négation)}
\end{aligned}$$ □

2.6) $\varphi\ \mathsf{W}\ \psi \coloneqq (\varphi\ \mathsf{U}\ \psi) \vee (\mathsf{G}\varphi)$

2.7) $\varphi\ \mathsf{R}\ \psi \coloneqq (\psi\ \mathsf{U}\ (\psi \wedge \varphi)) \vee \mathsf{G}\psi$, ou alternativement $\varphi\ \mathsf{R}\ \psi \coloneqq \psi\ \mathsf{W}\ (\psi \wedge \varphi)$

2.8)

a) Oui, nous avons bien $(\varphi \vee \psi)\ \mathsf{U}\ \psi \equiv \varphi\ \mathsf{U}\ \psi$. La formule de droite implique celle de gauche puisqu'elle est syntaxiquement plus restrictive. De plus, celle de gauche implique celle de droite car en choisissant la première position qui satisfait ψ, toutes les positions précédentes satisfont forcément φ.
Plus formellement, soit $\sigma \models (\varphi \vee \psi)\ \mathsf{U}\ \psi$. Il existe $j \in \mathbb{N}$ tel que

$$(\forall 0 \leq i < j : \sigma[i..] \models \varphi \vee \psi) \wedge \sigma[j..] \models \psi. \qquad (*)$$

Considérons l'indice j minimal qui satisfait (*). Par minimalité, aucun suffixe avant la position j ne satisfait ψ. Ainsi, nous avons:

$$(\forall 0 \leq i < j : \sigma[i..] \models \varphi) \wedge \sigma[j..] \models \psi,$$

ce qui implie $\sigma \models \varphi\ \mathsf{U}\ \psi$.

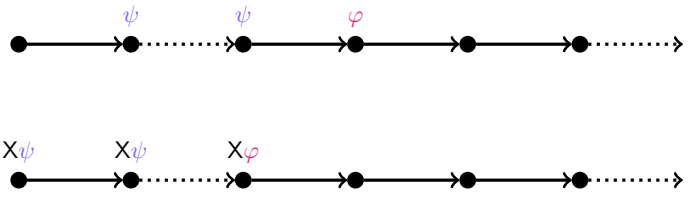
b) Non, par exemple, pour $\sigma \coloneqq \{p\}^\omega$:

$$\sigma \models p\ \mathsf{U}\ (p \vee q),$$
$$\sigma \not\models p\ \mathsf{U}\ q.$$

c) Non, par exemple, pour $\sigma \coloneqq \{p\}\emptyset^\omega$:

$$\sigma \models \neg(p\ \mathsf{U}\ q),$$
$$\sigma \not\models (\neg p)\ \mathsf{U}\ (\neg q).$$

d) Oui. Schématiquement, nous avons:



Plus formellement:

$$\begin{aligned}
\sigma \models \mathsf{X}(\varphi \,\mathsf{U}\, \psi) &\iff \sigma[1..] \models \varphi \,\mathsf{U}\, \psi \\
&\iff \exists j(\forall 0 \leq i < j : (\sigma[1..])[i..] \models \varphi) \wedge (\sigma[1..])[j..] \models \psi \\
&\iff \exists j(\forall 0 \leq i < j : (\sigma[i..])[1..] \models \varphi) \wedge (\sigma[j..])[1..] \models \psi \\
&\iff \exists j(\forall 0 \leq i < j : \sigma[i..] \models \mathsf{X}\varphi) \wedge (\sigma[j..]) \models \mathsf{X}\psi \\
&\iff \sigma \models (\mathsf{X}\varphi) \,\mathsf{U}\, (\mathsf{X}\psi).
\end{aligned}$$

2.9) Si $\sigma \models \mathsf{FG}(\varphi_1 \wedge \varphi_2)$, alors il existe un point à partir duquel tous les suffixes de σ satisfont $\varphi_1 \wedge \varphi_2$, et par conséquent $\sigma \models \mathsf{FG}\varphi_1$ et $\sigma \models \mathsf{FG}\varphi_2$.

L'autre direction est moins évidente. Supposons que $\sigma \models \mathsf{FG}\varphi_1$ et $\sigma \models \mathsf{FG}\varphi_2$. Cela signifie qu'il existe un point i à partir duquel tous les suffixes de σ satisfont φ_1, et un autre point j à partir duquel tous les suffixes de σ satisfont φ_2. Par conséquent, à partir du point $\max(i, j)$, tous les suffixes de σ satisfont à la fois φ_1 *et* φ_2.

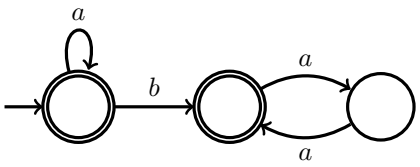
2.10)

	$\mathsf{GF}p \rightarrow \mathsf{FG}(q \vee r)$	$(r \,\mathsf{U}\, \mathsf{X}p) \,\mathsf{U}\, (q \wedge \neg\mathsf{XX}s)$
$\emptyset^\omega$	oui	non
$\{p,q,r,s\}^\omega$	oui	non
$\{p,q\}^\omega$	oui	oui
$\{r\}\emptyset\{p,q,s\}^\omega$	oui	non
$\{r\}\emptyset(\{p,q\}\{r,s\})^\omega$	oui	oui
$\{r\}\emptyset\{p\}\{q,r\}(\{p,s\}\emptyset)^\omega$	non	non

Chapitre 3

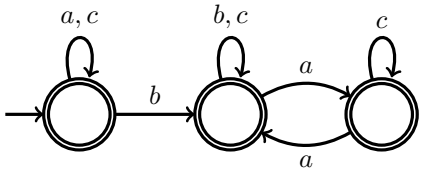
3.1)

a) $\overbrace{a^\omega}^{\text{sans } b} + \overbrace{a^*ba^\omega}^{\text{un seul } b} + \overbrace{a^*(b(aa)^*)^*a^\omega}^{\text{plusieurs } b} + \overbrace{a^*(b(aa)^*)^\omega}^{\text{infinité de } b}$ ou $a^*(b+aa)^\omega$

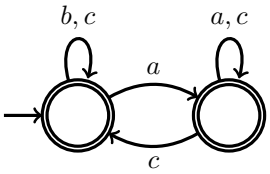


b)

$$\overbrace{(a+c)^\omega}^{\text{sans } b} + \overbrace{(a+c)^*b(a+c)^\omega}^{\text{un seul } b} + \underbrace{(a+c)^*(b(c+ac^*ac^*))^*(a+c)^\omega}_{\text{plusieurs } b} + \underbrace{(a+c)^*(b(c+ac^*ac^*))^\omega}_{\text{infinité de } b}$$



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



3.2) a) Oui, les deux décrivent $\{\sigma \in \{a,b\}^\omega : \sigma$ contient une infinité de a et $b\}$

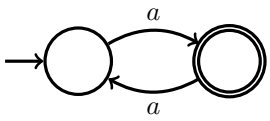
b) Non, la première expression reconnaît b^ω et pas l'autre.

Remarque.

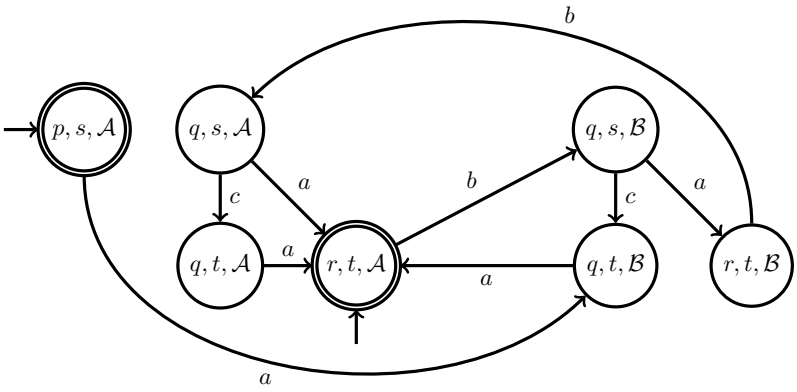
Si vous connaissez la théorie de la complexité, sachez que le problème qui consiste à déterminer si deux expressions ω-régulières décrivent le même langage est EXPSPACE-complet.

3.5) En les mettant simplement l'un à côté de l'autre, puisque plusieurs états initiaux sont permis.

3.6) Non, par exemple l'automate ci-dessous accepte a^ω. En inversant son état acceptant et son état non acceptant, l'automate accepte le même langage.



3.7)



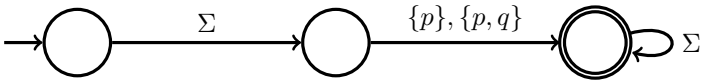
3.8) Oui, il suffit d'ajouter un nouvel état q_0, d'en faire l'unique état initial, et d'ajouter les transitions $\delta(q_0, a) \coloneqq \bigcup_{q \in Q_0} \delta(q, a)$.

3.9) Non, par exemple cela est impossible pour le langage $a^\omega + b^\omega$.

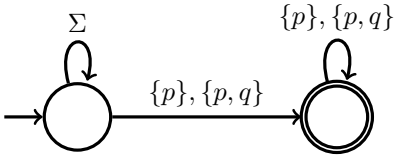
Chapitre 4

4.1) Posons $\Sigma \coloneqq 2^{AP}$.

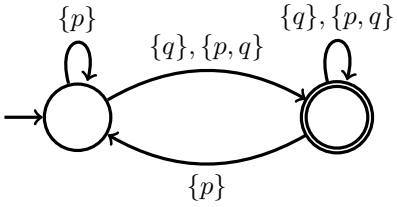
a) $\emptyset(\{p\} + \{p,q\})^\omega$



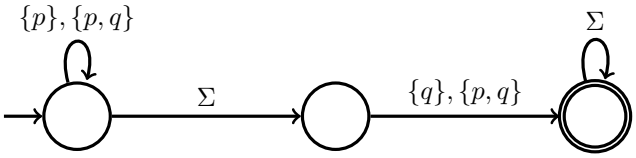
b) $\Sigma^*(\{p\} + \{p,q\})^\omega$



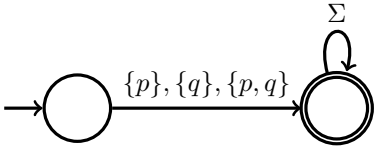
c) $(\{p\}^*(\{q\} + \{p,q\}))^\omega$



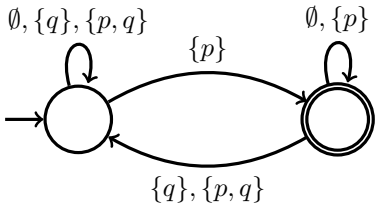
d) $(\{p\} + \{p,q\})^*\Sigma(\{q\} + \{p,q\})\Sigma^\omega$



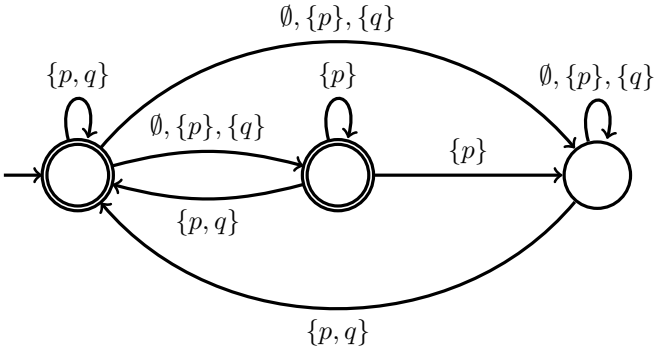
e) $(\{p\} + \{q\} + \{p,q\})\Sigma^\omega$



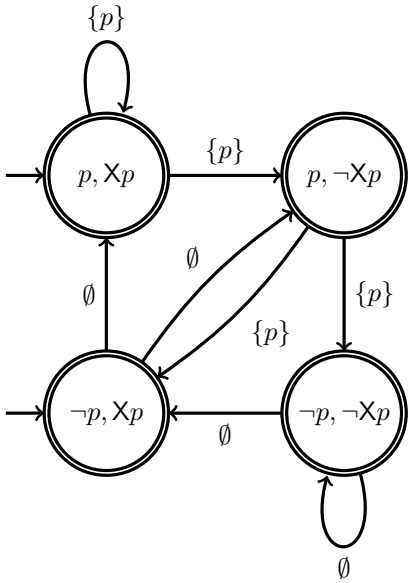
f) $(\emptyset + \{q\} + \{p,q\} + [\{p\}\Sigma^*(\{q\} + \{p,q\})])^\omega$



g) $[(\emptyset + \{p\} + \{q\})^* \{p, q\}]^\omega + [(\emptyset + \{p\} + \{q\})^* \{p, q\}]^* \, \Sigma \{p\}^\omega$



4.4)



4.5) On peut utiliser la même idée que pour l'exercice 3.10). Afin d'obtenir une contradiction, supposons qu'il existe un automate de Büchi déterministe

$\mathcal{A} = (Q, \Sigma, \delta, \{q_0\}, F)$ tel que $\mathcal{L}(\mathcal{A}) = [\![\mathsf{FG}p]\!]$. Remarquons que $\emptyset\{p\}^\omega \in \mathcal{L}(\mathcal{A})$. Il existe donc $n_1 \in \mathbb{N}$ tel que $\emptyset\{p\}^{n_1}$ atteint un état acceptant $q_1 \in F$ à partir de q_0. Le mot $\emptyset\{p\}^{n_1}\emptyset\{p\}^\omega$ est aussi accepté par $\mathcal{A}$. Ainsi, il existe $n_2 \in \mathbb{N}$ tel que $\emptyset\{p\}^{n_1}\emptyset\{p\}^{n_2}$ atteint un état acceptant $q_2 \in F$ à partir de q_0. Puisque l'automate est déterministe, le préfixe commun à ces deux mots traverse les mêmes états. Ainsi, nous avons:

$$q_0 \xrightarrow{\emptyset\{p\}^{n_1}} q_1 \xrightarrow{\emptyset\{p\}^{n_2}} q_2.$$

En répétant ce processus infiniment, nous obtenons des tailles $n_1, n_2, \ldots \in \mathbb{N}$ et des états $q_1, q_2, \ldots \in F$ tels que

$$q_0 \xrightarrow{\emptyset\{p\}^{n_1}} q_1 \xrightarrow{\emptyset\{p\}^{n_2}} q_2 \xrightarrow{\emptyset\{p\}^{n_3}} \cdots$$

Par conséquent, le mot $\sigma \coloneqq \prod_{i\geq 1} \emptyset\{p\}^{n_i}$ est accepté par $\mathcal{A}$ puisqu'il visite F infiniment souvent. Cela contradit $\sigma \not\models \mathsf{FG}p$. □

Chapitre 5

5.1)

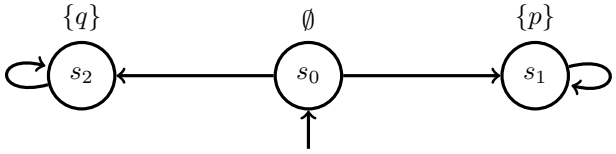
a) Oui, en prenant une trace de la forme $\{p\}\{q\}\cdots$.

b) Oui. De s_0 à s_2 on obtient $\{p\}\{q\}$ qui satisfait $\exists p \,\mathsf{U}\, \neg p$. L'état s_2 satisfait q, donc toute trace à partir de s_2 satisfait trivialement $\neg p \,\mathsf{U}\, q$.

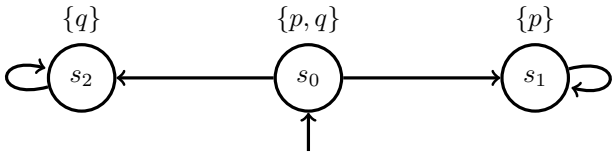
c) Non, la trace $(\{p\}\{p,q\})^\omega$ ne satisfait pas $p \,\mathsf{U}\, \neg p$, donc ce n'est pas nécessaire d'explorer le côté droit de la conjonction.

d) Non, cette propriété est plus forte que la précédente.

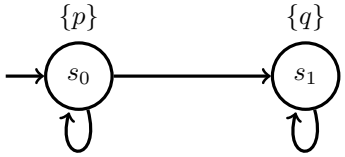
5.2) a) Cette structure de Kripke satisfait $\forall\mathsf{F}(p \vee q)$, mais pas $(\forall\mathsf{F}p) \vee (\forall\mathsf{F}q)$:



b) Cette structure de Kripke satisfait $(\exists\mathsf{G}p) \wedge (\exists\mathsf{G}q)$, mais pas $\exists\mathsf{G}(p \wedge q)$:

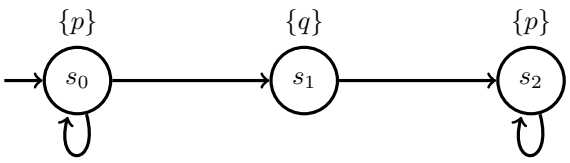


c) Cette structure satisfait $\forall((\neg p) \,\mathsf{U}\, (\neg q))$, mais pas $\neg\exists(p \,\mathsf{U}\, q)$:



5.4)

a) Cette structure de Kripke satisfait $\mathsf{FG}p$ mais pas $\forall\mathsf{F}\forall\mathsf{G}p$:



b) La structure de Kripke ci-dessus satisfait $\mathsf{FX}p$ mais pas $\forall\mathsf{F}\forall\mathsf{X}p$.

5.5) Soit $\mathcal{T}$ une structure de Kripke.

$\Rightarrow$) Supposons que $\mathcal{T} \models \forall\mathsf{G}\forall\mathsf{F}p$. Soit $s_0 s_1 \cdots$ une exécution infinie de $\mathcal{T}$. Nous devons montrer que trace$(s_0 s_1 \cdots) \models \mathsf{GF}p$, ou, autrement dit, que pour tout $j \in \mathbb{N}$, il existe $i \geq j$ tel que $p \in L(s_i)$. Soit $j \in \mathbb{N}$. Puisque $s_0 \models \forall\mathsf{G}\forall\mathsf{F}p$, en particulier trace$(s_0 s_1 \cdots) \models \mathsf{G}\forall\mathsf{F}p$. Ainsi, $s_j \models \forall\mathsf{F}p$. En particulier, cela implique que trace$(s_j s_{j+1} \cdots) \models \mathsf{F}p$. Il existe donc $i \geq j$ tel que $p \in L(s_i)$.

$\Leftarrow$) Supposons que $\mathcal{T} \models \mathsf{GF}p$. Soit s_0 un état initial de $\mathcal{T}$. Nous devons montrer que $s_0 \models \forall\mathsf{G}\forall\mathsf{F}p$, ou, autrement dit, que $s_j \models \forall\mathsf{F}p$ pour toute exécution $s_0 s_1 \cdots$ et tout $j \geq 0$. Soit $s_0 s_1 \cdots$ une exécution et soit $j \geq 0$. Considérons $\sigma \coloneqq s_j s_{j+1} \cdots$ un chemin infini de $\mathcal{T}$. Nous devons montrer que trace$(\sigma) \models \mathsf{F}p$. Par hypothèse, trace$(s_0 s_1 \cdots) \models \mathsf{GF}p$. Ainsi, p est satisfaite infiniment souvent sur cette trace. En particulier, il existe donc $i \geq j$ tel que $p \in L(s_i)$. Par conséquent, trace$(\sigma) \models \mathsf{F}p$. □

5.6)

a) Oui.

b) Non.

c) Oui.

Chapitre 6

6.1) Afin d'illustrer les étapes de calcul, ces solutions utilisent des ensembles d'états comme propositions atomiques (par abus de notation):

a) $\exists \mathsf{F}\{s_4, s_5\} \equiv \{s_4, s_5, s_6, s_7\}$

b) $\exists \mathsf{G}\{s_0, s_1, s_2, s_4\} \equiv \{s_0, s_2, s_4\}$

c)

$$\begin{aligned}
\forall \mathsf{X}(\neg\Phi_1 \vee \neg\Phi_2) &\equiv \neg\exists \mathsf{X}(\Phi_1 \wedge \Phi_2) \\
&\equiv \neg\exists \mathsf{X}(\{s_4, s_5, s_6, s_7\} \wedge \{s_0, s_2, s_4\}) \\
&\equiv \neg\exists \mathsf{X}\{s_4\} \\
&\equiv \neg\{s_6\} \\
&\equiv S \setminus \{s_6\}.
\end{aligned}$$

6.2) La réécriture de $\forall(\Phi_1 \ \mathsf{U} \ \Phi_2)$ copie trois fois Φ_2. Donc, si on compose des occurrences de $\forall\mathsf{U}$ à droite, la réécriture explosera exponentiellement.

Par exemple, posons $\Phi_0 \coloneqq q$ et $\Phi_n \coloneqq \forall(p \ \mathsf{U} \ \Phi_{n-1})$. La formule Φ_n possède n quantificateurs. Soit $f(n)$ le nombre de quantificateurs dans la forme normale existentielle de Φ_n. Par définition, nous avons:

$$f(n) = \begin{cases} 0 & \text{si } n = 0, \\ 3 \cdot f(n-1) + 2 & \text{sinon.} \end{cases}$$

Ainsi:

$$\begin{aligned}
f(n) &= 3 \cdot f(n-1) + 2 \\
&= 3[3 \cdot f(n-2) + 2] + 2 \\
&= 3^2 \cdot f(n-2) + 8 \\
&= 3^2 \cdot [3 \cdot f(n-3) + 2] + 8 \\
&= 3^3 \cdot f(n-3) + 26 \\
&= 3^3 \cdot f(n-3) + (3^3 - 1) \\
&\vdots \\
&= 3^i \cdot f(n-i) + (3^i - 1) \\
&\vdots \\
&= 3^n \cdot f(n-n) + (3^n - 1) \\
&= 3^n - 1.
\end{aligned}$$

Par conséquent, le nombre de quantificateurs de la forme normale appartient à $\Theta(3^n)$.

Remarquons aussi que l'arbre syntaxique de Φ_n a $2n+1$ sommets. Soit $g(n)$ le nombre de sommets de l'arbre syntaxique de Φ_n après sa mise en forme normale. Nous avons:

$$g(n) = \begin{cases} 1 & \text{si } n = 0, \\ (7 + 2 \cdot g(n-1)) + (3 + g(n-1)) = 10 + 3 \cdot g(n-1) & \text{sinon.} \end{cases}$$

En résolvant cette récurrence linéaire non homogène, on conclut qu'après la mise en forme normale, l'arbre syntaxique possède $g(n) = 6 \cdot 3^n - 5 \in \Theta(3^n)$ sommets.

6.3) On obtient des algorithmes en utilisant ces caractérisations:

- $[\![\exists \mathsf{F} \Phi]\!] = \{s \in S : \exists s' \in [\![\Phi]\!] \text{ t.q. } s \xrightarrow{*} s'\}$
- $[\![\forall \mathsf{F} \Phi]\!]$ est le plus petit ensemble $T \subseteq S$ tel que $[\![\Phi]\!] \subseteq S$ et $(\mathsf{Post}(s) \subseteq T \implies s \in T)$
- $[\![\forall \mathsf{X} \Phi]\!] = \{s \in S : \mathsf{Post}(s) \subseteq [\![\Phi]\!]\}$
- $[\![\forall(\Phi_1 \ \mathsf{U} \ \Phi_2)]\!]$ est le plus petit ensemble $T \subseteq S$ tel que $[\![\Phi_2]\!] \subseteq T$ et $(s \in [\![\Phi_1]\!] \wedge \mathsf{Post}(s) \subseteq T) \implies s \in T$.

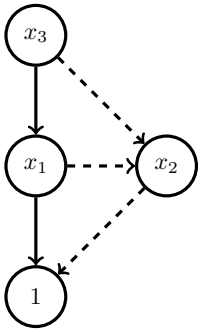
6.5) Voir `pile_avec_solutions.smv`.

6.7) a) On calcule d'abord le sous-graphe induit par Φ. Ensuite, on identifie ses composantes fortement connexes. Un état satisfait $\exists_{\text{équit}} \mathsf{G} \Phi$ ssi il peut atteindre une composante fortement connexe C telle que $\bigwedge_{1 \leq i \leq n} (C \cap S_i \neq \emptyset)$.

b) $\exists_{\text{équit}}(\Phi_1 \ \mathsf{U} \ \Phi_2) \equiv \exists(\Phi_1 \ \mathsf{U} \ (\Phi_2 \wedge \exists_{\text{équit}} \mathsf{G} \ \mathit{vrai}))$

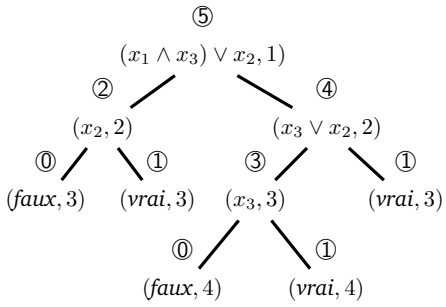
Chapitre 7

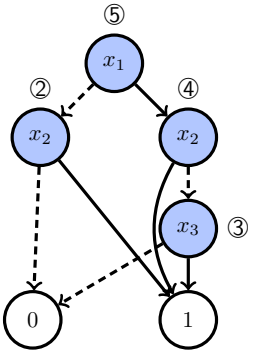
7.1) Par exemple, le BDD ci-dessous calcule $(x_1 \vee \neg x_2) \wedge (\neg x_2 \vee x_3)$ sous l'ordre $x_3 < x_1 < x_2$. Ce BDD possède trois sommets internes, alors que celui de la figure 7.2 en possède quatre.



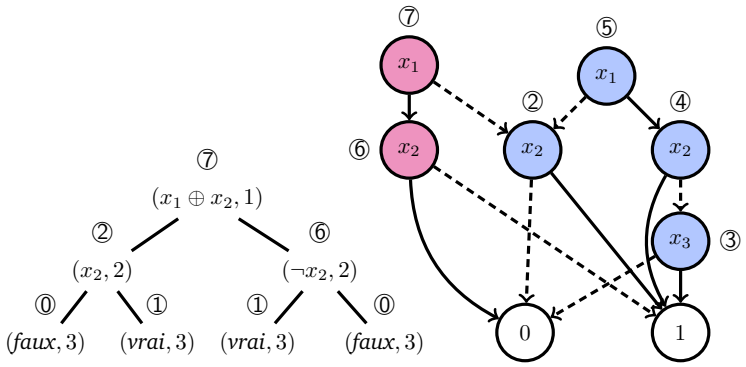
7.2) Posons $\varphi_n \coloneqq (x_1 \leftrightarrow y_1) \wedge \cdots \wedge (x_n \leftrightarrow y_n)$. Sous l'ordre $x_1 < y_1 < \cdots < x_n < y_n$, φ_n a un BDD de $3n$ sommets internes, alors que sous l'ordre $x_1 < \cdots < x_n < y_1 < \cdots < y_n$, φ_n a un BDD de taille $\Omega(2^n)$.

7.3) a) En exécutant `build`, nous obtenons:

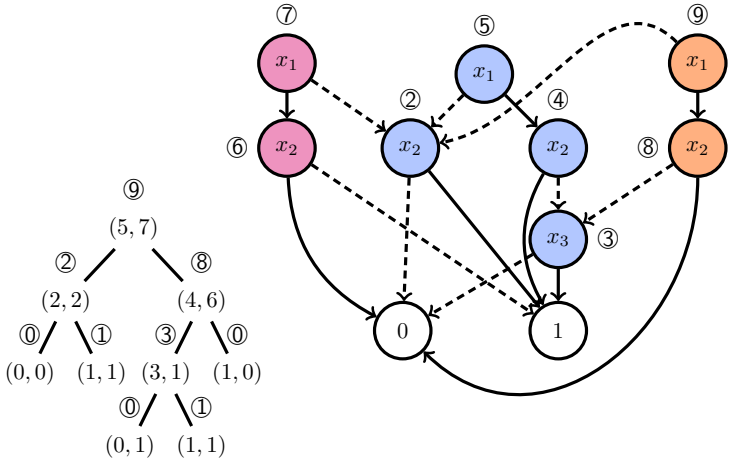




b) En exécutant `build`, nous obtenons:



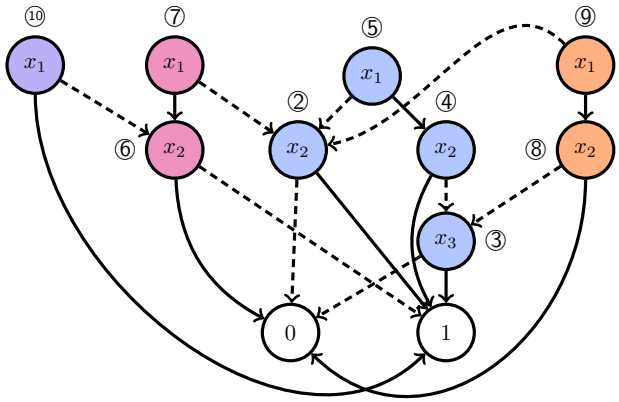
c) En exécutant $\texttt{apply}_{\wedge}(5, 7)$, nous obtenons:



d) Le résultat pourrait être obtenu à l'aide de $\texttt{exists}(5, 3)$. Procédons manuellement. Comme $f = (x_1 \vee \neg x_2) \wedge (\neg x_2 \vee x_3)$, nous avons:

$$\begin{aligned}
\exists x_3 : f &= [(x_1 \vee \neg x_2) \wedge (\neg x_2 \vee 0)] \vee [(x_1 \vee \neg x_2) \wedge (\neg x_2 \vee 1)] \\
&= [(x_1 \vee \neg x_2) \wedge \neg x_2] \vee [x_1 \vee \neg x_2] \\
&= \neg x_2 \vee [x_1 \vee \neg x_2] \\
&= x_1 \vee \neg x_2.
\end{aligned}$$

Cette fonction se représente par un sommet u tel que $T[u] = (x_1, 1, v)$ et $T[v] = (x_2, 0, 1)$. Un tel sommet $v = 6$ existe déjà. Aucun sommet tel que $T[u] = (x_1, 1, 6)$ n'existe. Il faut donc en ajouter un nouveau nommé 10. Nous obtenons donc:



7.4) a) Par le lemme 1, il suffit de tester si $u = v$.

b) Dans les trois algorithmes ci-dessous, les termes de la forme 0^k et 1^k dénotent la répétition de bits (par ex. $0^3 = 000$), et les termes de la forme 2^k dénotent l'exponentiation (par ex. $2^3 = 8$).

Entrées : sommet u
Sorties : $\{x \in \{0,1\}^n : f_u(x_1, \ldots, x_n) = \mathit{vrai}\}$

`sol(`u`):`
  `sol'(`u`,` i`):`
    si $u = 0$ **alors retourner** $\emptyset$
    sinon si $u = 1$ **alors retourner** $\{0,1\}^{n-i+1}$
    sinon
      $\ell \leftarrow \mathit{var}(u) - i$
      $X_0 \leftarrow$ `sol'(`$\mathit{lo}(u)$`,` $\mathit{var}(u) + 1$`)`
      $X_1 \leftarrow$ `sol'(`$\mathit{hi}(u)$`,` $\mathit{var}(u) + 1$`)`
      retourner $\{abc : a \in \{0,1\}^\ell, b \in \{0,1\}, c \in X_b\}$
  retourner `sol'(`u`,` 1`)`

---

c)

Entrées : sommet u
Sorties : $x \in \{0,1\}^n$ tel que $f_u(x_1, \ldots, x_n) = \mathit{vrai}$

```
sat(u):
    sat'(u, i):
        si u = 0 alors retourner ⊥
        sinon si u = 1 alors retourner 1^{n−i+1}
        sinon
            ℓ ← var(u) − i
            X_1 ← sat'(hi(u), var(u) + 1)
            si X_1 ≠ ∅ alors
                b ← 1
                c ← un élément de X_1
            sinon
                X_0 ← sat'(lo(u), var(u) + 1)
                b ← 0
                c ← un élément de X_0
            retourner 1^ℓ bc
    retourner sat'(u, 1)
```

d)

Entrées : sommet u
Sorties : $|\{x \in \{0,1\}^n : f_u(x_1, \ldots, x_n) = \mathit{vrai}\}|$

```
count(u):
    count'(u, i):
        si u = 0 alors retourner 0
        sinon si u = 1 alors retourner 2^{n−i+1}
        sinon
            ℓ ← var(u) − i
            c_0 ← count'(lo(u), var(u) + 1)
            c_1 ← count'(hi(u), var(u) + 1)
            retourner 2^ℓ · (c_0 + c_1)
    retourner count'(u, 1)
```

7.5) Nous expliquons le cas de ∃G. Soit $f\colon S \to S$ la fonction telle que $f(T) \coloneqq \mathrm{Pre}(T) \cap T$. Nous avons

$$[\![\exists \mathsf{G}]\!] = f(\cdots f(f([\![\Phi]\!])))$$

où f est appliquée jusqu'à l'atteinte d'un point fixe. Il suffit donc d'appliquer f symboliquement à répétition jusqu'à stabilisation. L'application de

f s'appuie sur le calcul de prédécesseurs (via $\rightarrow$ et `exists`) et de l'intersection (via $\texttt{apply}_\wedge$).

7.6) ★★ Supposons que la complexité de `build` soit polynomiale. Soit φ une expression booléenne. Afin de tester si φ est satisfaisable, nous construisons un BDD B pour φ enraciné en u. Comme `build` s'exécute en temps polynomial, B est forcément de taille polynomiale. Par le lemme 1, φ est satisfaisable ssi $u \neq 0$, ce qui se teste en temps constant. Ainsi, le problème SAT est soluble en temps polynomial, ce qui montre que $\mathsf{P} = \mathsf{NP}$. □

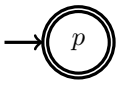
Chapitre 8

8.1)

a) Comme il n'y a pas de variables, le système n'a qu'un état p que nous n'avons pas à illustrer. Les règles sont comme suit:

$$\begin{array}{lcl} m_0 & \to & f_0 m_1 \\ m_1 & \to & \varepsilon \\ f_0 & \to & b_0 f_1 \mid f_1 \\ f_1 & \to & \varepsilon \\ b_0 & \to & f_0 b_1 \\ b_1 & \to & \varepsilon \mid b_2 \\ b_2 & \to & b_0 \end{array}$$

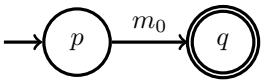
b) On construit un $\mathcal{P}$-automate $\mathcal{A}$ tel que *Conf*$(\mathcal{A}) = \{\langle p, \varepsilon\rangle\}$ (voir ci-dessous), on construit $\mathcal{B}$ tel que *Conf*$(\mathcal{B})$ = Pre*(*Conf*$(\mathcal{A})$), puis on vérifie si $\langle p, m_0\rangle \in$ *Conf*$(\mathcal{B})$.



Remarquons que cette vérification fonctionne même si le point d'entrée est appelé avec une pile non vide. En effet,

$$\langle p, m_0\rangle \xrightarrow{*} \langle p, \varepsilon\rangle \iff \langle p, m_0 w\rangle \xrightarrow{*} \langle p, w\rangle.$$

Alternativement, on peut construire $\mathcal{A}'$ tel que *Conf*$(\mathcal{A}') = \{\langle p, m_0\rangle\}$ (voir ci-dessous), construire $\mathcal{B}'$ tel que *Conf*$(\mathcal{B}')$ = Post*(*Conf*$(\mathcal{A})$), puis vérifier si $\langle p, \varepsilon\rangle \in$ *Conf*$(\mathcal{B}')$.



8.2) On construit un $\mathcal{P}$-automate $\mathcal{B}$ pour cet ensemble, puis on détermine si *Conf*$(\mathcal{B})$ est infini. Pour effectuer ce test, on peut (a) retirer les états non accessibles de $\mathcal{B}$, (b) retirer les états de $\mathcal{B}$ qui ne peuvent pas atteindre un état final, et (c) tester si l'automate résultant possède un cycle. Ces trois étapes s'implémentent en temps linéaire par rapport à la taille de $\mathcal{B}$.

8.3) Étiquetons chaque configuration dans l'état (x_i, y_j) avec la lettre a sur le dessus de la pile par la proposition atomique $p_{(x_i,y_j),a}$. Posons $\varphi_a \coloneqq \bigvee_{i,j\in\{0,1\}} p_{(x_i,y_j),a}$.

a) $\mathsf{GF}\varphi_{f_0}$

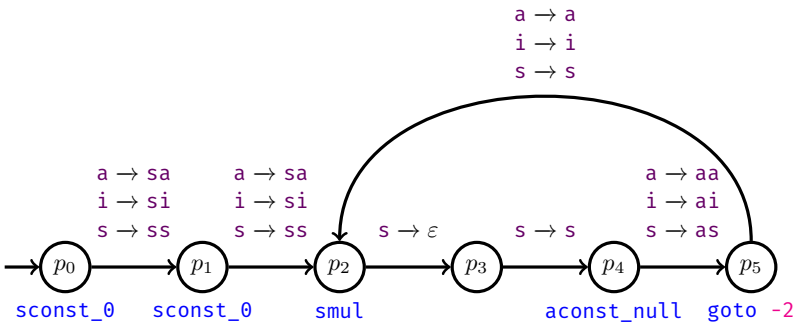
b) $\mathsf{F}\varphi_{\varepsilon}$ (en supposant que les configurations terminales bouclent sur elles-mêmes afin que les exécutions soient infinies)

c) $\mathsf{G}(\varphi_{f_0} \to \mathsf{F}\varphi_{m_0})$

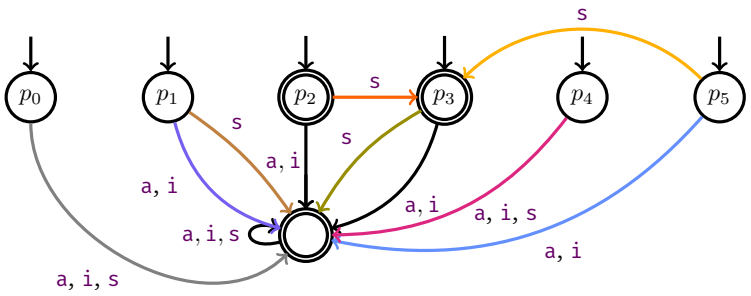
d) $\mathsf{FG} \bigvee_{a \in \Gamma} (p_{(x_0,y_0),a} \wedge p_{(x_1,y_1),a})$

8.4) Pour chaque état initial p_0 qui possède des transitions entrantes, il suffit de copier p_0 par un état p_0', de rediriger les transitions qui entrent dans p_0 vers p_0', et de copier les transitions sortantes de p_0 à p_0'. Le nombre d'états et transitions ajoutés est linéaire.

8.5) Rappelons d'abord $\mathcal{P}$:

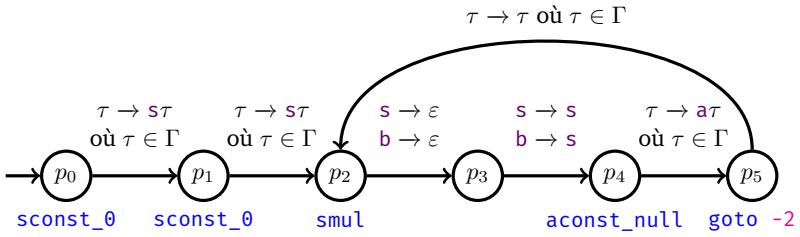


Nous obtenons:



Le programme mène donc à une erreur, peu importe le type initialement sur la pile, car p_0 mène directement à l'état final en lisant une lettre. Ce serait aussi le cas si la pile était vide, mais nous ne l'avons pas modélisé.

8.6) On ajoute simplement une nouvelle règle entre p_2 et p_3, et p_3 et p_4:



8.8) ★★

Proposition 17. *Conf*$(\mathcal{B}_i) \supseteq \text{Pre}^i(\textit{Conf}(\mathcal{A}))$ *pour tout* $i \in \mathbb{N}$.

Démonstration. Nous procédons par induction sur $i \in \mathbb{N}$. La validité du cas de base découle de

$$\textit{Conf}(\mathcal{B}_0) = \textit{Conf}(\mathcal{A}) = \text{Pre}^0(\textit{Conf}(\mathcal{A})).$$

Soient $i > 0$ et $\langle p, w\rangle \in \text{Pre}^i(\textit{Conf}(\mathcal{A}))$. Par définition, nous avons

$$p \xrightarrow{a \to u}_{\mathcal{P}} p' \text{ où } w = av,\ w' = uv,\ v \in \Gamma^* \text{ et } \langle p', w'\rangle \in \text{Pre}^{i-1}(\textit{Conf}(\mathcal{A})).$$

Par hypothèse d'induction, $\langle p', w'\rangle \in \textit{Conf}(\mathcal{B}_{i-1})$. Ainsi,

$$p' \xrightarrow{u}_{\mathcal{B}_{i-1}} p'' \xrightarrow{v}_{\mathcal{B}_{i-1}} p_f \text{ où } p' \in P,\ p'' \in Q \text{ et } p_f \in F.$$

Par les applications de la règle d'ajout qui mène de $\mathcal{B}_{i-1}$ à $\mathcal{B}_i$, ce dernier possède la transition (p, a, p''). Comme $\mathcal{B}_i$ contient toutes les transitions de $\mathcal{B}_{i-1}$, nous obtenons

$$p \xrightarrow{a}_{\mathcal{B}_i} p'' \xrightarrow{v}_{\mathcal{B}_i} p_f,$$

ce qui implique $\langle p, w\rangle = \langle p, av\rangle \in \textit{Conf}(\mathcal{B}_i)$. □

Proposition 18. $\text{Pre}^*(\textit{Conf}(\mathcal{A})) \supseteq \textit{Conf}(\mathcal{B}_i)$ *pour tout* $i \in \mathbb{N}$.

Démonstration. Nous montrons une propriété plus forte:

Si $p \in P$ et $p \xrightarrow{w}_{\mathcal{B}_i} q$, alors $\exists\, p' \xrightarrow{w'}_{\mathcal{A}} q$ et $\langle p, w\rangle \xrightarrow{*}_{\mathcal{P}} \langle p', w'\rangle$.

Procédons par double induction sur $i \in \mathbb{N}$, puis $|w|$. Le cas où $i = 0$ découle immédiatement de $\mathcal{B}_0 = \mathcal{A}$, et le cas où $w = \varepsilon$ est trivialement vrai. Supposons que $i > 0$ et $|w| > 0$. Si $p \xrightarrow{w}_{\mathcal{B}_{i-1}} q$, alors nous avons terminé par hypothèse d'induction. Sinon, $w = uav$, où $a \in \Gamma$, $u, v \in \Gamma^*$,

$$p \xrightarrow{u}_{\mathcal{B}_{i-1}} r \xrightarrow{a}_{\mathcal{B}_i} r' \xrightarrow{v}_{\mathcal{B}_i} q,$$

et $\mathcal{B}_{i-1}$ ne contient pas la transition (r, a, r'). Autrement dit, cette transition provient de la règle d'ajout. En particulier, cela implique que $r \in P$. Par hypothèses d'induction, nous avons:

— $\exists\, p'' \xrightarrow{w''}_{\mathcal{A}} r$ et $\langle p, u\rangle \xrightarrow{*}_{\mathcal{P}} \langle p'', w''\rangle$, et

— $\exists\, p' \xrightarrow{w'}_{\mathcal{A}} q$ et $\langle r, av\rangle \xrightarrow{*}_{\mathcal{P}} \langle p', w'\rangle$.

Comme $r \in P$ et aucune transition n'entre dans un état initial de $\mathcal{A}$, nous avons forcément $w'' = \varepsilon$ et nous pouvons donc simplifier par:

— $\langle p, u\rangle \xrightarrow{*}_{\mathcal{P}} \langle r, \varepsilon\rangle$, et

— $p' \xrightarrow{w'}_{\mathcal{A}} q$ et $\langle r, av\rangle \xrightarrow{*}_{\mathcal{P}} \langle p', w'\rangle$.

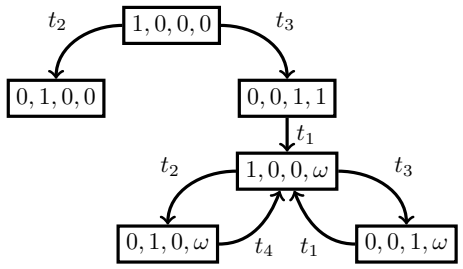
Par conséquent:

$$\langle p, w\rangle = \langle p, uav\rangle \xrightarrow{*}_{\mathcal{P}} \langle r, av\rangle \xrightarrow{*}_{\mathcal{P}} \langle p', w'\rangle.$$

Ainsi, $p' \xrightarrow{w'}_{\mathcal{A}} q$ et $\langle p, w\rangle \xrightarrow{*}_{\mathcal{P}} \langle p', w'\rangle$ comme désiré. □

Chapitre 9

9.1)



9.2)

Itér.	**Base** B	**Prédecesseurs**
0	$\{(0,1,1)\}$	$(0,1,1)_{t_1} = (0,1,1)$ $(0,1,1)_{t_2} = (1,2,0)$ $(0,1,1)_{t_3} = (0,0,2)$
1	$\{(0,1,1),(1,2,0),(0,0,2)\}$	$(1,2,0)_{t_1} = (0,2,0)$ $\quad (0,0,2)_{t_1} = (0,1,2)$ $(1,2,0)_{t_2} = (2,3,0)$ $\quad (0,0,2)_{t_2} = (1,1,1)$ $(1,2,0)_{t_3} = (1,1,1)$ $\quad (0,0,2)_{t_3} = (0,0,3)$
2	$\{(0,1,1),(0,2,0),(0,0,2)\}$	$(0,2,0)_{t_1} = (0,2,0)$ $(0,2,0)_{t_2} = (1,3,0)$ $(0,2,0)_{t_3} = (0,1,1)$
3	$\{(0,1,1),(0,2,0),(0,0,2)\}$	base inchangée

9.3) Un déclenchement en profondeur de t_2, t_3 et t_1 mène à cinq sommets. Un déclenchement de t_2, t_1 et t_3 mène à quatre sommets.

9.4) Comme $\boldsymbol{k} \geq \boldsymbol{m}$, il existe $\boldsymbol{d} \in \mathbb{N}^P$ tel que $\boldsymbol{k} = \boldsymbol{m} + \boldsymbol{d}$. Nous avons $\boldsymbol{k}(p) \geq \boldsymbol{m}(p) \geq F(p,t)$ pour toute place p, ainsi t est déclenchable en $\boldsymbol{k}$. Posons $\boldsymbol{k}' \coloneqq \boldsymbol{m}' + \boldsymbol{d}$. Nous avons:

$$\boldsymbol{k} = (\boldsymbol{m} + \boldsymbol{d}) \xrightarrow{t} (\boldsymbol{m}' + \boldsymbol{d}) = \boldsymbol{k}' \geq \boldsymbol{m}'. \qquad \square$$

9.6) Voir l'annexe A de [BHO20].

Chapitre 10

10.1)

a) Nous avons:

$$\begin{aligned}
&\mathbb{P}(s_0 \models \mathsf{F} s_2) \\
&= \sum_{i=0}^{\infty}((1/2)\cdot(1/4))^i \cdot (1/2) + (1/2)\cdot\sum_{i=0}^{\infty}((1/4)\cdot(1/2))^i \cdot (1/2) \\
&= (1/2)\cdot\sum_{i=0}^{\infty}(1/8)^i + (1/4)\cdot\sum_{i=0}^{\infty}(1/8)^i \\
&= (1/2)\cdot(1/(1-1/8)) + (1/4)\cdot(1/(1-1/8)) \\
&= (3/4)\cdot(8/7) \\
&= 6/7.
\end{aligned}$$

b) Posons $A \coloneqq [\![p]\!] = \{s_0, s_2, s_4, s_5, s_7\}$ et $A' \coloneqq \{s_4, s_5, s_6, s_7\}$. Nous avons:

$$\begin{aligned}
\mathbb{P}(\mathsf{GF}p) &= \mathbb{P}(\mathsf{GF}A) \\
&= \mathbb{P}(\mathsf{F}A') \\
&= (3/4)\cdot\mathbb{P}(s_0 \models \mathsf{F}A') + (1/4)\cdot\mathbb{P}(s_5 \models \mathsf{F}A') \\
&= (3/4)\cdot\mathbb{P}(s_0 \models \mathsf{F}A') + (1/4) \\
&= (3/4)\cdot\boldsymbol{x}(s_0) + (1/4).
\end{aligned}$$

Il nous reste à déterminer $\mathbf{A}$ et $\boldsymbol{b}$ afin d'identifier $\boldsymbol{x}$. Nous avons $S_0 = \emptyset$, $S_1 = \{s_4, s_5, s_6, s_7\}$ et $S_? = \{s_0, s_1, s_2, s_3\}$. Ainsi:

$$\mathbf{A} \coloneqq \begin{pmatrix} 0 & 1/2 & 1/2 & 0 \\ 1/4 & 0 & 1/2 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \text{ et } \boldsymbol{b} \coloneqq \begin{pmatrix} 0 \\ 1/4 \\ 0 \\ 1 \end{pmatrix}.$$

En résolvant $(\mathbf{I} - \mathbf{A}) \cdot \boldsymbol{x} = \boldsymbol{b}$, nous obtenons $\boldsymbol{x} = (1, 1, 1, 1)$. En particulier, $\boldsymbol{x}(s_0) = 1$ et ainsi $\mathbb{P}(\mathsf{GF}p) = (3/4)\cdot 1 + (1/4) = 1$.

Alternativement, nous pouvons (bien plus) simplement invoquer le fait que A' correspond à l'ensemble de toutes les CFC terminales. Ainsi, $\mathbb{P}(s_i \models \mathsf{F}A') = 1$ pour tout état s_i.

c) Nous avons $S_0 = \{s_6, s_7\}$, $S_1 = \{s_1, s_3, s_4\}$ et $S_? = \{s_0, s_2, s_5\}$. Ainsi,

$$\mathbf{A} = \begin{pmatrix} 0 & 1/2 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \text{ et } \boldsymbol{b} = \begin{pmatrix} 1/2 \\ 1 \\ 1 \end{pmatrix}.$$

En résolvant $(\mathbf{I} - \mathbf{A}) \cdot \boldsymbol{x} = \boldsymbol{b}$, nous obtenons $\boldsymbol{x} = (1, 1, 1)$. Ainsi, $\mathbb{P}(p \,\mathsf{U}\, q) = \mathbf{init}(s_0)\cdot 1 + \mathbf{init}(s_5)\cdot 1 = 1$.

10.2)

a) $\bigwedge_{1\leq k\leq 6} \mathcal{P}_{=1/6}(\mathsf{F}\, p_k)$, où $AP \coloneqq \{p_k : 1 \leq k \leq 6\}$ et la proposition p_i est associée au $i^{\text{ème}}$ état du bas (à partir de la gauche).

b) Nous vérifions seulement $\mathcal{P}_{=1/6}(\mathsf{F}\, p_1)$; les autres formules sont identiques ou plus simples. Nous avons $S_1 = \{\texttt{001}\}$, $S_? = \{\texttt{---}, \texttt{0--}, \texttt{00-}\}$ et $S_0 = S \setminus (S_1 \cup S_?)$. Ainsi:

$$\mathbf{A} = \begin{pmatrix} 0 & 1/2 & 0 \\ 0 & 0 & 1/2 \\ 0 & 1/2 & 0 \end{pmatrix} \text{ et } \boldsymbol{b} = \begin{pmatrix} 0 \\ 0 \\ 1/2 \end{pmatrix}.$$

En résolvant $(\mathbf{I} - \mathbf{A}) \cdot \boldsymbol{x} = \boldsymbol{b}$, nous obtenons $\boldsymbol{x} = (1/6, 1/3, 2/3)$. En particulier, $\boldsymbol{x}(\texttt{---}) = 1/6$ et ainsi $\mathbb{P}(\mathsf{F} p_1) = \textbf{init}(\texttt{---}) \cdot (1/6) = 1 \cdot (1/6) = 1/6$ comme désiré.

Alternativement, nous obtenons le même résultat avec:

$$\begin{aligned} \mathbb{P}(\mathsf{F} p_1) &= (1/2) \cdot (1/2) \cdot \sum_{i=0}^{\infty} ((1/2) \cdot (1/2))^i \cdot (1/2) \\ &= (1/8) \cdot \sum_{i=0}^{\infty} (1/4)^i \\ &= (1/8) \cdot (1/(1 - 1/4)) \\ &= (1/8) \cdot (4/3) \\ &= 1/6. \end{aligned}$$

10.7)

a) $\mathcal{P}_{=1}(\mathsf{F}\, p_{\text{IP}})$

b) $\mathcal{P}_{\geq 1/2}((\neg p_{\text{IP}})\ \mathsf{U}^{\leq 2}\ p_{\text{IP}})$

c) $\mathcal{P}_{=0}(\mathsf{F}\, p_{\perp})$

d) $\mathcal{P}_{>0}(\mathsf{G}\, \mathcal{P}_{>0}(\mathsf{F}\, p_?))$

e) $\mathcal{P}_{=1}(\mathsf{G}(p_{\text{IP}} \to \mathcal{P}_{=1}(\mathsf{G}\, p_{\text{IP}})))$

Fiches récapitulatives

Les fiches des pages suivantes résument le contenu de chacun des chapitres. Elles peuvent être imprimées recto-verso, ou bien au recto seulement afin d'être découpées et pliées en deux. À l'ordinateur, il est possible de cliquer sur la plupart des puces « ▶ » pour accéder à la section du contenu correspondant.

1. Systèmes de transition

Systèmes de transition

- Modélise formellement un système concret
- *États*: ensemble S (sommets)
- *Relation de transition*: $\to \subseteq S \times S$ (arcs)
- *États initiaux*: $I \subseteq S$ (où S peut débuter)

Prédécesseurs et successeurs

- *Successeurs immédiats*: $\text{Post}(s_1) = \{s_2\}$
- *Prédécesseurs immédiats*: $\text{Pre}(s_1) = \{s_0, s_2\}$
- *Successeurs*: $\text{Post}^*(s_1) = \{s_1, s_2, s_3\}$
- *Prédécesseurs*: $\text{Pre}^*(s_1) = \{s_0, s_1, s_2\}$
- *États terminaux*: s_3 car $\text{Post}(s_3) = \emptyset$

Chemins et exécutions

- *Chemin fini*: $s_0 \to s_1 \to s_2 \to s_1$
- *Chemin infini*: $s_1 \to s_2 \to s_1 \to s_2 \to \cdots$
- *Ch. maximal*: ne peut être étendu, par ex. $s_0 \to s_1 \to s_2 \to s_3$
- *Exécution*: chemin maximal qui débute par un état initial

Structures de Kripke

- Décrit les propriétés des états d'un système
- *Système de transition* $(S, \to, I)$
- *Propositions atomiques AP*
- *Fonction d'étiquetage* $L\colon S \to 2^{AP}$
- *Exemple*: si $AP = \{p, q, r\}$ et $L(s_1) = \{p, q\}$, alors s_1 satisfait p et q, mais pas r

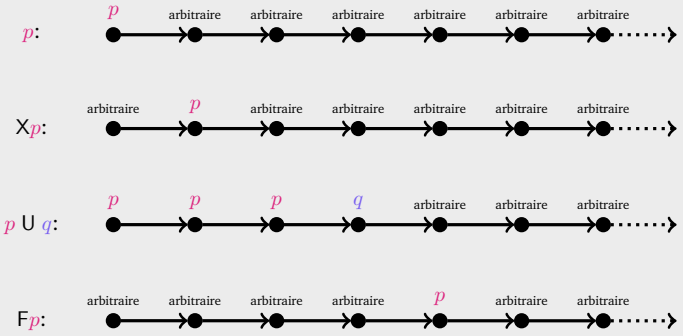
Explosion combinatoire

- Nombre d'états peut croître rapidement, par ex. $\mathcal{O}(2^n)$
- Existe techniques pour surmonter ce problème

2. Logique temporelle linéaire (LTL)

Logique

- *Intérêt*: spécifier formellement des propriétés
- *Syntaxe*: *vrai* | p | $\varphi \wedge \varphi$ | $\varphi \vee \varphi$ | $\neg\varphi$ | $\mathsf{X}\varphi$ | $\varphi\ \mathsf{U}\ \varphi$ | $\mathsf{F}\varphi$ | $\mathsf{G}\varphi$
- *Interprétation*: sur des traces, c.-à-d. des mots infinis de $(2^{AP})^\omega$
- *Sémantique*:



Équivalences

- *Distributivité*: X, G, U (gauche) sur $\wedge$; X, F, U (droite) sur $\vee$
- *Dualité*: X dual de lui-même, F dual de G
- *Autre*: seules combinaisons de F et G: {F, G, FG, GF}

Types de propriétés

- *Invariant*: propriété toujours vraie: $\mathsf{G}\varphi$
- *Sûreté*: réfutable avec préfixe fini
- *Vivacité*: comportements vers l'infini

Vérification

- *Trace*: états d'une exécution infinie $\mapsto$ leurs étiquettes
- *Satisfaisabilité*: $\mathcal{T} \models \varphi \iff \text{Traces}(\mathcal{T}) \subseteq [\![\varphi]\!]$
- *Équité*: omettre traces triviales où un processus est ignoré
- *En pratique*: Spin (avec Promela), par ex. algorithme de Lamport, protocole de Needham-Schroeder

3. Langages ω-réguliers

Expressions ω-régulières

- *Décrivent*: les langages ω-réguliers de mots infinis
- *Syntaxe*:

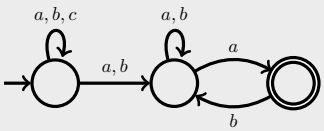
$$s ::= r^\omega \mid (r \cdot s) \mid (s + s)$$
$$r ::= r^* \mid (r \cdot r) \mid (r + r) \mid a \mid \varepsilon$$

- *Exemples*:

$a(a+b)^\omega$: mots qui débutent par a,
$(ab)^\omega$: l'unique mot $ababab\cdots$
$b^*(aa^*bb^*)^\omega$: mots avec une infinité de a *et* de b
$(a+b)^*b^\omega$: mots avec un nombre fini de a

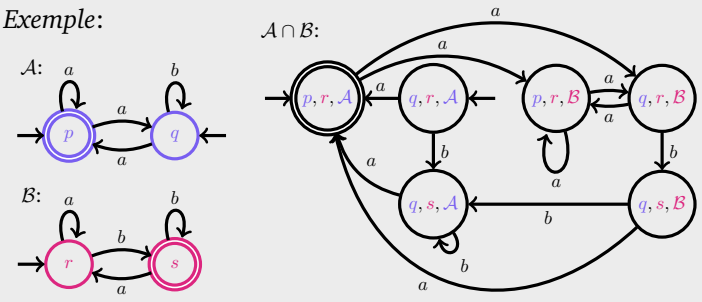
Automates de Büchi

- *Définition*: automates usuels; plusieurs états initiaux
- *Langage*: mots qui visitent états acceptants ∞ souvent
- *Expressivité*: $\equiv$ expressions ω-rég.; déterminisme $\neq$ non dét.
- *Exemple*: mots tels que $\#a = \infty$, $\#b = \infty$ et $\#c \neq \infty$:



Intersection d'automates de Büchi

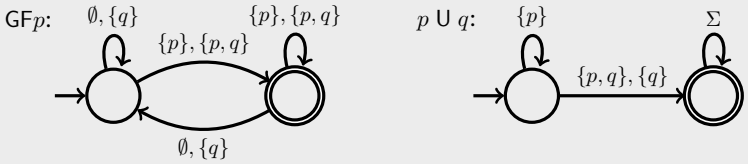
- $1^{ère}$ *idée*: simuler $\mathcal{A}$ et $\mathcal{B}$ en parallèle via produit; pas suffisant
- *Solution*: faire deux copies, alterner aux états acceptants
- *Exemple*:



4. Vérification algorithmique de formules LTL

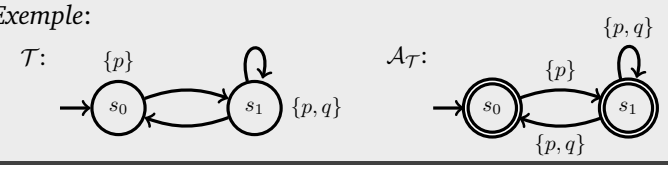
LTL vers automates

- *Alphabet*: $\Sigma := 2^{AP}$
- *Conversion*: $\varphi \to \mathcal{A}_\varphi$ (pire cas: $2^{\mathcal{O}(|\varphi|)}$ états)
- *Exemples*:



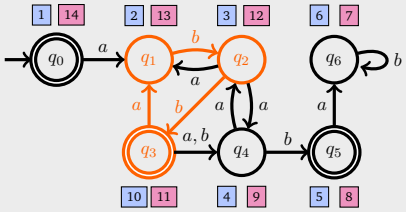
Structures de Kripke vers automates

- *Conversion*: étiquettes ≡ lettres + tout acceptant
- *Exemple*:

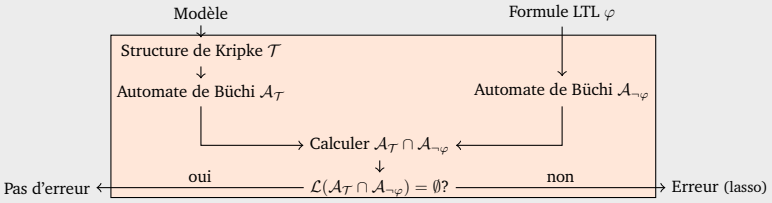


Test du vide

- *Vérification*: $\mathcal{T} \models \varphi \iff \mathcal{L}(\mathcal{A}_\mathcal{T}) \cap \mathcal{L}(\mathcal{A}_{\neg\varphi}) = \emptyset$
- *Lassos*: $\mathcal{L}(\mathcal{B}) \neq \emptyset \iff \exists q_0 \xrightarrow{*} q \xrightarrow{+} q$ où $q_0 \in Q_0$, $q \in F$
- *Détection*: double parcours en profondeur (temps linéaire)



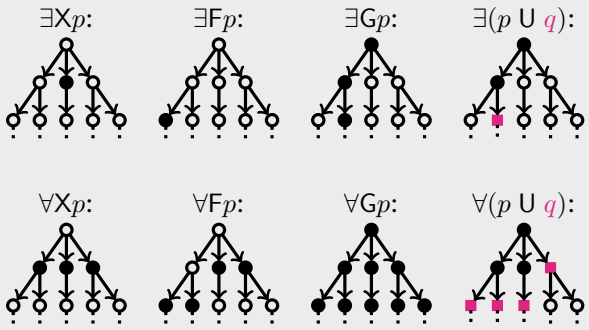
- *Sommaire*:



5. Logique temporelle arborescente (CTL)

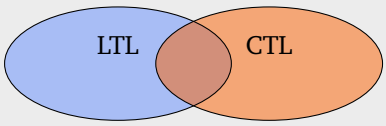
Logique

- *Intérêt*: raisonne sur le temps avec un futur indéterminé
- *Syntaxe*: *vrai* | p | $\Phi \wedge \Phi$ | $\Phi \vee \Phi$ | $\neg\Phi$ | $QT\Phi$ | $Q(\Phi$ U $\Phi)$
  où $Q \in \{\exists, \forall\}$, $T \in \{\mathsf{X}, \mathsf{F}, \mathsf{G}\}$
- *Interprétation*: sur l'arbre de calcul d'une structure de Kripke
- *Sémantique*:



Propriétés d'un système

- *Satisfiabilité dépend des états*: $[\![\Phi]\!] := \{s \in S : s \models \Phi\}$
- *Spécification*: $\mathcal{T} \models \Phi \iff I \subseteq [\![\Phi]\!]$
- *Expressivité*: incomparable à LTL



Équivalences

- *Distributivité*:

$$\exists\mathsf{F}(\Phi_1 \vee \Phi_2) \equiv (\exists\mathsf{F}\Phi_1) \vee (\exists\mathsf{F}\Phi_2)$$
$$\forall\mathsf{G}(\Phi_1 \wedge \Phi_2) \equiv (\forall\mathsf{G}\Phi_1) \wedge (\forall\mathsf{G}\Phi_2)$$

- *Attention*: pas équiv. si on change les quantificateurs
- *Dualité*: effet d'une négation: $\exists \leftrightarrow \forall$, X $\leftrightarrow$ X et F $\leftrightarrow$ G
- *Idempotence*: $QT\,QT\Phi \equiv QT\Phi$ où $Q \in \{\exists, \forall\}$, $T \in \{\mathsf{F}, \mathsf{G}\}$

6. Vérification algorithmique de formules CTL

Algorithme

- *Approche*: calculer $[\![\Phi']\!]$ pour chaque sous-formule Φ' de Φ
- *Vérification*: tester $I \subseteq [\![\Phi]\!]$
- *Forme normale existentielle* plus simple, mais pas nécessaire
- *Complexité*: $\mathcal{O}((|S| + |\to|) \cdot |\Phi|)$ avec bonne implémentation
- *En pratique*: NuSMV + langage de description de haut niveau

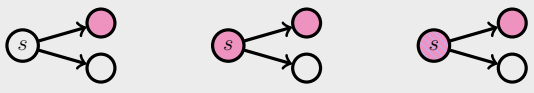
Logique propositionnelle

- *Règles récursives*:

$$[\![\mathit{vrai}]\!] = S,$$
$$[\![p]\!] = \{s \in S : p \in L(s)\},$$
$$[\![\Phi_1 \wedge \Phi_2]\!] = [\![\Phi_1]\!] \cap [\![\Phi_2]\!],$$
$$[\![\neg\Phi]\!] = S \setminus [\![\Phi]\!].$$

Opérateurs temporels existentiels

- *Calcul de* $[\![\exists\mathsf{X}\Phi]\!]$: $\{s \in S : \mathrm{Post}(s) \cap [\![\Phi]\!] \neq \emptyset\}$
- *Calcul de* $[\![\exists\mathsf{G}\Phi]\!]$: $T \subseteq [\![\Phi]\!]$ max. t.q. $\forall s \in T : \mathrm{Post}(s) \cap T \neq \emptyset$
- *Calcul de* $\exists(\Phi_1$ U $\Phi_2)$: $T \supseteq [\![\Phi_2]\!]$ min. t.q.
  $s \in [\![\Phi_1]\!] \wedge \mathrm{Post}(s) \cap T \neq \emptyset) \implies s \in T$



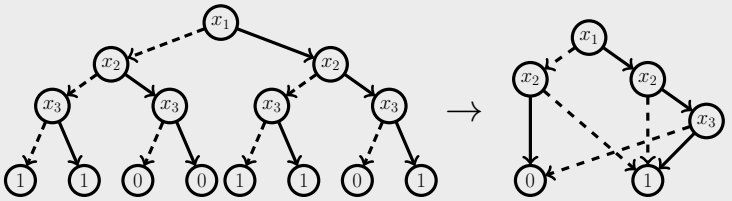
Optimisations

- *Autres opérateurs*: $\forall$ et F s'implémentent directement; nécessaire pour obtenir un algorithme polynomial
- *Points fixes*: temps linéaire si calculs directs sans raffinements itératifs; par ex. calcul de composantes fortement connexes

7. Vérification symbolique : diagrammes de décision binaire

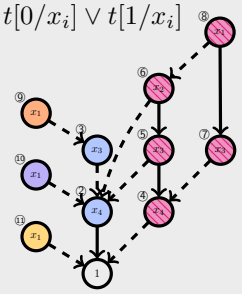
Diagramme de décision binaire

- *But*: représenter des fonctions booléennes de façon compacte
- *Utilité*: manipuler efficacement de grands ensembles d'états
- *Propriétés*:
  - graphe dirigé acyclique
  - sommets étiquetés par variables ordonnées sauf 0 et 1
  - les chemins respectent l'ordre des variables
  - sommets *uniques* et *non redondants* ($lo(u) \neq hi(u)$)
- *Canonicité*: pas deux BDDs pour la même fonction booléenne



Manipulation

- *Représentation*: tableau associatif *sommet* $\leftrightarrow (x_i, lo, hi)$
- *Ajout d'un sommet*: temps constant avec $\texttt{make}(x_i, lo, hi)$
- *Construction*: par substitutions récursives avec $\texttt{build}(\varphi)$
- *Op. bool.*: application récursive « synchronisée » avec $\texttt{apply}_{\bigcirc}$
- *Quantif.*: $\texttt{exists} x_i \in \{0,1\} : t$ obtenu via $t[0/x_i] \vee t[1/x_i]$
- *Complexité*: polynomiale sauf pour `build`



Vérification

- *État*: représenté par identifiant binaire
- *Transition*: paire d'identifiants binaires
- *Ensemble*: représenté par sommet de BDD
- *Logique prop.*: manipulation de BDD
- *Opérateurs temporels*: via calculs de Post ou Pre sur BDD
- *Satisfiabilité*: $I \subseteq [\![\Phi]\!] \Leftrightarrow I \cap \overline{[\![\Phi]\!]} = \emptyset \Leftrightarrow \texttt{apply}_{\wedge}(u_I, u_{\overline{[\![\Phi]\!]}}) = 0$

8. Systèmes avec récursion

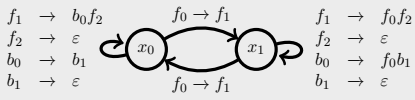
Contexte

- *Espace d'états*: pile d'appel ou d'éléments (+ valeurs locales)
- *Défi*: gérer un nombre infini ou arbitraire d'états
- *Approche*: construction et analyse de systèmes à pile

Systèmes à pile

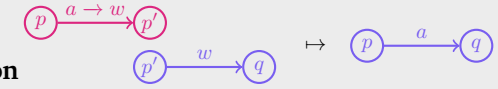
- *Définition*: états P, alphabet Γ, transitions $\{p \xrightarrow{a \to u} q, \ldots\}$
- *But*: décrire un ensemble de piles (et non accepter des mots)
- *Configuration*: $\langle p, w \rangle \in P \times \Gamma^* \mapsto$ (p) + pile $\begin{array}{|c|}\hline w_1 \\ \vdots \\ w_n \\ \hline \end{array}$
- *Exemple de modélisation*:

```
bool x ∈ {faux,vrai}

foo():                    bar():
f0:  x = ¬x:              b0:  si x: foo()
f1:  si x: foo()          b1:  retourner
     sinon: bar()
f2:  retourner
```



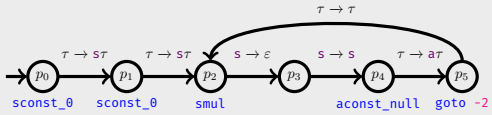
Calcul de prédécesseurs/successeurs

- *Déf.*: $\text{Pre}^*(C) := \bigcup_{i \geq 0} \text{Pre}^i(C)$ et $\text{Post}^*(C) := \bigcup_{i \geq 0} \text{Post}^i(C)$
- *Représentation*: symbolique de C avec un $\mathcal{P}$-automate $\mathcal{A}$
- *Idée*: (états initiaux = états de $\mathcal{P}$) + mots sur alphabet de pile
- *Décrit*: $\mathit{Conf}(\mathcal{A}) := \{\langle p, w \rangle : p \xrightarrow{w}_{\mathcal{A}} ◎\}$
- *Algorithme*: permet de calculer $\text{Pre}^*(\mathit{Conf}(\mathcal{A}))$ par saturation
- *Approche*: init. $\mathcal{B} := \mathcal{A}$ puis enrichir avec cette règle:



Vérification

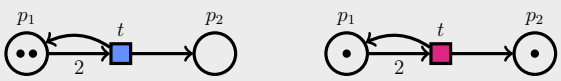
- *Approche*: système $\mapsto$ sys. à pile, spécification $\mapsto \mathcal{P}$-automate, vérification: par Pre^*/Post^*/automate de Büchi (LTL)
- *Applications*: raisonnement sur piles, par ex. « bytecode »



9. Systèmes infinis

Réseaux de Petri

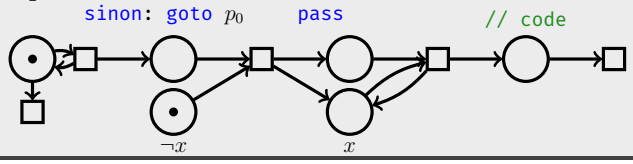
- *Déf.*: places et transitions reliées par arcs pondérés
- *Marquage*: nombre de jetons par place
- *Déclenchement*: si assez de jetons pour chaque arc entrant, les retirer, et en ajouter de nouveaux selon les arcs sortants
- *Successeurs*: $\text{Post}^*(\boldsymbol{m}) = \{\boldsymbol{m}' \in \mathbb{N}^P : \boldsymbol{m} \xrightarrow{*} \boldsymbol{m}'\}$
- *Prédecesseurs*: $\text{Pre}^*(\boldsymbol{m}') = \{\boldsymbol{m} \in \mathbb{N}^P : \boldsymbol{m} \xrightarrow{*} \boldsymbol{m}'\}$
- *Exemple*:



Modélisation

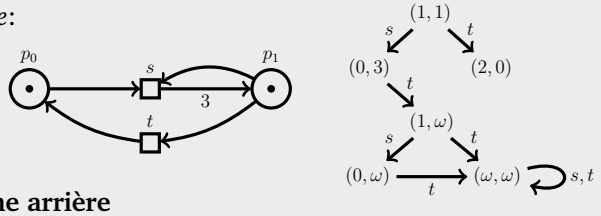
- *Processus*: comptés par les places
- *Exemple*:

```
si ¬x: x = vrai tant que ¬x:
    sinon: goto p0
    pass
// code
```



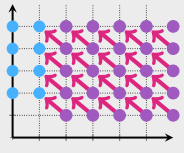
Graphes de couverture

- *Idée*: construire $\text{Post}^*(\boldsymbol{m})$ mais accélérer avec ω si $\boldsymbol{x} < \boldsymbol{x}'$
- *Test*: $\boldsymbol{m}'$ couvrable ssi le graphe contient un $\boldsymbol{m}'' \geq \boldsymbol{m}'$
- *Exemple*:



Algorithme arrière

- *Idée*: construire $\uparrow\text{Pre}^*(\uparrow\boldsymbol{m}')$ en déclenchant vers l'arrière
- *Représentation*: d'ensemble clos par le haut par base finie
- *Test*: $\boldsymbol{m}$ peut couvrir $\boldsymbol{m}'$ ssi découvert



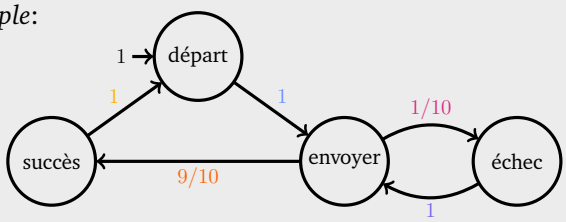
Accessibilité

- *Problème*: tester si $\boldsymbol{m}' \in \text{Post}^*(\boldsymbol{m})$
- *Décidable* mais plus compliqué

10. Systèmes probabilistes

Chaîne de Markov

- *But*: remplacer non-déterminisme par probabilités
- *Déf.*: struct. de Kripke avec proba. sur transitions / états init.
- *Représentation*: probabilités = matrice **P** et vecteur **init**
- *Exemple*:



- *Événements*: exéc. inf. décrites par préfixes finis (cylindres)
- *Probabilité*: somme du produit des transitions de cylindres
- *Exemple*: $\mathbb{P}(\mathsf{F}\,\text{succès}) = \sum_{i=0}^{\infty} 1 \cdot ((1/10) \cdot 1)^i \cdot (9/10) \cdot 1 = 1$
- *Outils*: PRISM/Storm (PCTL, analyse quantitative, etc.)

Accessibilité

- *Accessibilité*: événement de la forme $A \,\mathsf{U}\, B$
- *Partition*: $\underbrace{S_0 \coloneqq [\![\neg\exists(A \,\mathsf{U}\, B)]\!]}_{\text{prob. nulle}}$, $\underbrace{S_1 \coloneqq B}_{\text{prob. = 1}}$, $\underbrace{S_? \coloneqq S \setminus (S_0 \cup S_1)}_{\text{prob. à déterminer}}$
- *Approche*: $\mathbf{A} \coloneqq \mathbf{P}$ sur $S_?$; $\boldsymbol{b}(s) \coloneqq$ proba. d'aller de s vers S_1; $\boldsymbol{x}(s) = \mathbb{P}(s \models A \,\mathsf{U}\, B)$ est la solution de $(\mathbf{I} - \mathbf{A}) \cdot \boldsymbol{x} = \boldsymbol{b}$
- *Approx.*: $A \,\mathsf{U}^{\leq n}\, B$ obtenu par $f^n(\mathbf{0})$ où $f(\boldsymbol{y}) \coloneqq \mathbf{A} \cdot \boldsymbol{y} + \boldsymbol{b}$

Comportements limites

- *CFC terminales*: une est atteinte et parcourue avec proba. 1
- FG *et* GF: se calculent via accessibilité et CFC terminales

CTL probabiliste (PCTL)

- *Syntaxe*: comme CTL, mais $\exists$ / $\forall$ deviennent $\mathcal{P}_I$, et ajout $\mathsf{U}^{\leq n}$
- $\mathcal{P}_I(\varphi)$: proba. de φ dans intervalle I?
- $\mathsf{U}^{\leq n}$: côté droit satisfait en $\leq n$ étapes?
- *Vérification*: calcul récursif + éval. proba. d'accessibilité

Bibliographie

[And98] Henrik Reif Andersen : An introduction to binary decision diagrams, 1998.

[BEM97] Ahmed Bouajjani, Javier Esparza et Oded Maler : Reachability analysis of pushdown automata: Application to model-checking. *In Proc. 8th International Conference on Concurrency Theory (CONCUR)*, pages 135–150, 1997.

[BHO20] Michael Blondin, Christoph Haase et Philip Offtermatt : Directed reachability for infinite-state systems. *CoRR*, 2020.

[BK08] Christel Baier et Joost-Pieter Katoen : *Principles of model checking*. MIT Press, 2008.

[CLL+19] Wojciech Czerwiński, Sławomir Lasota, Ranko Lazić, Jérôme Leroux et Filip Mazowiecki : The reachability problem for Petri nets is not elementary. *In Proc. 51st Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 24–33, 2019.

[CVWY90] Costas Courcoubetis, Moshe Y. Vardi, Pierre Wolper et Mihalis Yannakakis : Memory efficient algorithms for the verification of temporal properties. *In Proc. 2nd International Workshop on Computer Aided Verification (CAV)*, volume 531, pages 233–242, 1990.

[DLF+16] Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault et Laurent Xu : Spot 2.0 – A framework for LTL and ω-automata manipulation. *In Proc. 14th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, pages 122–129. Springer, 2016. Outil disponible à https://spot.lrde.epita.fr/.

[EHRS00] Javier Esparza, David Hansel, Peter Rossmanith et Stefan Schwoon : Efficient algorithms for model checking pushdown systems. *In Proc. 12th International Conference on Computer Aided Verification (CAV)*, pages 232–247, 2000.

[Esp19] Javier ESPARZA : Automata theory: An algorithmic approach, 2019.

[FMW+17] Yu FENG, Ruben MARTINS, Yuepeng WANG, Isil DILLIG et Thomas W. REPS : Component-based synthesis for complex APIs. *In Proc. 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 599–612, 2017.

[Fé19] Vincent FÉLY : Génération de tests de vulnérabilité pour des programmes java card itératifs. Mémoire de maîtrise, 2019.

[HJK+20] Christian HENSEL, Sebastian JUNGES, Joost-Pieter KATOEN, Tim QUATMANN et Matthias VOLK : The probabilistic model checker storm. *CoRR*, abs/2002.07080, 2020.

[KMS18] Jan KRETÍNSKÝ, Tobias MEGGENDORFER et Salomon SICKERT : Owl: A library for ω-words, automata, and LTL. *In Proc. 16th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, pages 543–550. Springer, 2018. Outil disponible à https://owl.model.in.tum.de/.

[KNP11] M. KWIATKOWSKA, G. NORMAN et D. PARKER : PRISM 4.0: Verification of probabilistic real-time systems. *In Proc. 23rd International Conference on Computer Aided Verification (CAV)*, pages 585–591, 2011.

[Kos82] S. Rao KOSARAJU : Decidability of reachability in vector addition systems (preliminary version). *In Proc. 14th Symposium on Theory of Computing (STOC)*, pages 267–281, 1982.

[Lam92] Jean-Luc LAMBERT : A structure to decide reachability in Petri nets. *Theoretical Computer Science*, 99(1):79–104, 1992.

[Ler12] Jérôme LEROUX : Vector addition systems reachability problem (A simpler solution). *In Turing-100 – The Alan Turing Centenary*, pages 214–228, 2012.

[Lip76] Richard J. LIPTON : The reachability problem requires exponential space. Rapport technique 63, Department of Computer Science, Yale University, 1976.

[May81] Ernst W. MAYR : An algorithm for the general Petri net reachability problem. *In Proc. 13th Symposium on Theory of Computing (STOC)*, pages 238–246, 1981.

[McN66] Robert MCNAUGHTON : Testing and generating infinite sequences by a finite automaton. *Information and Control*, 9(5):521–530, 1966.

[Mer01] Stephan MERZ : Model checking: A tutorial overview. *In Modeling and Verification of Parallel Processes*, pages 3–38. 2001.

[Rac78] Charles RACKOFF : The covering and boundedness problems for vector addition systems. *Theoretical Computer Science*, 6:223–231, 1978.

[ST77] George S. SACERDOTE et Richard L. TENNEY : The decidability of the reachability problem for vector addition systems (preliminary version). *In Proc. 9^{th} Symposium on Theory of Computing (STOC)*, pages 61–76, 1977.

Index