IGL752 – Techniques de vérification et de validation

NOTES DE COURS

Michael Blondin

3 avril 2019

Ce document

Ce document sert de notes complémentaires au cours IGL752 – Techniques de vérification et de validation de l'Université de Sherbrooke.

Si vous trouvez des coquilles ou des erreurs dans le document, veuillez s.v.p. me les indiquer par courriel à michael.blondin@usherbrooke.ca.

Table des matières

1

Systèmes de transition

Afin de vérifier rigoureusement qu'un système satisfait sa spécification, nous devons modéliser ce système formellement. Les systèmes de transition permettent une telle modélisation sous forme de graphes.

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 que $\mathcal{T}$ est *fini* si S est fini.

1.1 Exemples

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

$$S \stackrel{\text{def}}{=} \{s_0, s_1, s_2\},$$
$$I \stackrel{\text{def}}{=} \{s_0\},$$
$$\to \stackrel{\text{def}}{=} \{(s_0, s_0), (s_0, s_1), (s_0, s_2), (s_1, s_2), (s_2, s_2)\}.$$

Le système $\mathcal{T}_1$ peut être représenté graphiquement tel qu'illustré à la figure 1.1.

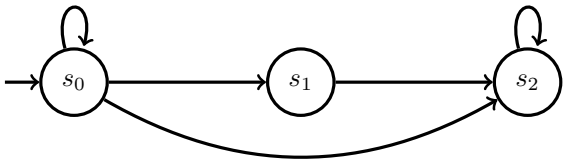


FIGURE 1.1 – Exemple de système de transition.

Considérons les deux processus suivants s'exécutant de façon concurrente et partageant une variable $tour \in \{0, 1\}$:

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

Le système de transition naturellement associé à ce système est illustré à la figure 1.2.

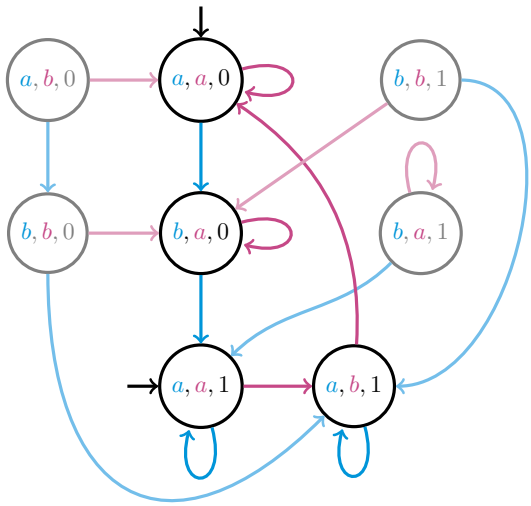


FIGURE 1.2 – Système de transition associé aux processus P_0 et P_1. Un état (x, y, z) indique que P_0 est à la ligne x, que P_1 est à la ligne y, et que $tour = z$. Les composantes et les transitions associées à P_0 (resp. P_1) apparaissent en cyan (resp. magenta). Les états non accessibles à partir des états initiaux sont plus pâles.

Considérons le programme suivant constitué de trois fonctions et dont le point d'entrée est la fonction `main`, 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 associé à la pile d'appel de ce programme est illustré à la figure 1.3. Ce système de transition est infini puisqu'il existe une exécution infinie: `main() foo() bar() foo() bar()` $\cdots$.

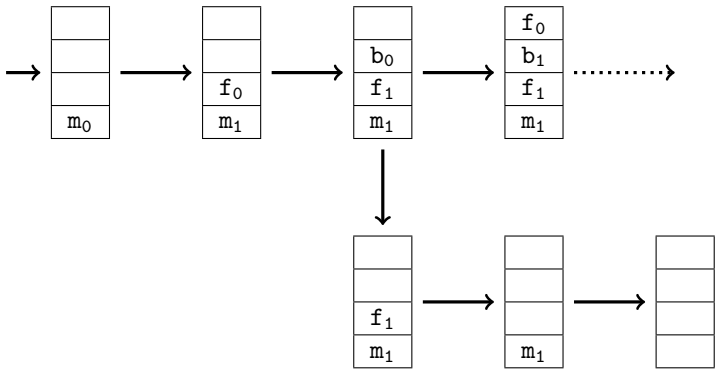


FIGURE 1.3 – Système de transition associé à la pile d'appel à partir de `main`.

1.2 Prédecesseurs 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 *prédecesseur immédiat* de t, et que t est un *successeur immédiat* de s. L'ensemble des successeurs et prédecesseurs immédiats d'un état $s \in S$ sont respectivement dénotés:

$$\mathrm{Post}(s) \stackrel{\text{def}}{=} \{t \in S : s \to t\},$$
$$\mathrm{Pre}(s) \stackrel{\text{def}}{=} \{t \in S : t \to s\}.$$

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

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

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

$$\mathrm{Post}^*(s) \stackrel{\text{def}}{=} \{t \in S : s \stackrel{*}{\to} t\},$$
$$\mathrm{Pre}^*(s) \stackrel{\text{def}}{=} \{t \in S : t \stackrel{*}{\to} s\}.$$

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

Reconsidérons le système de transition $\mathcal{T}_1$ illustré à la figure 1.1. Nous avons:

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

1.3 Chemins et exécutions

Soit $\mathcal{T} = (S, \to, I)$ un système de transition. Un *chemin fini* est une séquence finie d'états $s_0 s_1 \cdots s_n$ telle que $s_0 \to s_1 \to \cdots \to s_n$. Similairement, un *chemin infini* est une séquence infinie d'états $s_0 s_1 \cdots$ telle que $s_0 \to s_1 \to \cdots$. Un *chemin* est un chemin fini ou infini. Nous disons qu'un chemin est *initial* si $s_0 \in I$. Nous disons qu'un chemin 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.

Reconsidérons le système de transition $\mathcal{T}_1$ illustré à la figure 1.1. La séquence $\rho = s_1 s_2 s_1 s_2$ est un chemin fini de $\mathcal{T}_1$ et la séquence $\rho' = s_0 s_0 s_1 s_2 s_1 s_2 \cdots$ 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.

1.4 Explosion combinatoire

En général, la taille d'un système de transition fini 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 prouver la présence ou l'absence d'erreurs.

Nous verrons certaines de ces approches plus tard.

1.5 Structures de Kripke

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. 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.

Un exemple de structure de Kripke $\mathcal{T}_2$ est illustré à la figure 1.4.

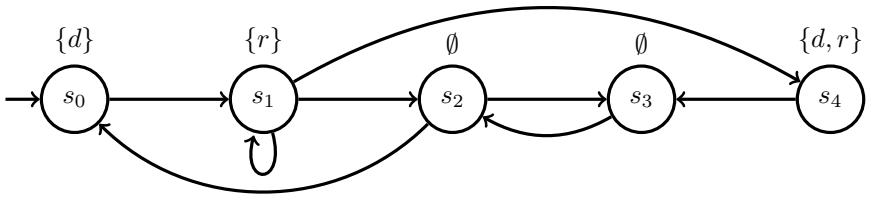


FIGURE 1.4 – Exemple de structure de Kripke où $AP = \{d, r\}$.

Supposons que les propositions atomiques d et r correspondent respectivement à l'occurrence d'une *demande* et d'une *réponse*. Il est possible de 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 puisqu'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 $\mathcal{T}_2$ 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 $\rho = s_0 s_1 s_4 s_3 s_2 s_3 s_2 \cdots$ n'est pas suivie d'une demande.

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

Nous verrons plus tard comment de telles propriétés peuvent être formalisées en logique temporelle afin d'éviter toute ambiguïté et d'automatiser leur vérification.

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. 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 de raisonner sur les comportements infinis; 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 à ce problème, nous introduisons une logique temporelle; une logique qui étend la logique propositionnelle avec des opérateurs permettant de raisonner sur le temps.

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:

$$\varphi = \mathit{vrai} \mid p \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \mathsf{X}\varphi \mid \varphi_1 \,\mathsf{U}\, \varphi_2$$

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

- *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.

Notons que nous utilisons le symbole X pour dénoter le symbole $\bigcirc$ utilisé dans le manuel [BK08]. Les opérateurs X et U se nomment respectivement *next* et *until*.

2.2 Autres opérateurs

Nous introduisons des opérateurs supplémentaires qui seront utiles afin de modéliser des propriétés. Ces opérateurs sont définis en fonction des opérateurs déjà définis:

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

Notons que nous utilisons les symboles F et G pour dénoter respectivement les symboles $\Diamond$ et $\Box$ utilisés dans le manuel [BK08]. Les opérateurs F et G se nomment respectivement *finally* et *globally*.

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*).

2.3 Sémantique

Nous associons un sens formel aux formules LTL. Soit Σ un ensemble. Un *mot infini* sur Σ est une fonction $\sigma\colon \mathbb{N} \rightarrow \Sigma$ 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 $u, v \in \Sigma^*$, nous dénotons par uv^ω le mot infini $uvvv\cdots$. Pour tous $\sigma \in \Sigma^\omega$ et $i \in \mathbb{N}$, nous définissons $\sigma[i..] \stackrel{\text{def}}{=} \sigma(i)\sigma(i+1)\cdots$. Autrement dit, $\sigma[i..]$ est le suffixe infini de σ obtenu en débutant à sa $i^{\text{ème}}$ composante (à partir de 0).

Pour tout $\sigma \in (2^{AP})^\omega$, nous disons que:

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

La notation $\sigma \models \varphi$ signifie « le mot σ satisfait la formule φ ». Pour toute formule LTL φ sur AP, l'ensemble des mots qui satisfont φ est défini par:

$$[\![\varphi]\!] \stackrel{\text{def}}{=} \{\sigma \in (2^{AP})^\omega : \sigma \models \varphi\}.$$

Notons que nous utilisons la notation $[\![\varphi]\!]$ plutôt que la notation $Words(\varphi)$ du manuel [BK08]. L'intuition derrière la sémantique des formules LTL est illustrée à la figure 2.1.

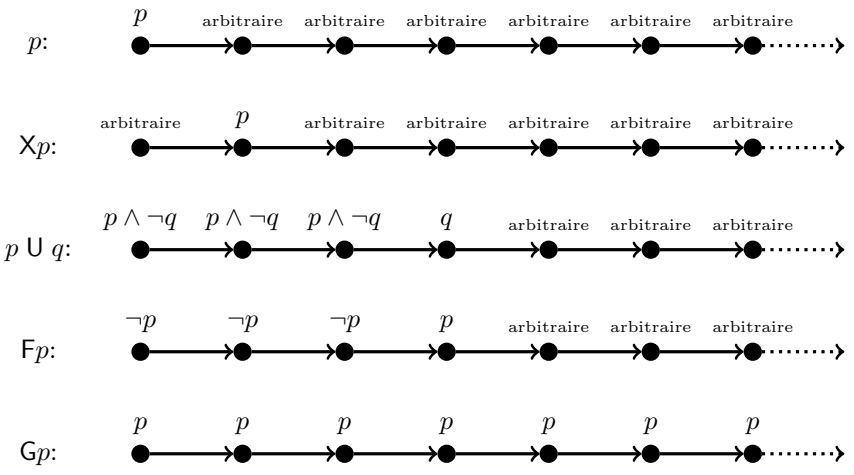


FIGURE 2.1 – Illustration de la sémantique de la logique linéaire temporelle. Chaque ligne représente un mot w infini où les cercles sont les positions de w et l'étiquette au-dessus du $i^{\text{ème}}$ cercle indique les propositions de $w(i)$.

Exemple 2.1. Soient $AP = \{p, q\}$ et le mot infini $\sigma \in (2^{AP})^\omega$ tel que

$$\sigma \stackrel{\text{def}}{=} \{p\}\emptyset\{q\}\{p,q\}(\{p\}\{q\})^\omega.$$

Nous avons:

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

Il est possible de démontrer que la sémantique de F et G 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) $\sigma \models \mathsf{F}\varphi \iff \exists j \geq 0 : \sigma[j..] \models \varphi$,

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

Démonstration.

(a)

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

(b)

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

Certains concepts de la logique propositionnelle peuvent être étendus 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*.

2.4 Équivalences

Nous répertorions quelques équivalences entre formules LTL qui peuvent simplifier la spécification de propriétés.

2.4.1 Distributivité

Les opérateurs temporels se distribuent de la façon suivante sur les opérateurs logiques:

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

Notons qu'en général $\mathsf{F}(\varphi_1 \wedge \varphi_2)$ n'est pas équivalent à $\mathsf{F}\varphi_1 \wedge \mathsf{F}\varphi_2$. En effet, posons $AP = \{p, q\}$ et $\sigma = (\{p\}\{q\})^\omega$. 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 la façon suivante 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 les règles d'idempotence suivantes:

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

2.4.4 Absorption

Les opérateurs temporels satisfont les règles d'absorption suivantes:

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

Ainsi, il n'existe que quatre combinaisons des opérateurs F et G:

$$\{\mathsf{F}, \mathsf{G}, \mathsf{FG}, \mathsf{GF}\}.$$

2.5 Propriétés d'un système

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

$$\text{trace}(s_0 s_1 \cdots) \stackrel{\text{def}}{=} L(s_0)L(s_1)\cdots .$$

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

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

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

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

Autrement dit, $\mathcal{T}$ satisfait φ si la trace de chacune de ses exécutions infinies satisfait φ.

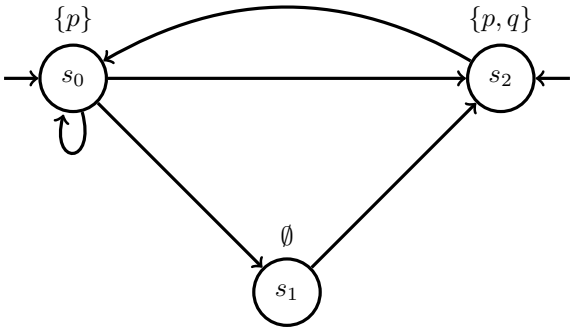


FIGURE 2.2 – Structure de Kripke avec propositions atomiques $AP = \{p, q\}$.

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

$$\begin{aligned}
&\mathcal{T} \models p,\\
&\mathcal{T} \not\models \mathsf{G}p,\\
&\mathcal{T} \models \mathsf{GF}p,\\
&\mathcal{T} \not\models (\neg q)\ \mathsf{U}\ q.
\end{aligned}$$

Notons qu'il est possible qu'une propriété *et* sa négation ne soit pas satisfaite par un système. Par exemple, dans l'exemple précédent, 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_1 ne satisfait pas $\neg(p \wedge q)$.

Exemple 2.2. Considérons le circuit logique $\mathcal{C}$ de la figure 2.3 constitué des registres r_1 et r_2, et de la sortie s. Nous 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 est 1. Nous spécifions d'abord que la sortie de $\mathcal{C}$ doit valoir 1 au moins une fois par cycle de taille trois:

$$\varphi_{>1} \stackrel{\text{def}}{=} \mathsf{G}(s \vee \mathsf{X}s \vee \mathsf{XX}s).$$

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

$$\varphi_{\leq 1} \stackrel{\text{def}}{=} \mathsf{G}(s \rightarrow (\mathsf{X}\neg s \wedge \mathsf{XX}\neg s)).$$

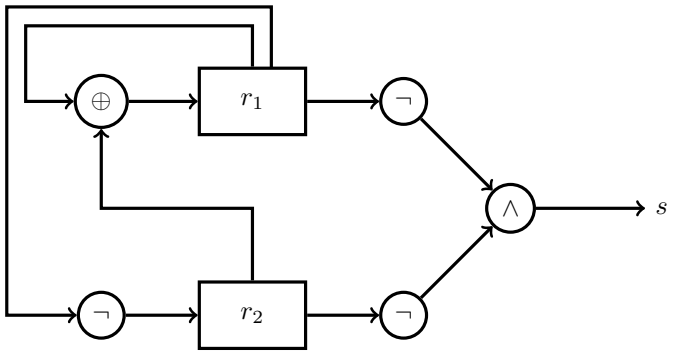


FIGURE 2.3 – Circuit logique d'un compteur modulo 3. Inspiré de l'exemple 5.11 de [BK08].

La formule $\psi = s \wedge \varphi_{>1} \wedge \varphi_{\leq 1}$ 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

2.6.1 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, considérons 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 »: $\{p, q\}\{p\}\emptyset$.

2.6.2 Vivacité

Informellement, une *propriété de vivacité* est une propriété qui décrit une forme de progrès. Contrairement aux propriétés de sûreté, une propriété de vivacité ne peut pas être prouvée ou réfutée à l'aide d'un préfixe fini. Voici quelques exemples de propriétés de vivacité:

— $\mathsf{GF}p$: p est satisfaite infiniment souvent;
— $\mathsf{FG}p$: p est éventuellement toujours satisfaite (*propriété de persistance*);
— $\mathsf{G}(p \rightarrow \mathsf{F}q)$: lorsque p est satisfaite, q est éventuellement satisfaite.

2.6.3 Invariants et accessibilité

Un *invariant* est une propriété de la forme $\mathsf{G}\varphi$. En particulier, l'*exclusion mutuelle* est un invariant qui s'écrit $\mathsf{G}\neg(\varphi_1 \wedge \varphi_2)$. Une *propriété d'accessibilité* est

une propriété de la forme $\mathsf{F}\varphi$. Tout invariant est la négation d'une propriété d'accessibilité par dualité de G et F.

2.7 Équité

Certaines propriétés formelles d'un système concurrent peuvent parfois être enfreintes en raison d'exécutions inéquitables qui priorisent systématiquement certains processus. Par exemple, supposons qu'un système possède n processus $P_1, P_2, \ldots, P_n$. Pour tout $1 \leq i \leq n$, définissons les propositions atomiques:

$$p_i \stackrel{\text{def}}{=} P_i \text{ peut exécuter une instruction,}$$
$$q_i \stackrel{\text{def}}{=} P_i \text{ exécute une instruction.}$$

Considérons la trace $\sigma \stackrel{\text{def}}{=} \{p_1, q_1, p_2\}^\omega$. Cette trace correspond à une exécution où seules les instructions du processus P_1 sont exécutées, bien que les instructions du processus 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 P_2 est toujours exécutable, mais n'est jamais exécuté.

Équité faible. Il peut souvent être pratique d'ignorer ce type d'exécutions inéquitables. Plus formellement, posons $\varphi_i \stackrel{\text{def}}{=} \mathsf{FG}(p_i \wedge \neg q_i)$ 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:

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

Il est possible de spécifier que toutes les traces faiblement équitables d'un système doivent satisfaire une proprité ψ, et ignorer les traces non équitables, avec la 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.

Notons que:

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

Ainsi de façon équivalente, qui est celle utilisée dans [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 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 $\varphi'_i \stackrel{\text{def}}{=} (\mathsf{GF}p_i \wedge \mathsf{FG}\neg q_i)$ 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 si 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:

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

Ainsi de façon équivalente, qui est celle utilisée dans [BK08], une trace σ est faiblement équitable ssi:

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

3

Langages ω-réguliers

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 tester algorithmiquement si $\text{Traces}(\mathcal{T}) \subseteq [\![\varphi]\!]$?

Dans cette optique, observons que $\text{Traces}(\mathcal{T})$ et $[\![\varphi]\!]$ sont tous deux des langages de mots infinis sur l'alphabet 2^{AP}. Ainsi, nous chercherons à représenter ces deux langages sous forme d'automates pouvant être manipulés algorithmiquement.

3.1 Expressions ω-régulières

Avant d'introduire les automates sur les mots infinis, nous introduisons les expressions ω-régulières qui faciliteront la description de langages d'automates.

3.1.1 Syntaxe

Soit Σ un alphabet fini. La syntaxe des *expressions ω-régulières* sur alphabet Σ est définie par la grammaire suivante:

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

où $a \in \Sigma$. Autrement dit, l'ensemble des expressions ω-régulières sur Σ est défini récursivement par:

- Si r est une expression régulière, 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.

3.1.2 Sémantique

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

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

Rappelons que le *langage* $L(r) \subseteq \Sigma^*$ d'une expression régulière r est défini récursivement par:

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

3.1.3 Exemples

Considérons l'alphabet $\Sigma = \{a, b\}$. Voici quelques exemples d'expressions ω-régulières sur Σ:

$$\begin{aligned} (a+b)^\omega &= \text{ensemble de tous les mots (infinis)}, \\ a(a+b)^\omega &= \text{ensemble des mots débutant par la lettre } a, \\ (ab)^\omega &= \text{ensemble contenant l'unique mot } ababab\cdots, \\ b^*(aa^*bb^*)^\omega &= \text{ensemble des mots avec une infinité de } a \textit{ et de } b, \\ (b^*a^*ab)^\omega &= \text{ensemble des mots avec une infinité de } a \textit{ et de } b, \\ (a^*ab + b^*ba)^\omega &= \text{ensemble des mots avec une infinité de } a \textit{ et de } b, \\ (a+b)^*b^\omega &= \text{ensemble des mots avec un nombre fini de } a, \\ (a(a+b))^\omega &= \text{ensemble des mots avec un } a \text{ à chaque position paire}. \end{aligned}$$

3.2 Automates de Büchi

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 une *fonction de transition*,
- $Q_0 \subseteq Q$ est un ensemble d'*états initiaux*,
- $F \subseteq Q$ est un ensemble d'*états acceptants*.

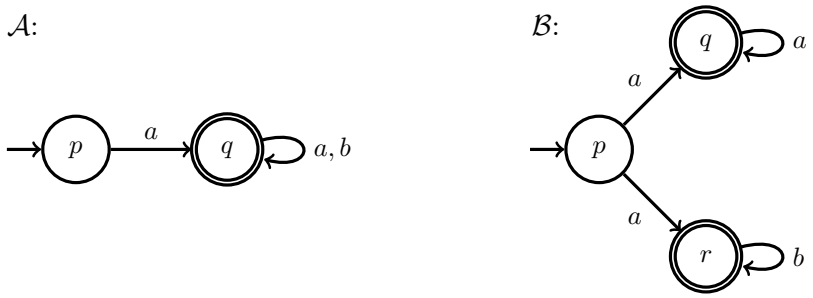


FIGURE 3.1 – Exemples d'automates de Büchi. L'automate $\mathcal{A}$ accepte le langage $a(a+b)^\omega$ et l'automate $\mathcal{B}$ accepte le langage $a^\omega + ab^\omega$.

3.2.1 Transitions

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. Par exemple, considérons les automates de Büchi $\mathcal{A}$ et $\mathcal{B}$ illustrés à la figure 3.1. Pour l'automate $\mathcal{A}$, nous avons:

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

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

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

3.2.2 Déterminisme

Nous disons qu'un automate de Büchi est *déterministe* si $|\delta(q, a)| \leq 1$ pour tous $q \in Q$ et $a \in \Sigma$. Autrement dit, un automate de Büchi est déterministe s'il ne possède aucun état ayant deux transitions sortantes étiquetées par la même lettre. 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 $p \xrightarrow{a} r$.

3.2.3 Langage d'un automate

Nous disons qu'un mot infini $\sigma \in \Sigma^\omega$ est *accepté* par un automate de Büchi $\mathcal{A}$ s'il existe une suite 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 σ est accepté s'il est possible de lire σ à partir d'un état initial et via une suite de transitions qui visite infiniment souvent des états acceptants.

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

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

Par exemple, le langage des automates $\mathcal{A}$ et $\mathcal{B}$ illustrés à la figure 3.1 est, respectivement, celui décrit par les expressions $a(a+b)^\omega$ et $a^\omega + ab^\omega$.

Notons que contrairement aux automates sur les mots finis, le déterminisme est strictement plus faible que le non déterminisme. 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 possédant un nombre fini de b.

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

3.2.4 Exemples

Considérons les automates de Büchi, sur alphabet $\Sigma = \{a, b, c\}$, illustrés à la figure 3.2. Le langage accepté par ces automates est décrit au tableau suivant:

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

3.3 Intersection d'automates de Büchi

Soient $\mathcal{A} = (Q_\mathcal{A}, \Sigma, \delta_\mathcal{A}, Q_{0,A}, F_\mathcal{A})$ et $\mathcal{B} = (Q_\mathcal{B}, \Sigma, \delta_\mathcal{B}, Q_{0,B}, F_\mathcal{B})$ des automates de Büchi sur un alphabet commun. Nous montrons comment construire un automate de Büchi $\mathcal{C} = (Q, \Sigma, \delta, Q_0, F)$ tel que $L(\mathcal{C}) = L(\mathcal{A}) \cap L(\mathcal{B})$. Autrement dit, $\mathcal{C}$ acceptera précisément les mots acceptés par $\mathcal{A}$ *et* $\mathcal{B}$.

3.3.1 Construction à partir d'un exemple

Avant de voir la construction générale de $\mathcal{C}$, considérons le cas particulier des automates $\mathcal{A}$ et $\mathcal{B}$ illustré à la figure 3.3. Le langage de l'automate $\mathcal{A}$ est l'ensemble des mots avec 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.

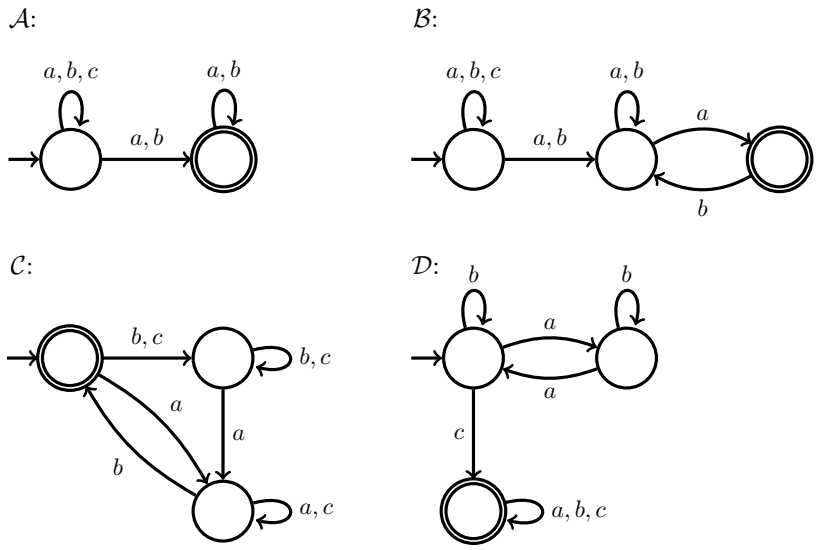


FIGURE 3.2 – Autres exemples d'automates de Büchi.

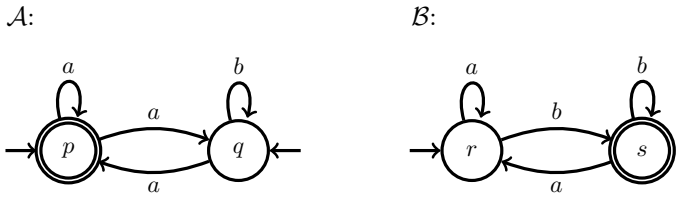


FIGURE 3.3 – Deux automates de Büchi $\mathcal{A}$ et $\mathcal{B}$ pour lesquels nous désirons construire l'intersection.

Afin de s'assurer qu'un mot peut être lu à la fois par $\mathcal{A}$ et par $\mathcal{B}$, nous simulons les deux automates simultanément « en parallèle ». Cela peut être réalisé en opérant sur le produit des états de $\mathcal{A}$ et $\mathcal{B}$. L'automate $\mathcal{D}$ obtenu de cette manière est illustré à la figure 3.4.

L'automate $\mathcal{D}$ peut lire précisément les mots pouvant être lus dans $\mathcal{A}$ et $\mathcal{B}$. Toutefois, $\mathcal{D}$ ne possède pas assez d'information pour décider quels mots doivent être acceptés ou refusés. En effet, aucun mot ne franchit les états acceptants de $\mathcal{A}$ et $\mathcal{B}$ simultanément.

Afin de pallier ce problème, nous ajoutons une composante additionnelle aux états de $\mathcal{D}$ qui indique lequel de $\mathcal{A}$ et $\mathcal{B}$ est le prochain à devoir franchir un état acceptant. L'automate $\mathcal{C}$ obtenu de cette façon est illustré à la figure 3.5.

L'idée principale derrière l'automate $\mathcal{C}$ 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

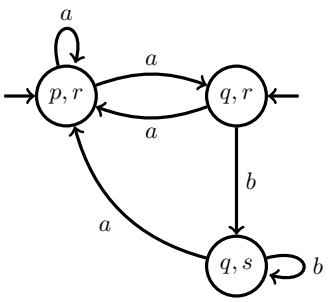


FIGURE 3.4 – Produit des automates $\mathcal{A}$ et $\mathcal{B}$ de la figure 3.3.

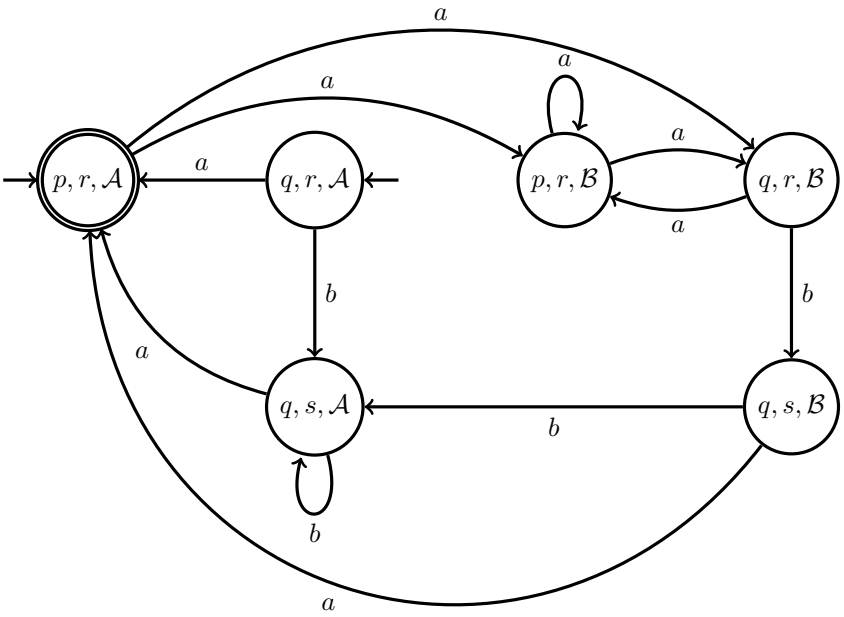


FIGURE 3.5 – Intersection des automates $\mathcal{A}$ et $\mathcal{B}$ de la figure 3.3.

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. Cela permet d'alterner indéfiniment entre les états acceptant de $\mathcal{A}$ et $\mathcal{B}$, *tout en* simulant les deux automates en parallèles. Les états acceptants de $\mathcal{C}$ 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'un des deux afin d'assurer que $\mathcal{C}$ alterne entre les deux copies.

3.3.2 Construction générale

Nous donnons maintenant la construction générale de l'automate de Büchi $\mathcal{C}$ pour deux automates $\mathcal{A}$ et $\mathcal{B}$ arbitraires. Ses états, états initiaux et états acceptants sont définis respectivement par:

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

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 $\mathcal{C}$ la transition:

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

La figure 3.6 illustre la construction de $\mathcal{C}$ sur un autre exemple.

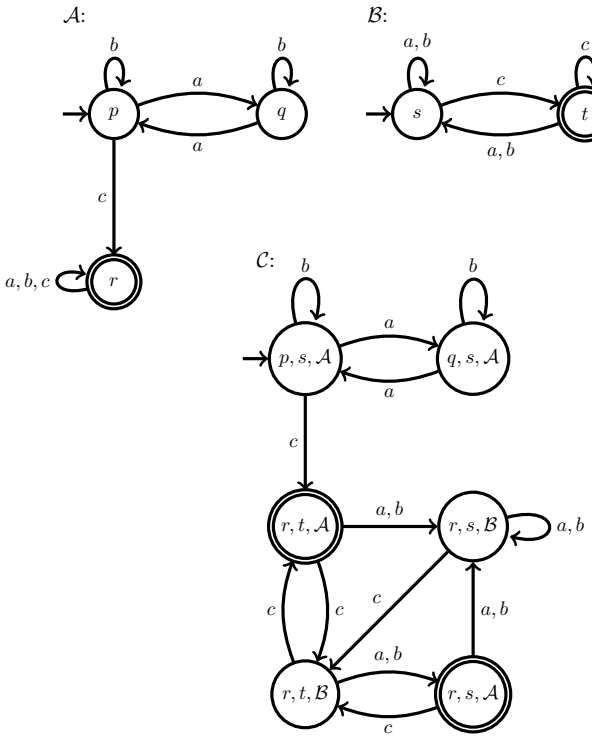


FIGURE 3.6 – Automate $\mathcal{C}$ pour l'intersection des automates $\mathcal{A}$ et $\mathcal{B}$.

4

Vérification algorithmique de formules LTL

Dans ce chapitre, nous voyons comment automatiser la vérification de formules LTL à partir d'automates de Büchi.

4.1 Formules LTL vers automates de Büchi

Il est possible de traduire une formule LTL φ, sur propositions atomiques AP, vers un automate de Büchi $\mathcal{A}$, sur alphabet 2^{AP}, tel que $L(\mathcal{A}) = [\![\varphi]\!]$. Voyons quelques exemples.

Posons $AP = \{p, q\}$ et $\Sigma = 2^{AP}$. 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, les mots qui satisfont ces formules peuvent être représentés par les automates illustrés à la figure 4.1.

Tel que décrit dans [BK08], il existe un algorithme qui convertit une formule LTL arbitraire vers un automate de Büchi équivalent. Dans le pire cas, cet automate est exponentiellement plus grand que la formule. En pratique, il existe plusieurs heuristiques pour réduire la taille de cet automate. Nous n'entrerons pas dans les détails (assez complexes) de cet algorithme et de ces heuristiques.

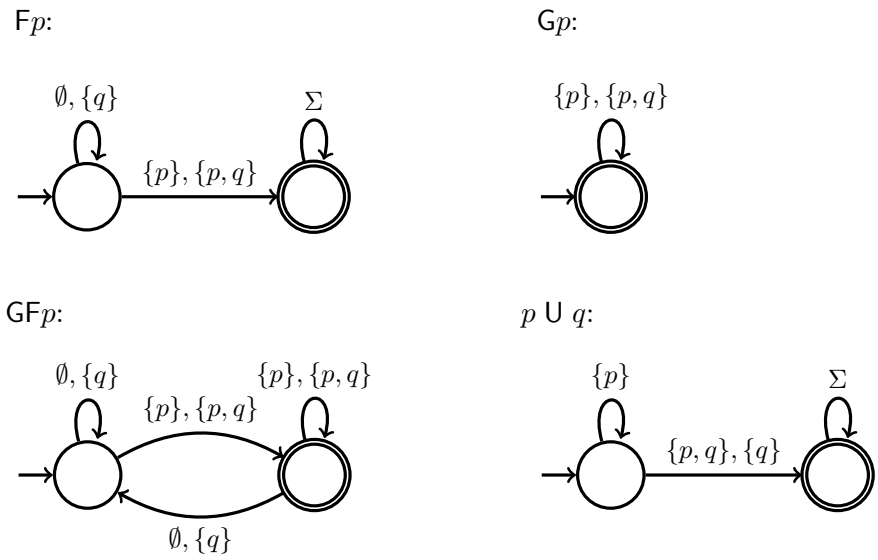


FIGURE 4.1 – Automates de Büchi pour différentes formules LTL.

4.2 Structures de Kripke vers automates de Büchi

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 $L(\mathcal{A}_\mathcal{T}) = \text{Traces}(\mathcal{T})$. L'automate $\mathcal{A}_\mathcal{T}$ est obtenu à partir de $\mathcal{T}$ en rendant ses états acceptants et en étiquetant chaque transition $s \to t$ par l'étiquette $L(s)$. La figure 4.2 donne un exemple de cette construction.

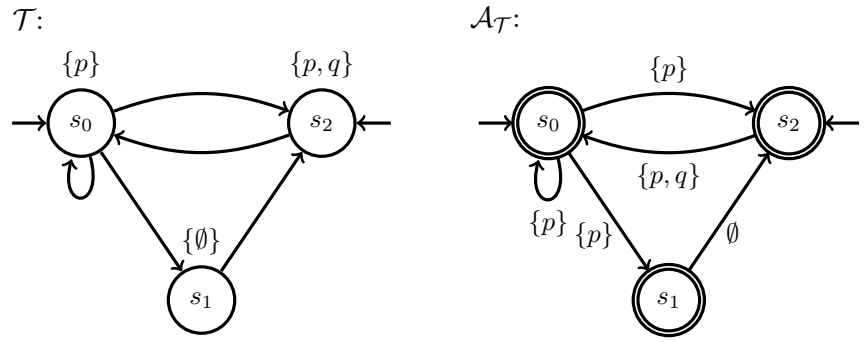


FIGURE 4.2 – Conversion d'une structure de Kripke $\mathcal{T}$ vers un automate de Büchi $\mathcal{A}_\mathcal{T}$ équivalent.

Plus formellement, l'automate $\mathcal{A}_\mathcal{T}$ est défini par $\mathcal{A}_\mathcal{T} \stackrel{\text{def}}{=} (S, 2^{AP}, \delta, I, S)$ où

δ est défini par:

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

Nous avons comme conséquence quasi immédiate de la définition que:

Proposition 2. *$L(\mathcal{A}_\mathcal{T}) = \text{Traces}(\mathcal{T})$ pour toute structure de Kripke $\mathcal{T}$.*

4.3 Vérification d'une spécification

Soient $\mathcal{T}$ une structure de Kripke, φ une formule LTL et $\mathcal{B}_{\neg\varphi}$ un automate de Büchi dont le langage est $[\![\neg\varphi]\!]$. Nous avons:

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

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

4.3.1 Lassos

Nous présentons une condition suffisante et nécessaire afin de déterminer si le langage d'un automate de Büchi est vide. Soit $\mathcal{C} = (Q, \Sigma, \delta, Q_0, F)$ un automate de Büchi. Un *lasso* de $\mathcal{C}$ 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 $v \in \Sigma^+$. Nous avons:

Proposition 3. *Soit $\mathcal{C}$ un automate de Büchi. Nous avons $L(\mathcal{C}) \neq \emptyset$ si et seulement si $\mathcal{C}$ possède un lasso.*

Démonstration. $\Leftarrow$) Supposons que $\mathcal{C}$ 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, $uv^\omega \in L(\mathcal{C})$, ce qui implique que $L(\mathcal{C})$ est non vide.

$\Rightarrow$) Supposons que $L(\mathcal{C}) \neq \emptyset$. 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 $i < j$ tels que $q_i = q_j$. Nous obtenons donc le lasso suivant:

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

4.3.2 Détection algorithmique de lassos

Une approche simple afin de détecter un lasso dans un automate de Büchi consiste à:

(1) calculer l'ensemble R des états accessibles à partir des états initiaux;

(2) pour chaque état *acceptant* $p \in R$, chercher un chemin non vide de p vers lui-même.

Cette procédure est décrite à l'algorithme 1.

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

Algorithme 1 : Algorithme quadratique de détection de lasso.

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

```
// Calculer l'ensemble des états accessibles
R ← ∅

pour q ∈ Q_0 faire
    si q ∉ R alors
        dfs(q)

dfs(q):
    ajouter q à R
    pour r : q → r faire
        si r ∉ R alors dfs(r)

// Chercher un état acceptant qui peut accéder à lui-même
pour p ∈ R ∩ F faire
    S_p ← ∅
    cycle(p)

    cycle(q):
        ajouter q à S_p
        pour r : q → r faire
            si r = p alors resultat "non vide"
            sinon si r ∉ S_p alors cycle(r)

resultat "vide"
```

En pratique, les structures de Kripke possèdent une quantité massive d'états. Une complexité quadratique n'est donc pas raisonnable. Il existe plusieurs algorithmes de complexité linéaire afin de détecter des lassos. Par exemple, l'algorithme de parcours en profondeur imbriqué de Courcoubetis, Vardi, Wolper

et Yannakakis combine les deux étapes de l'algorithme quadratique. Cette approche est décrite à l'algorithme 2, telle que présentée dans [Esp19]. Voir [Esp19, chapitre 13], par exemple, pour une preuve que cet algorithme fonctionne, ainsi qu'une présentation d'autres algorithmes de détection de lasso.

Algorithme 2 : Algorithme linéaire de détection de lasso.

Entrées : Automate de Büchi $\mathcal{C} = (Q, \Sigma, \delta, Q_0, F)$
Sorties : $L(\mathcal{C}) = \emptyset$?
$S_1 \leftarrow \emptyset$
$S_2 \leftarrow \emptyset$

pour $q \in Q_0$ **faire**
  si $q \notin S_1$ **alors**
    `dfs1(`q`)`
resultat *"vide"*

`dfs1(`q`):`
  ajouter q **à** S_1
  pour $r : q \to r$ **faire**
    si $r \notin S_1$ **alors** `dfs1(`r`)`
  si $q \in F$ **alors**
    $c \leftarrow q$
    `dfs2(`q`)`

`dfs2(`q`):`
  ajouter q **à** S_2
  pour $r : q \to r$ **faire**
    si $r = c$ **alors resultat** *"non vide"*
    sinon si $r \notin S_2$ **alors** `dfs2(`r`)`

5

Logique temporelle arborescente (CTL)

La logique temporelle linéaire 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. LTL ne permet pas de spécifier une propriété telle que: « il est toujours possible de revenir à l'état initial de $\mathcal{T}$ ». Cette limitation est due à la modélisation linéaire du temps inhérente à LTL. La logique temporelle arborescente (CTL) adopte une autre modélisation: 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. Dans ce chapitre, nous étudions cette logique.

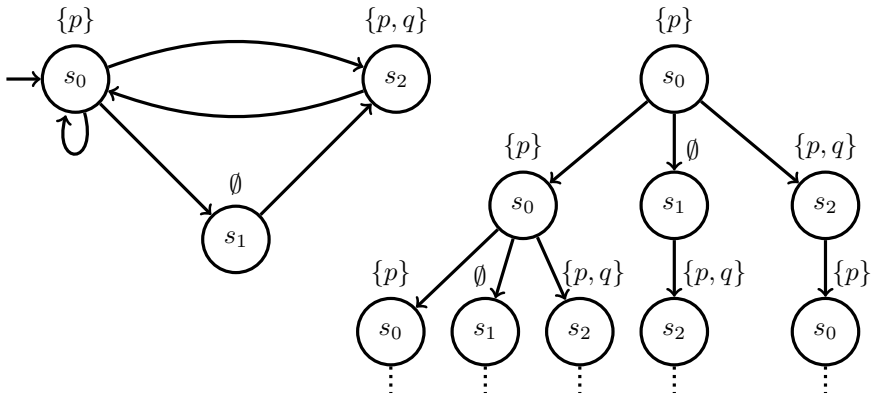


FIGURE 5.1 – Une structure de Kripke (à gauche), et son arbre de calcul (à droite), autrement dit le « déroulement » des chemins de la structure sous forme d'arborescence.

5.1 Syntaxe

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

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

où $p \in AP$. Nous disons qu'une telle formule Φ est une *formule d'état* et qu'une telle formule φ est une *formule de chemin*. Notons que nous utilisons le symbole X pour dénoter le symbole $\bigcirc$ utilisé dans le manuel [BK08].

5.2 Autres opérateurs

Les opérateurs logiques $\vee$, $\rightarrow$, $\leftrightarrow$ et $\oplus$ peuvent être définis de façon naturelle à partir de $\wedge$ et $\neg$. Nous introduisons des opérateurs supplémentaires qui seront utiles afin de modéliser des propriétés:

$$\exists\mathsf{F}\Phi \stackrel{\text{def}}{=} \exists(\mathit{vrai} \;\mathsf{U}\; \Phi)$$
$$\forall\mathsf{F}\Phi \stackrel{\text{def}}{=} \forall(\mathit{vrai} \;\mathsf{U}\; \Phi)$$
$$\exists\mathsf{G}\Phi \stackrel{\text{def}}{=} \neg\forall\mathsf{F}\neg\Phi$$
$$\forall\mathsf{G}\Phi \stackrel{\text{def}}{=} \neg\exists\mathsf{F}\neg\Phi$$

Notons que nous utilisons les symboles F et G pour dénoter respectivement les symboles $\Diamond$ et $\Box$ utilisés dans le manuel [BK08].

5.3 Sémantique

Soit $\mathcal{T} = (S, \rightarrow, I, AP, L)$ une structure de Kripke. Nous associons un sens formel aux formules CTL en fonction de $\mathcal{T}$. Les formules d'état sont interprétées sur S et les formules de chemin sur les chemins infini de $\mathcal{T}$. Pour tout $s \in S$, nous disons que:

$s \models \mathit{vrai}$

$s \models p \quad \stackrel{\text{def}}{\Longleftrightarrow} \quad p \in L(s)$

$s \models \neg\Phi \quad \stackrel{\text{def}}{\Longleftrightarrow} \quad \neg(s \models \Phi)$

$s \models \Phi_1 \wedge \Phi_2 \quad \stackrel{\text{def}}{\Longleftrightarrow} \quad (s \models \Phi_1) \wedge (s \models \Phi_2)$

$s \models \exists\varphi \quad \stackrel{\text{def}}{\Longleftrightarrow} \quad \sigma \models \varphi$ pour un certain chemin infini de $\mathcal{T}$ débutant en s

$s \models \forall\varphi \quad \stackrel{\text{def}}{\Longleftrightarrow} \quad \sigma \models \varphi$ pour tout chemin infini de $\mathcal{T}$ débutant en s

Pour tout $\sigma \in (2^{AP})^\omega$, nous disons que:

$$\sigma \models \mathsf{X}\Phi \quad \stackrel{\text{def}}{\Longleftrightarrow} \quad \sigma(1) \models \Phi$$
$$\sigma \models \Phi_1 \;\mathsf{U}\; \Phi_2 \quad \stackrel{\text{def}}{\Longleftrightarrow} \quad \exists j \geq 0 : [(\sigma(j) \models \Phi_2) \wedge (\forall 0 \leq i < j : \sigma(i) \models \Phi_1)]$$

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

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

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

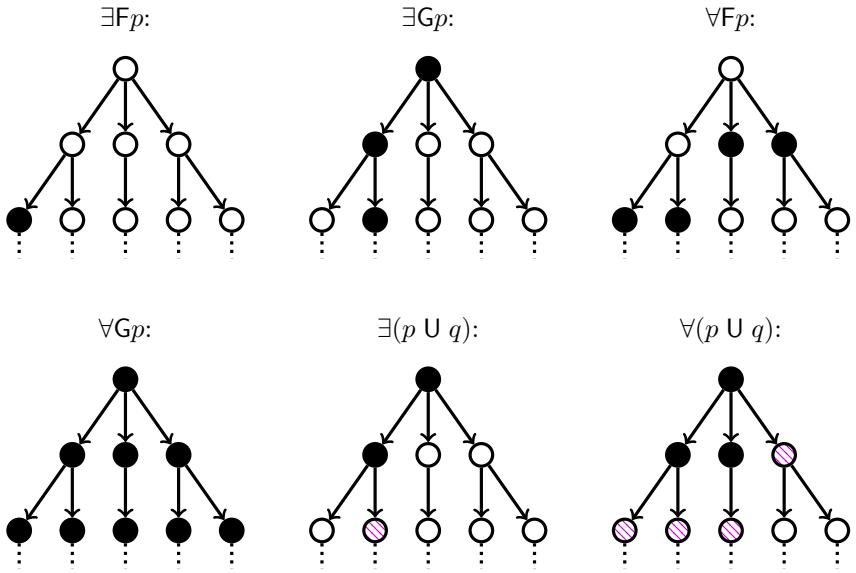


FIGURE 5.2 – Illustration de la sémantique de la logique linéaire arborescente. Chaque arbre correspond à l'arbre de calcul d'une structure de Kripke. Les états pleins (en noir) satisfont p et les états hachurés (en magenta) satisfont q.

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

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

5.4 Propriétés d'un système

Soient $\mathcal{T} = (S, \to, I, AP, L)$ 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:

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

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

$$\mathcal{T} \models \Phi \iff^{\text{def}} \forall s_0 \in I \; s_0 \models \Phi.$$

Les exemples 6.3 et 6.7 du manuel [BK08] donnent des propriétés CTL satisfaites par certaines structures de Kripke.

5.5 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. Autrement dit, 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 de propriétés.

5.5.1 Distributivité

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

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

Notons que

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

5.5.2 Dualité

La négation interagit de la façon suivante avec les opérateurs temporels quantifiés:

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

5.5.3 Idempotence

Les opérateurs temporels quantifiés satisfont les règles d'idempotence suivantes:

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

6

Vérification algorithmique de formules CTL

Dans ce chapitre, nous présentons une approche afin d'automatiser la vérification de formules CTL. Cette approche consiste à déterminer si $\mathcal{T} \models \Phi$ en calculant $[\![\Phi']\!]$ récursivement pour chaque sous-formule Φ' de Φ, puis en testant $I \subseteq [\![\Phi]\!]$. Par exemple, pour la formule $\Phi = \exists \mathsf{X} p \wedge \exists (q \,\mathsf{U}\, \exists \mathsf{G} \neg r)$, nous calculons 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. $[\![\Phi]\!]$.

6.1 Forme normale existentielle

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

6.1.1 Syntaxe

Une formule CTL est en *forme normale existentielle* si elle peut être dérivée à partir de la grammaire suivante:

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

6.1.2 Mise en forme normale

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

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

6.2 Calcul de $[\![\Phi]\!]$

Le calcul de $[\![\Phi]\!]$ peut s'effectuer récursivement en utilisant la caractérisation suivante des ensembles d'états:

$$\begin{aligned}
[\![\mathit{vrai}]\!] &= S, \\
[\![p]\!] &= \{s \in S : p \in L(s)\} \\
[\![\Phi_1 \wedge \Phi_2]\!] &= [\![\Phi_1]\!] \cap [\![\Phi_2]\!] \\
[\![\neg\Phi]\!] &= S \setminus [\![\Phi]\!] \\
[\![\exists \mathsf{X}\Phi]\!] &= \{s \in S : \mathrm{Post}(s) \cap [\![\Phi]\!] \neq \emptyset\} \\
[\![\exists \mathsf{G}\Phi]\!] &= \text{plus grand ensemble } T \subseteq S \text{ tel que} \\
&\qquad T \subseteq [\![\Phi]\!] \text{ et } s \in T \implies \mathrm{Post}(s) \cap T \neq \emptyset \\
[\![\exists(\Phi_1\ \mathsf{U}\ \Phi_2)]\!] &= \text{plus petit ensemble } T \subseteq S \text{ tel que} \\
&\qquad [\![\Phi_2]\!] \subseteq T \text{ et } (s \in [\![\Phi_1]\!] \wedge \mathrm{Post}(s) \cap T \neq \emptyset) \implies s \in T
\end{aligned}$$

Les cinq premiers cas s'implémentent assez directement. Les deux derniers cas peuvent être implémentés tel que décrit aux algorithme 3 et 4.

Algorithme 3 : Algorithme qui calcule $[\![\exists \mathsf{G}\Phi]\!]$ à partir de $[\![\Phi]\!]$.

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

$T \leftarrow [\![\Phi]\!]$ `// États pouvant satisfaire` $\exists \mathsf{G}\Phi$

`// Raffiner` T
tant que $\exists s \in T : \mathrm{Post}(s) \cap T = \emptyset$ **faire**
  retirer s **de** T
retourner T

Algorithme 4 : Algorithme qui calcule $[\![\exists(\Phi_1 \mathsf{U} \Phi_2)]\!]$ à partir de $[\![\Phi_1]\!]$ et $[\![\Phi_2]\!]$.

Entrées : ensembles $[\![\Phi_1]\!]$ et $[\![\Phi_1]\!]$
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 $\exists s \in C : \mathrm{Post}(s) \cap T \neq \emptyset$ **faire**
  ajouter s **à** T
  retirer s **de** C
retourner T

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 par rapport au nombre d'états de $\mathcal{T}$ et la taille de φ. Toutefois, les algorithmes de vérification manipulent des ensembles d'états qui peuvent être gigantesques. 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 de façon *symbolique*. Une telle représentation permet de réduire la quantité de données tout en offrant de bonnes propriétés algorithmiques.

Dans la majorité du chapitre, nous suivons de près l'excellente présentation de [And98].

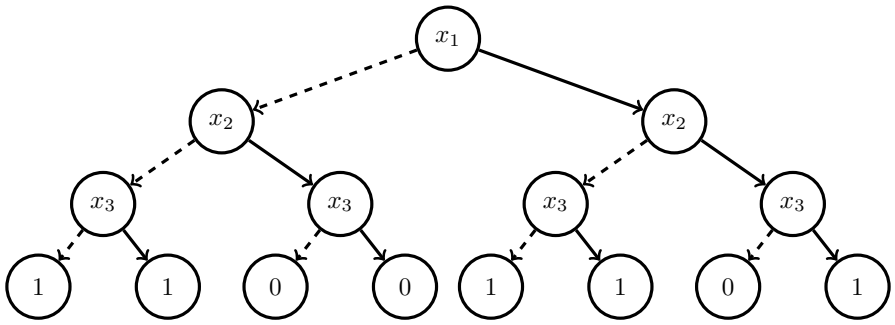


FIGURE 7.1 – Arbre de décision de l'expression booléenne $(x_1 \vee \neg x_2) \wedge (\neg x_2 \vee x_3)$. Une arête tiretée (resp. pleine) sortant d'un sommet x_i indique la décision où la variable x_i prend la valeur *faux* (resp. *vrai*). Les sommets 0 et 1 correspondent respectivement aux valeurs 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 booléenne $(x_1 \vee \neg x_2) \wedge (\neg x_2 \vee x_3)$, 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 les arbres de décision binaire 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 $x_1 = x_2 = \mathit{faux}$, le résultat est *vrai* indépendamment de la valeur de x_3.

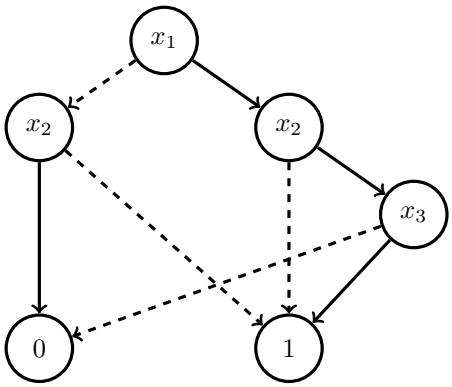


FIGURE 7.2 – Diagramme de décision binaire de l'expression $(x_1 \vee \neg x_2) \wedge (\neg x_2 \vee x_3)$ obtenu à partir de l'arbre de décision de la figure 7.1.

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

Définition 1. Un *diagramme de décision binaire (réduit et ordonné)*, abrégé par *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:

$$u \to v \implies var(u) < var(v);$$

— chaque sommet est unique, autrement dit:

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

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

Chaque sommet u d'un BDD B à n variables représente une expression booléenne $f_u \colon \{0,1\}^n \to \{0,1\}$. Ainsi, un BDD représente *plusieurs* expressions booléennes à la fois.

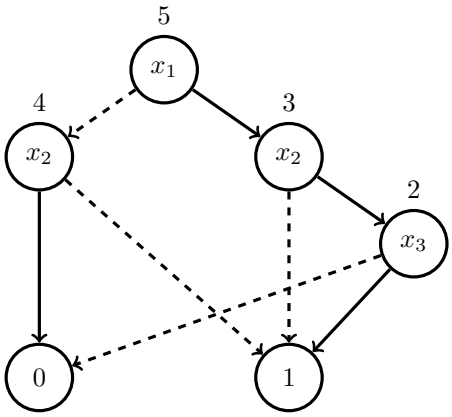


FIGURE 7.3 – Exemple d'un BDD sur trois variables $x_1 < x_2 < x_3$.

Par 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:

Lemma 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 $f_u = t$.*

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 où chaque entrée indique cet information. Par exemple, le BDD de la figure 7.3 est représenté par la table suivante:

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.

L'ajout d'un nouveau sommet u tel que $var(u) = x_i$, $lo(u) = \ell$ et $hi(u) = h$ s'effectue à l'aide de l'algorithme 5.

Algorithme 5 : Algorithme de création d'un sommet de BDD.

```
make(i, ℓ, h):
    si ℓ = h alors retourner ℓ
    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 ← nouveau nombre
        ajouter u : (x_i, ℓ, h) au BDD
        retourner u
```

Nous supposons qu'il est possible d'obtenir $(var(u), lo(u), hi(u))$ à partir de u en temps constant, et inversement qu'il est possible de déterminer l'identificateur d'un sommet étiqueté par (x_i, ℓ, h) en temps constant. En pratique, cette hypothèse peut être satisfaite à l'aide de tables de hachage.

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 $t[b/x_i] = t$ 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 $(x_1 \wedge x_2)[0/x_1] = faux$.

Étant donné une expression booléenne t, l'unique sommet représentant t se calcule à l'aide de l'algorithme 6. La figure 7.4 montre les sommets obtenus

Algorithme 6 : Algorithme de conversion d'une expression booléenne vers un BDD.

```
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)
```

lors de l'appel de `build`$(x_1 \vee \neg x_2)$ (cyan) suivi de l'appel `build`$(\neg x_2 \vee x_3)$ (magenta). Les sommets 3 et 5 représentent respectivement les expressions $x_1 \vee \neg x_2$ et $\neg x_2 \vee x_3$. Notons que des appels subséquents à `build` pourraient réutiliser certains sommets existants.

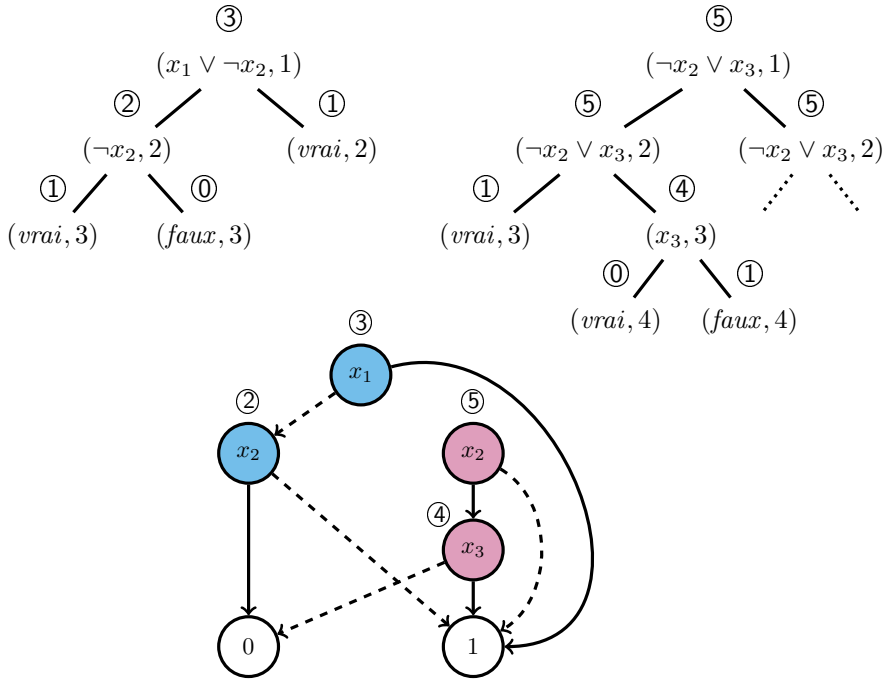


FIGURE 7.4 – *Haut droite*: arbre d'appel de `build`$(x_1 \vee \neg x_2)$; *haut gauche*: arbre d'appel de `build`$(\neg x_2 \vee x_3)$; où $x_1 < x_2 < x_3$, et chaque nombre encerclé indique la valeur retournée par l'appel. *Bas*: BDD résultant des deux appels.

7.3 Manipulation de BDD

7.3.1 Opérations logiques binaires

Étant donné deux sommets u_1 et u_2, il est possible de 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 7.

Algorithme 7 : Algorithme de calcul d'une opération binaire $\bigcirc$ de deux sommets d'un BDD.

$\texttt{apply}_{\bigcirc}(u_1, u_2)$*:*
  $v_1, \ell_1, h_1 \leftarrow var(u_1), lo(u_1), hi(u_1)$
  $v_2, \ell_2, h_2 \leftarrow var(u_2), lo(u_2), hi(u_2)$
  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**
    retourner $\texttt{make}(v_1, \texttt{apply}(\ell_1, u_2), \texttt{apply}(h_1, u_2))$
  sinon si $v_1 > v_2$ **alors**
    retourner $\texttt{make}(v_2, \texttt{apply}(u_1, \ell_2), \texttt{apply}(u_1, h_2))$
  sinon
    retourner $\texttt{make}(v_1, \texttt{apply}(\ell_1, \ell_2), \texttt{apply}(h_1, h_2))$

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 $\texttt{apply}_{\wedge}(3,5)$ génère le nouveau sommet 6, tel que $f_6 = f_3 \wedge f_5$, illustré à la figure 7.5.

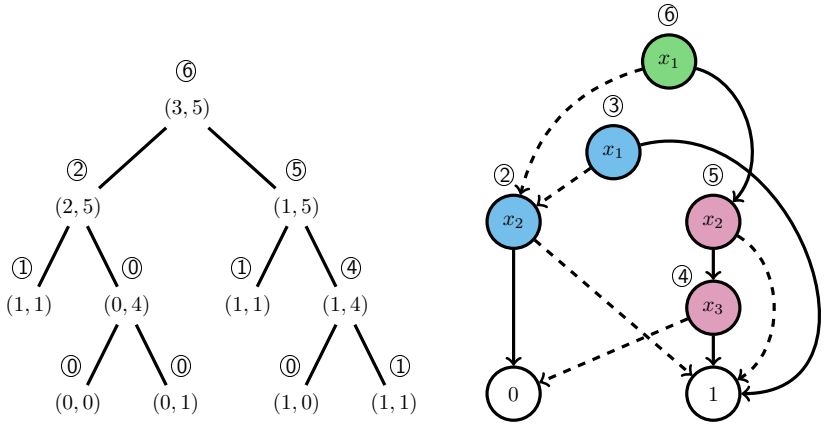


FIGURE 7.5 – *Gauche*: arbre d'appel de $\texttt{apply}_{\wedge}(3,5)$ sur le BDD de la figure 7.4; *droite*: BDD résultant où $f_6 = f_3 \wedge f_5$.

7.3.2 Restriction et quantification existentielle

Considérons un sommet u d'un BDD, une variable x_i et une valeur booléenne $b \in \{0,1\}$. Il est possible de calculer un sommet v tel que $f_v = f_u[b/x_i]$ à l'aide de l'algorithme 8.

Algorithme 8 : Algorithme de restriction d'une variable.

```
restrict(u, i, b):
    restrict'(u):
        si var(u) > x_i alors
            retourner u
        sinon si var(u) < x_i alors
            retourner make(var(u), lo(u), hi(u))
        sinon si b = 0 alors
            retourner restrict'(lo(u))
        sinon
            retourner restrict'(hi(u))

    retourner restrict'(u)
```

L'algorithme de restriction s'avère utile pour quantifier existentiellement des variables. En effet, étant une 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, tel qu'illustré à l'algorithme 9, deux appels à `restrict` et un appel à `apply` permettent d'implémenter la quantification existentielle.

Algorithme 9 : Algorithme de quantification existentielle.

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

7.4 Vérification CTL à l'aide de BDD

Les opérations vues 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. Les états de $\mathcal{T}$ 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:

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

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

Cherchons à déterminer si $\mathcal{T} \models \exists \mathsf{X} p$ à l'aide de BDD.

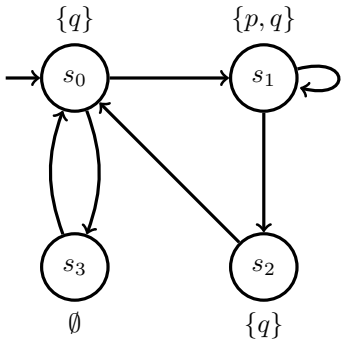


FIGURE 7.6 – Une structure de Kripke.

Soit $M \subseteq S$. Observons que:

$$\begin{aligned} \text{Post}(M) &= \{t : \exists s \ (s,t) \in \rightarrow \cap \ (M \times S)\}, \\ \text{Pre}(M) &= \{s : \exists t \ (s,t) \in \rightarrow \cap \ (S \times M)\}. \end{aligned}$$

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

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

Les ensembles $\rightarrow$ et $S \times [\![p]\!] = \{0001, 0101, 1001, 1101\}$ sont représentés respectivement par les sommets 8 (magenta) et 3 (cyan) du BDD de la figure 7.7. Ces sommets peuvent être obtenus à l'aide de deux appels à `build`. L'ensemble $\rightarrow \cap \ (S \times [\![p]\!])$ est représenté par le sommet 9 (vert) de la figure 7.7. Ce sommet est obtenu par un appel à $\texttt{apply}_{\wedge}(3, 8)$.

Nous devons maintenant calculer le résultat de la quantification existentielle $\ll \exists t \gg$. 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 (orange) 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 que

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

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 $f_u \wedge f_v$;

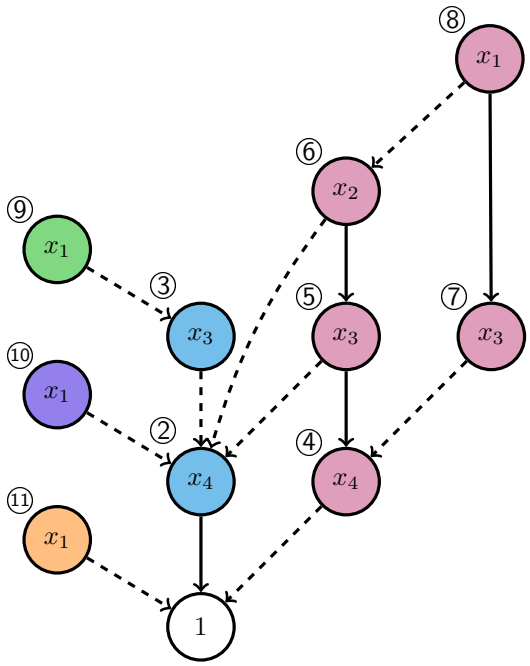


FIGURE 7.7 – BDD partiel obtenu lors de la vérification de $\mathcal{T} \models \exists \mathsf{X} p$. Les arêtes vers le sommet 0 sont omises par soucis de lisibilité.

— tester si $f_w \equiv \emptyset$.

Grâce au 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.

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`, peuvent être implémentées afin de fonctionner en temps polynomial:

Opération	Complexité dans le pire cas
`make`(i, ℓ, h)	$O(1)$
`build`(t)	$O(2^n)$
`apply`(u_1, u_2)	$O(\lvert u_1\rvert \cdot \lvert u_2\rvert)$
`restrict`(u, i, b)	$O(\lvert u\rvert)$
`exists`(u, i)	$O(\lvert u\rvert^2)$

Notons que cette complexité polynomiale nécessite un usage de la mémoïsation afin d'éviter de recalculer des valeurs déjà calculées.

8

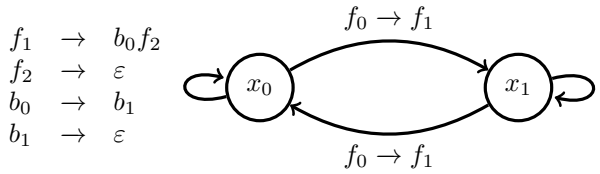
Systèmes avec récursion

Dans ce chapitre, nous voyons comment vérifier des programmes qui effectuent des appels récursifs. Considérons le programme suivant constitué de deux fonctions et d'une variable booléenne globale x:

```
      bool x ∈ {faux,vrai}

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

L'ensemble des états accessibles d'un tel programme est à priori non borné. Ainsi, nous ne pouvons pas utiliser les méthodes de vérification présentées jusqu'ici. Le programme peut être modélisé à l'aide du diagramme suivant:



Les états x_0 et x_1 indiquent que $x = \mathit{faux}$ et $x = \mathit{vrai}$ respectivement. Les transitions représentent le flot du programme. Par exemple, $f_1 \to b_0 f_2$ indique qu'à partir de l'étiquette f_1 on se déplace à l'étiquette b_0 et on effectue un retour vers l'étiquette f_2.

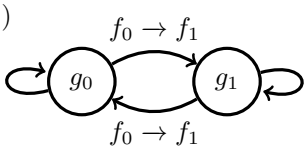
Ce type de modélisation peut également être étendu aux variables locales. Par exemple, considérons le programme suivant:

```
bool g ∈ {faux,vrai}

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

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

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



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

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? 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 d'*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 $((p,a),(q,u)) \in \Delta$ indique que dans l'état p, il est possible de dépiler a, d'empiler u et de se déplacer à l'état q. Dans ce cas, nous écrivons

$\langle p, av\rangle \rightarrow \langle q, uv\rangle$. Plus généralement, nous écrivons $\langle p, w\rangle \xrightarrow{*} \langle q', w'\rangle$ s'il est possible de passer de la configuration $\langle p, w\rangle$ à la configuration $\langle p', w'\rangle$ en zéro, une ou plusieurs transitions.

Soit C un ensemble de configurations. Nous définissons l'ensemble des *prédecesseurs* de C par:

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

8.2 Calcul des prédecesseurs

L'ensemble $\mathrm{Pre}^*(C)$ permet de déterminer certaines propriétés d'un système à pile. Par exemple $C = \{\langle g_i, \perp w\rangle : i \in \{0,1\}, w \in \Gamma^*\}$ représente l'ensemble des configurations où l'assertion est enfreinte dans le deuxième programme ci-dessus. Ainsi, le programme contient un bogue ssi $\langle g_0, m_0\rangle \in \mathrm{Pre}^*(C)$ ou $\langle g_1, m_0\rangle \in \mathrm{Pre}^*(C)$.

Nous donnons une procédure afin de calculer $\mathrm{Pre}^*(C)$. Puisque C et $\mathrm{Pre}^*(C)$ 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 $\mathcal{A}$.

Nous disons qu'un $\mathcal{P}$-automate *accepte* une configuration $\langle p, w\rangle$ si $p \in P$, $q \in F$ et $p \xrightarrow{w} q$. L'ensemble des configurations acceptées par $\mathcal{A}$ est dénoté par $\mathit{Conf}(\mathcal{A})$.

Proposition 4 ([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 $\mathit{Conf}(\mathcal{B}) = \mathrm{Pre}^*(\mathit{Conf}(\mathcal{A}))$.*

Algorithme 10 : Algorithme de calcul de prédecesseurs.

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 $\mathit{Conf}(\mathcal{B}) = \mathrm{Pre}^*(\mathit{Conf}(\mathcal{A}))$

$\mathcal{B}, c \leftarrow \mathcal{A}, \mathit{vrai}$

tant que c **faire**
  $c \leftarrow \mathit{faux}$
  pour $((p, a), (p', w)) \in \Delta$ **faire**
    pour $q \in Q$ **faire**
      si $p' \xrightarrow{w} q$ *dans* $\mathcal{B}$ **alors**
        ajouter (p, a, q) à $\mathcal{B}$
        $c \leftarrow \mathit{vrai}$

retourner $\mathcal{B}$

Le $\mathcal{P}$-automate $\mathcal{B}$ de la proposition 4 se calcule en ajoutant itérativement à $\mathcal{A}$ de nouvelles transitions à l'aide de la règle suivante:

> Si $\langle p, a\rangle \to \langle p', w\rangle$ dans $\mathcal{P}$ et $p' \xrightarrow{w} q$ dans $\mathcal{B}$, alors la transition (p, a, q) est ajoutée à $\mathcal{B}$.

Autrement dit, $\mathcal{B}$ est obtenu à partir de $\mathcal{A}$ en appliquant la règle ci-dessus jusqu'à saturation. Cet algorithme est illustré à l'algorithme 10.

Appliquons l'algorithme 10 au système à pile $\mathcal{P}$ et au $\mathcal{P}$-automate $\mathcal{A}$ de la figure 8.1. Cinq nouvelles transitions sont ajoutées à $\mathcal{A}$:

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

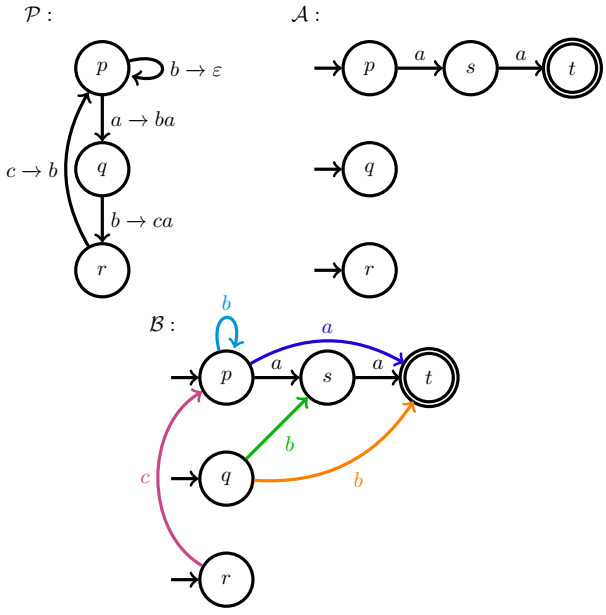


FIGURE 8.1 – Exemple de $\mathcal{P}$-automate $\mathcal{B}$ calculé à partir d'un système à pile $\mathcal{P}$ et d'un $\mathcal{P}$-automate $\mathcal{A}$.

8.3 Vérification à l'aide de système à pile

Considérons le programme suivant:

```
        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 peut être modélisé par le système à pile $\mathcal{P}$ illustré à la figure 8.2.

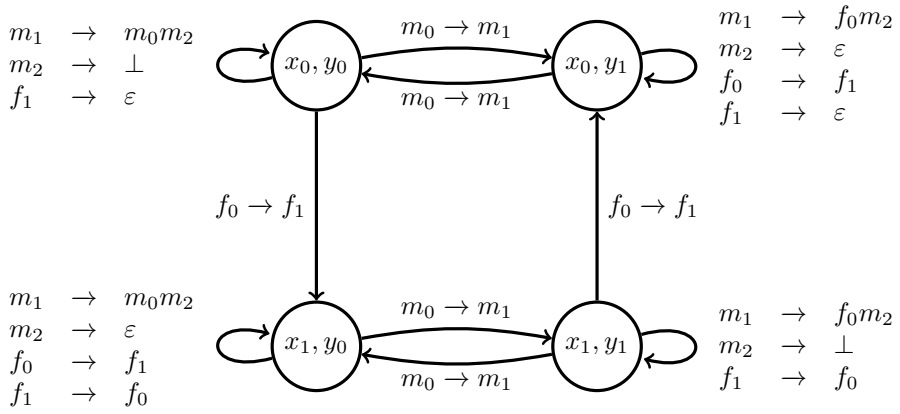


FIGURE 8.2 – Système à pile modélisant un programme.

Afin de déterminer si l'assertion peut être enfreinte, il suffit de

— calculer un $\mathcal{P}$-automate $\mathcal{B}$ acceptant $\mathrm{Pre}^*(\mathit{Conf}(\mathcal{A}))$, où $\mathcal{A}$ est le $\mathcal{P}$-automate illustré à la figure 8.3;

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

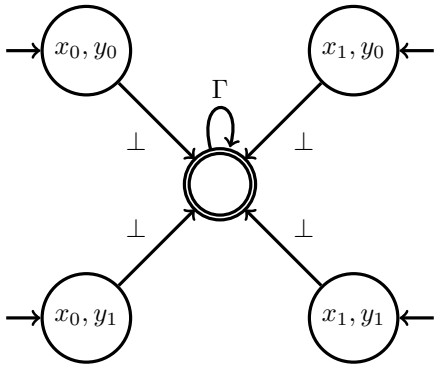


FIGURE 8.3 – $\mathcal{P}$-automate $\mathcal{A}$ représentant les configurations de $\mathcal{P}$ où l'assertion est enfreinte.

9

Systèmes infinis

Dans ce chapitre, nous voyons comment vérifier des systèmes pouvant posséder une quantité infinie de configurations à l'aide de réseaux de Petri.

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 (*transitions*);
- $F \colon ((P \times T) \cup (T \times P)) \to \mathbb{N}$ (*fonction de flot*).

Par exemple, le réseau de Petri suivant est illustré à la figure 9.1:

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

Un *marquage* est un vecteur $M \in \mathbb{N}^P$. Une transition $t \in T$ est *déclenchable* dans M si $M(p) \geq F(p, t)$ pour tout $p \in P$. Si t est déclenchable, alors $M \xrightarrow{t} M'$ où $M'(p) \stackrel{\text{def}}{=} M(p) - F(p, t) + F(t, p)$ pour tout $p \in P$. Nous écrivons $M \to M'$ s'il existe $t \in T$ telle que $M \xrightarrow{t} M'$. Nous écrivons $M \xrightarrow{*} M'$ si $M = M'$ ou s'il existe une séquence de marquages et de transitions tels que:

$$M = M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \cdots \xrightarrow{t_k} M_k = M'.$$

Par exemple, dans le réseau de la figure 9.1, nous avons $(1, 1) \xrightarrow{*} (3, 3)$ puisque $(1, 1) \xrightarrow{s} (0, 3) \xrightarrow{t} (4, 1) \xrightarrow{s} (3, 3)$.

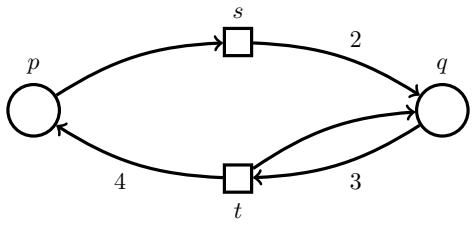


FIGURE 9.1 – Exemple de réseau de Petri. Les places et les transitions sont respectivement représentées par des cercles et des carrés. Les arcs représentent la fonction de flot.

L'ensemble des *successeurs* et *prédecesseurs* d'un marquage M est respectivement dénoté:

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

Nous écrivons $M \geq M'$ si $M(p) \geq M'(p)$ pour tout $p \in P$. 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.

Supposons que nous cherchons à déterminer si plusieurs processus peuvent atteindre la ligne p_2. Cela est équivalent à déterminer s'il existe un marquage M' tel que:

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

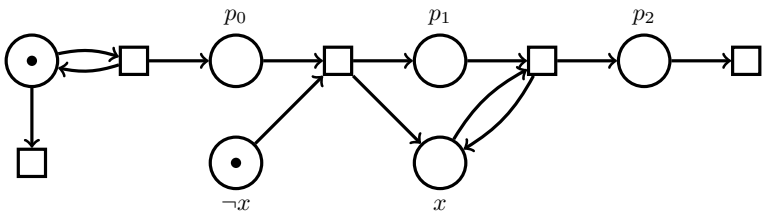


FIGURE 9.2 – Réseaux de Petri modélisant l'exécution du processus `proc`. La place tout à gauche permet de lancer un nombre arbitraire de processus et de potentiellement cesser d'en lancer.

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 que nous cherchons à déterminer si le programme peut se terminer avec $x =$ `vrai`. Cela est équivalent à déterminer si:

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

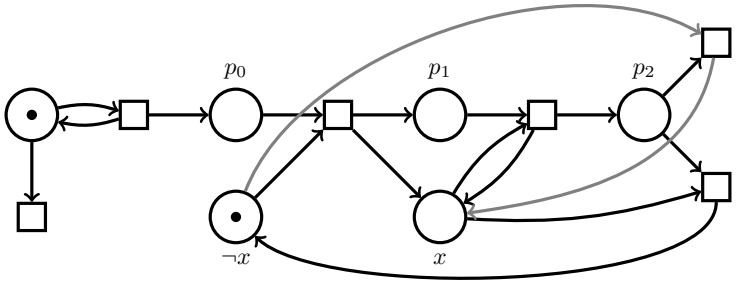


FIGURE 9.3 – Réseau de Petri modélisant l'exécution du processus `proc2`.

9.3 Vérification

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

PROBLÈME D'ACCESSIBILITÉ

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

QUESTION: $M \xrightarrow{*} M'$?

PROBLÈME DE COUVERTURE

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

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

Ces deux problèmes sont décidables; autrement dit, ils sont tous deux solubles à l'aide d'un algorithme. Nous mettons l'emphase sur le problème de couverture et présentons des algorithmes permettant de résoudre ce problème.

9.3.1 Problème de couverture

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

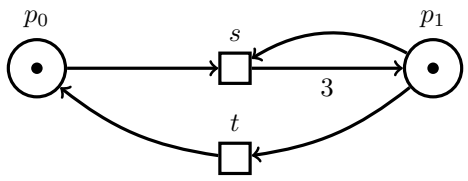


FIGURE 9.4 – Autre exemple de réseau de Petri.

Par exemple, considérons le réseau de Petri illustré à la figure 9.4. Supposons que nous cherchons à déterminer si $M = (1, 1)$ peut couvrir un marquage 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.

Graphes de couverture. 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 $\mathbb{N}_\omega \stackrel{\text{def}}{=} \mathbb{N} \cup \{\omega\}$ où

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

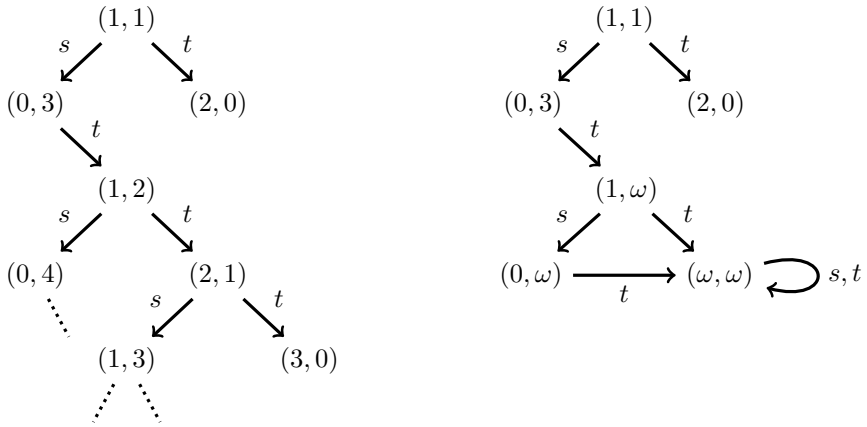


FIGURE 9.5 – Graphe d'accessibilité (gauche) et graphe de couverture (droite) du réseau de Petri de la figure 9.4 à partir du marquage $(1, 1)$.

Un *marquage étendu* est un vecteur $M \in \mathbb{N}_\omega^P$. 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 $(1, 1) \xrightarrow{*} (1, 2)$ et $(1, 2) \geq (1, 1)$. Ainsi, en itérant la séquence st, nous pouvons faire croître arbitrairement la deuxième place. Nous remplaçons donc ce marquage par le marquage étendu $(1, \omega)$. Ainsi, en inspectant les ancêtres d'un marquage lors de son ajout, nous obtenons un graphe *fini* qui capture la forme des marquages couvrables. La figure 9.6 présente un autre exemple de cette procédure.

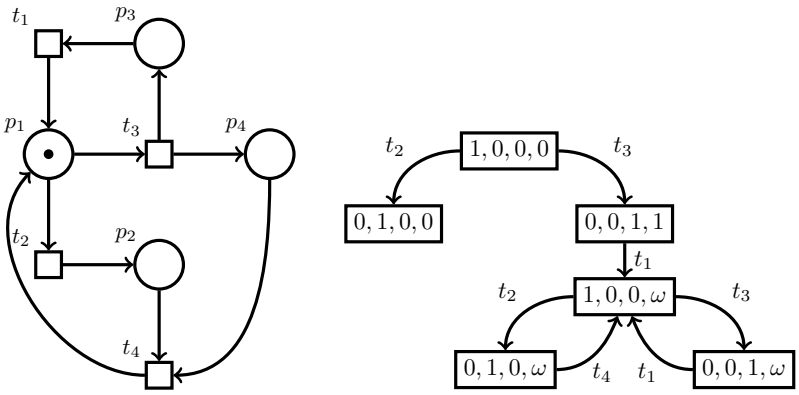


FIGURE 9.6 – *Gauche:* un réseau de Petri; *droite:* un graphe de couverture à partir du marquage $(1, 0, 0, 0)$.

Algorithme 11 : Algorithme de calcul de graphe de couverture.

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

```
cover(N, M):
    V ← ∅                                          /* Sommets */
    E ← ∅                                          /* Arcs */
    W ← {M}

    tant que W ≠ ∅ faire
        M' ← retirer de W
        pour t ∈ T déclenchable en M' faire
            M'' ← marquage tel que M' -t-> M''
            accel(M'')
            si M'' ∉ V alors
                ajouter M'' à V et W
            ajouter (M', M'') à E
    retourner (V, E)

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

La procédure de calcul d'un graphe de couverture est décrite à l'algorithme 11. La proposition suivante explique comment déterminer si un marquage M' est couvrable à l'aide d'un graphe de couverture:

Proposition 5. *Soit $\mathcal{N} = (P, T, F)$ un réseau de Petri, et soient deux marquages $M, M' \in \mathbb{N}^P$. Soit G un graphe de couverture calculé par $\texttt{cover}(\mathcal{N}, M)$. Le marquage M' est couvrable à partir de M si et seulement si G possède un marquage M'' tel que $M'' \geq M'$.*

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

Proposition 6. *L'algorithme 11 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 $M_0 \to M_1 \to \cdots$.

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

$$[\![M_i]\!] \stackrel{\text{def}}{=} \{p \in P : M_i(p) = \omega\}.$$

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

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 $M \in \mathbb{N}^P$. La *clôture par le haut* de M est l'ensemble de marquages $\uparrow M \stackrel{\text{def}}{=} \{M' \in \mathbb{N}^P : M' \geq M\}$. La *clôture par le haut* d'un ensemble $X \subseteq \mathbb{N}^P$ est l'ensemble:

$$\uparrow X \stackrel{\text{def}}{=} \bigcup_{M \in X} \uparrow M.$$

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 $x, y \in B$ tels que $x \geq y$ et $x \neq y$. Autrement dit, B est minimale si tous ses éléments sont incomparables. La figure 9.7 donne un exemple d'ensemble clos par le haut et de base minimale.

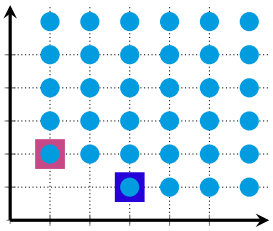


FIGURE 9.7 – Illustration d'un ensemble clos par le haut. Sa base minimale $\{(1,2),(3,1)\}$ est représentée par des carrés.

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

Proposition 7. *Soit $M' \in \mathbb{N}^P$. L'ensemble des marquages qui peuvent couvrir M' est égal à $\uparrow \mathrm{Pre}^*(\uparrow M')$.*

Démonstration. $\subseteq$) Soit M un marquage qui peut couvrir M'. Par définition de couverture, il existe un marquage M'' tel que $M \xrightarrow{*} M''$ et $M'' \geq M'$. Ainsi,

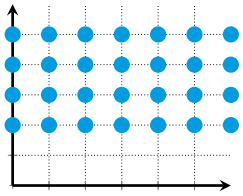
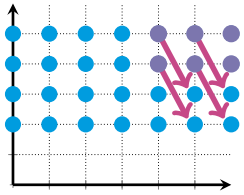
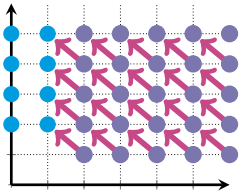
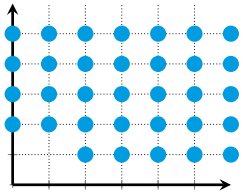
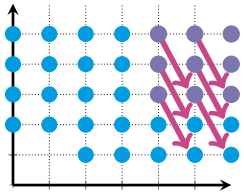
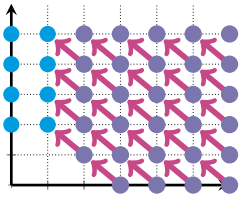
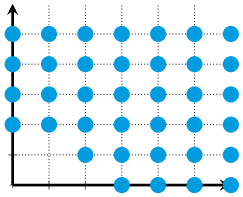
$$\begin{aligned} M &\in \mathrm{Pre}^*(M'') && \text{(par définition de Pre}^*\text{)} \\ &\subseteq \mathrm{Pre}^*(\uparrow M') && \text{(car } M'' \in \uparrow M'\text{)} \\ &\subseteq \uparrow \mathrm{Pre}^*(\uparrow M') && \text{(par l'identité } X \subseteq \uparrow X\text{)} \end{aligned}$$

$\supseteq$) Soit $M \in \uparrow \mathrm{Pre}^*(\uparrow M')$. Il existe des marquages K et M''' tels que

$$\begin{array}{ccc} M & & \\ \vee| & & \\ K & \xrightarrow{*} & M'' \\ & & \vee| \\ & & M' \end{array}$$

Ainsi, par monotonicité, il existe un marquage M''' tel que $M \xrightarrow{*} M'''$ et $M''' \geq M'' \geq M'$. Nous concluons donc que M peut couvrir M'. □

L'algorithme arrière calcule ainsi $\uparrow \mathrm{Pre}^*(\uparrow M')$. Par exemple, considérons le réseau de Petri illustré du côté gauche de la figure 9.8 avec $M' = (0, 2)$. L'algorithme arrière débute avec l'ensemble $\uparrow M'$, puis calcule itérativement Pre de chacun de ses marquages jusqu'à ce que l'ensemble se stabilise:

Itér.	Prédecesseurs sous t_1	Prédecesseurs sous t_2	Ensemble actuel
0	—	—	
1			
2			
3	ensemble inchangé		

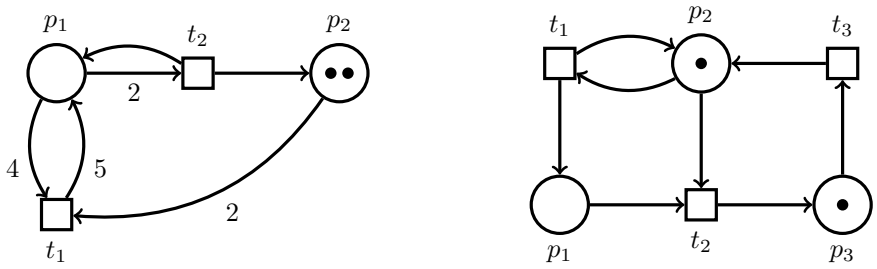


FIGURE 9.8 – Deux réseaux de Petri.

Puisque les ensembles clos par le haut son 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 prédecesseurs de tous les marquages d'un ensemble clos par le haut.

Le premier point est rendu possible par l'observation suivante:

Proposition 8. *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 puisque:

$$M \in \uparrow B \iff \text{il existe } K \in B \text{ tel que } M \geq K.$$

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. Plus formellement, soit $t \in T$ une transition et $M \in \mathbb{N}^P$ un marquage. Le *plus petit marquage pouvant couvrir M en déclechant t* est le marquage M_t tel que

$$M_t(p) \stackrel{\text{def}}{=} \max\left(F(p,t), M(p) + F(p,t) - F(t,p)\right) \qquad \text{pour tout } p \in P.$$

L'algorithme complet est décrit à l'algorithme 12. Reconsidérons le réseau de Petri illustré du côté gauche de la figure 9.8 avec $M' = (0,2)$. Nous exécutons cette fois l'algorithme arrière en manipulant directement une base:

Itér.	**Base B**	**Prédecesseurs**
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

Comme exemple supplémentaire, nous exécutons l'algorithme arrière sur le réseau de Petri illustré du côté droit de la figure 9.8 avec $M' = (0,1,1)$:

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	$\{(0,1,1),(1,2,0),(0,0,2)\}$	$(1,2,0)_{t_1} = (0,2,0)$ $(0,0,2)_{t_1} = (0,1,2)$ $(1,2,0)_{t_2} = (2,3,0)$ $(0,0,2)_{t_2} = (1,1,1)$ $(1,2,0)_{t_3} = (1,1,1)$ $(0,0,2)_{t_3} = (0,0,3)$
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

Algorithme 12 : Algorithme arrière.

Entrées : Réseau de Petri $\mathcal{N} = (P,T,F)$ et $M' \in \mathbb{N}^P$
Sorties : La base minimale de $\uparrow \mathrm{Pre}^*(\uparrow M')$

```
arriere(N, M'):
    B' ← {M'}
    faire
        B ← B'                                   /* Base actuelle */
        pour M ∈ B faire
            pour t ∈ T faire
                ajouter M_t à B'
        minimiser(B')   /* Retirer les marquages redondants */
    tant que B' ≠ B
    retourner B

minimiser(X):
    pour x ∈ X faire
        pour y ∈ X faire
            si x ≤ y ∧ x ≠ y alors retirer y de X
```

Bibliographie

[And98] Henrik Reif Andersen. An introduction to binary decision diagrams, 1998.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[EHRS00] Javier Esparza, David Hansel, Peter Rossmanith, and Stefan Schwoon. Efficient algorithms for model checking pushdown systems. In *Proc.* 12^{th} *International Conference on Computer Aided Verification (CAV)*, pages 232–247, 2000.

[Esp19] Javier Esparza. Automata theory: An algorithmic approach, 2019.

Index