École Polytechnique
Majeure 2
Spécialité Informatique

Vérification des Systèmes Réactifs Temps-Réel

Jean-Pierre Jouannaud
Professeur
Université Paris-Sud
École Polytechnique

Adresses de l’auteur :

LIX
Bureau 101
École Polytechnique
F-91128 Palaiseau CEDEX

Email : Jean-Pierre.Jouannaud@lix.polytechnique.fr
Tél : 01 69 33 40 70
Fax : Tél : 01 69 33 30 14
URL : http://www.lix.polytechnique.fr/

&

LRI, Bâtiment 490
bureau 128
Université Paris-Sud
F-91405 Orsay CEDEX

URL : http://www.lri.fr/~jouannau/

Table des matières

Chapitre 1

Introduction

La science du logiciel est jeune, encore balbutiante. Les besoins ayant crû plus vite que les savoirs, l'industrie a fait face aux problèmes en concentrant d'importantes forces de travail pour la réalisation de gros projets. Aujourd'hui, un projet logiciel se gère encore souvent comme la construction d'une cathédrale au moyen âge, ou une ascension himalayenne il y a un demi siècle. Les même causes entraînant les même effets, la complexité croissante des applications informatiques a malheureusement pour corollaire l'existence de comportements anormaux aux conséquences parfois irréparables : panne du réseau téléphonique aux USA en 89, rappel de centaines de milliers de Pentium par INTEL en 96, (discrète) panne du réseau de téléphonie mobile de France Télécom en région parisienne en janvier 98, destruction de la première fusée Ariane 5 en 99, etc. Pour mesurer la difficulté du problème, il faut savoir que les logiciels envahissent *tous* les objets de notre vie courante (30 lignes de code dans le programmateur d'une machine à laver), et que certains grands logiciels comptent plusieurs dizaines de millions d'instructions (plus d'un million dans un téléphone portable), et que leur conception obéit à des principes méthodologiques encore rudimentaires. La délivrance sur le marché d'un logiciel suppose bien sûr une (coûteuse) phase de tests. Mais si ces derniers se révèlent aptes à identifier une bonne part des erreurs, ils ne constituent en rien une garantie.

L'erreur est parfois interdite, comme dans le cas de l'aéronautique, de la téléchirurgie, du télépaiement par carte à puce, ou du contôle d'une centrale nucléaire. Les logiciels concernés sont dits *critiques*. La *vérification* de logiciels critiques est un domaine de recherche en plein essor, qui s'appuie sur une demande industrielle certes récente mais en forte croissance. Très souvent, ces logiciels critiques ont pour objectif le contrôle d'un processus qui communique avec son environnement par l'intermédiaire de capteurs, themomètres, signaux, claviers, etc. Ces logiciels n'ont pas pour but de calculer un résultat, mais d'assurer le fonctionnement permanent du processus contrôlé, on les apppelle des systèmes *réactifs* . Une autre caractéristique fréquente de ces systèmes critiques est que leur comportement global dépend de l'interaction de plusieurs sous-systèmes évoluant en parallèle, ce sont des systèmes *distribués*. Enfin, le paramètre temps intervient généralement de manière explicite, ce sont des systèmes *temps réel*.

Qu'ils soient critiques ou pas, c'est à la spécification et à la vérification de systèmes réactifs temps réels distribués que nous allons nous atteller. Il y a pour cela deux grandes (familles de) techniques : les méthodes de preuve, où l'ingénieur utilise un *assistant de preuve* afin de prouver que le système étudié satisfait ses spécifications ; les méthodes de vérification (en anglais, *model checking*) qui ont pour but de prouver automatiquement certaines propriétés dites de *sureté* pour les plus importantes (ou leur négation ...).

Ce cours a pour but d'aborder les différentes phases, de la conception à la vérification, de la construction de systèmes distribués réactifs temps réel.

La première étape dans l'élaboration d'un tel système est celle de la spécification. Nous étudierons donc comment spécifier ces systèmes réactifs à l'aide d'automates temporisés et de produits de

synchronisation. La théorie des automates va donc intervenir de manière essentielle dans ce cours, dont le but sera donc en fait de vérifier des systèmes réactifs modélisés sous forme d'automates.

La seconce étape consiste à spécifier les propriétés attendues d'un système réactif, essentiellement des propriétés de sureté : par exemple, la barrière doit obligatoirement se fermer un certain temps avant le passage du train. Nous étudierons comment spécifier ces propriétés par des énoncés en logique temporelle (ou plus précisément, en logique temporelle temporisée).

La troisième étape consiste à déterminer si une certaine propriété de sureté est vraie d'un certain automate. Il s'agit de la vérification proprement dite. Nous étudierons plusieurs algorithmes adaptés à certains fragments de la logique temporelle, et analyserons leur complexité.

Enfin, nous trerminerons par l'étude de plusieurs logiciels de vérification, en particulier Chronos, UPPAAL et Hytech. Dans le document, une brève introduction à ces langages est donnée au chapitre **??**, juste après l'introduction.

Le chapitre 2 consistera en un rappel sur la notion d'automate, déterministe ou non-déterministe, le langage reconnu par un automate, la déterminisation des automates non-déterministes et la minimisation des automates déterministes, et non concluerons par les propriétés de clôture Booléennes.

Dans le chapitre 3, nous examinerons comment spécifier des systèmes réactifs à l'aide d'automates. Ce sera l'occasion d'introduire de une nouvelle construction fondamentale visant à modulariser la spécification des systèmes, le produit de synchonisation. D'autres constructions, comme l'ajout de variables locales ou globales, l'utilisation de messages de synchronisation, qui sont des cas particuliers de produits de synchronisation, ont pour but de faciliter l'écriture des spécifications. Nous terminerons par le modèle des automates temporisés, pour lesquelles il existe deux types de transitions : les transitions d'états, qui sont instantanées, et les transitions temporelles, pour lesquelles le temps évolue dans un état donné. Ce modèle est beaucoup plus riche que le précédent, il permet d'aborder de vraies applications, mais ses aspects algorithmiques seront plus complexes.

Le chapitre 4 abordera un troisième sujet : la logique temporelle propositionnelle, et ses liens avec la logique du premier ordre. Nous examinerons deux sous-langages particuliers, CTL et PLTL, et donnerons des algorithmes de décision de formules de CTL et PLTL interprétées sur des automates, avant d'en étudier la complexité, linéaire pour CTL et PSPACE pour PLTL.

Le chapitre 5 consistera en un autre rappel, sur les classes de complexité P, NP, CoNP, PSPACE et NPSPACE, et les notions de complétude associées. Nous verrons deux problèmes paradigmatiques du point de vue de la complexité : SAT et QBF, prototypes de problème NP-complet et PSPACE-complet respectivement. Nous rappellerons également la notion de *complexité non-élémentaire*.

Le chapitre 6 sera consacré aux automates de Büchi, qui servent à caractériser la reconnaissabilité des mots infinis. Ces automates joueront un rôle essentiel dans le chapitre 7, qui sera consacré à la décision de certains fragments de la logique temporelle.

Comme tout cours, celui-ci fait de nombreux emprunts, en particulier aux ouvrages [1, 2, 4, 5] dont une lecture approfondie est recommandée.

Les logiciels étudiés sont en principe disponibles sur les machines de l'école, et peuvent être téléchargés aux URL suivants :

CHRONOS à l'URL `http : //www-verimag.imag.fr/TEMPORISE/chronos`
CMC à l'URL `http : //www.lsv.ens-cachan.fr/~{}fl`
HyTech à l'URL `http : //www.eecs.berkeley.edu/~tah/Hytech`
UPPAAL à l'URL `http : //www.docs.uu.se/docs/rtmv/uppaal`

Chapitre 2

Automates de mots finis

Les automates sont un outil de modélisation fondamental, qui servent à représenter des dispositifs automatiques, par exemples des systèmes réactifs, tout autant que des objets mathématiques ou physiques. Dans ce capitre, nous nous intéressons aux automates de mots finis, le cas important des mots infinis étant abordé dans un chapitre ultérieur.

2.1 Automates déterministes

Définition 2.1 *Un* automate déterministe $\mathcal{A}$ *est un triplet* (V_t, Q, T) *où*

1. V_t *est le* vocabulaire *de l'automate ;*
2. Q *est l'ensemble fini des* états *de l'automate ;*
3. $T : Q \times V_t \to Q$, *est une application partielle appellée* fonction de transition *de l'automate.*

On notera $q \xrightarrow{a} q'$ pour $T(q, a) = q'$. Lorsque T est totale, autrement dit s'il y a dans chaque état exactement une transition pour toute lettre de l'alphabet, l'automate est dit *complet*.

Définition 2.2 *Étant donné un automate déterministe* $A = (V_t, Q, T)$, *on appelle* calcul *toute suite (éventuellement vide) de transitions* $q_0 \xrightarrow{\alpha_1} q_1 \cdots q_{n-1} \xrightarrow{\alpha_n} q_n$, *aussi notée* $q_0 \xrightarrow{w} {}^* q_n$, *issue d'un état* q_0 *et atteignant un état* q_n *en ayant lu le mot* $w = \alpha_1 \cdots \alpha_n$.

On écrira souvent $q_0 \xrightarrow{w} q_n$ au lieu de $q_0 \xrightarrow{w} {}^* q_n$.

Définition 2.3 *Un automate déterministe vient avec la donnée*

- *d'un* état initial $q_0 \in Q$ *;*
- *d'un ensemble d'*états acceptants $F \subseteq Q$ *;*

On notera par $A^F_{q_0}$ *le quintuplet* (V_t, Q, q_0, F, T), *ou plus simplement* A *si* q_0 *et* F *sont donnés sans ambiguité dans le contexte.*

Un mot w *est* reconnu *par l'automate* $A^F_{q_0}$ *s'il existe un calcul issu de l'état initial* q_0 *et terminant en état acceptant après avoir lu le mot* w. *On note par* $\mathcal{L}ang(A)$ *le langage des mots reconnus par l'automate* A. *Un langage reconnu par un automate est dit* reconnaissable.

On note par $\mathcal{R}ec$ *l'ensemble des langages reconnaissables.*

On a l'habitude de dessiner les automates, en figurant les états par des cercles, en indiquant l'état initial par une flèche entrante, les états acceptants par un double cercle ou une flèche sortante, et la transition de l'état q à l'état q' en lisant la lettre α par une flèche allant de q vers q' et étiquetée par α. La figure 2.1 représente un automate (incomplet) reconnaissant le langage des entiers naturels en représentation binaire.

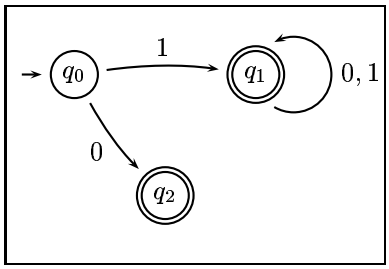


FIG. 2.1 – Automate déterministe reconnaissant les entiers naturels en numération binaire.

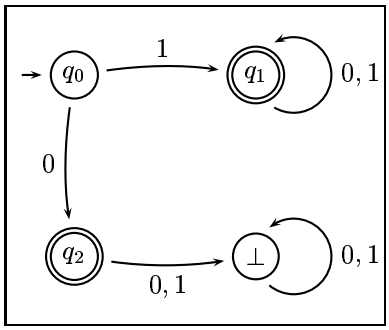


FIG. 2.2 – Automate déterministe complet reconnaissant les entiers naturels en numération binaire.

Notons que la reconnaissance d'un mot w par un automate est automatique et se trouve donc exempte d'ambiguïté : la lecture des lettres composant le mot provoque des transitions bien définies jusqu'à être bloqué (en cas de transitions manquantes si l'automate est incomplet, et le mot n'est alors pas reconnu, c'est le cas du mot 00 pour notre exemple) ou bien jusqu'à atteindre un état qui peut être (et le mot est reconnu) ou ne pas être (et le mot n'est pas reconnu) un état de satisfaction. En pratique, il peut être utile de rendre un automate complet en ajoutant un nouvel état appelé *poubelle*, étiqueté par $\perp$, vers lequel vont toutes les transitions manquantes. L'automate obtenu reconnaît exactement le même langage si l'on convient que le nouvel état poubelle n'est pas acceptant. Ainsi, la figure 2.2 présente un automate complet reconnaissant le langage des entiers naturels en représentation binaire.

Si l'automate A est complet, on pourra donc étendre l'application T aux mots, en posant $T(q,u) = q' \in Q$ tel que $q \xrightarrow{u} q'$.

2.2 Automates non déterministes

On en considère de deux formes, avec ou sans transitions vides.

Définition 2.4 *Un* automate non-déterministe $\mathcal{A}$ *est un triplet* (V_t, Q, T) *où*

1. V_t *est le* vocabulaire *de l'automate ;*
2. Q *est l'ensemble fini des* états *de l'automate ;*
3. $T : Q \times V_t \to \mathcal{P}(Q)$*, est la* fonction de transition *de l'automate.*

On notera comme précédemment $q \xrightarrow{\alpha} q'$ *pour* $q' \in T(q,\alpha)$ *avec* $\alpha \in V_t$*, et par* $T(q,u)$ *l'ensemble (peut-être vide) des états atteignables depuis* q *en lisant le mot* u*.*

Notons qu'un automate déterministe est un cas particulier d'automate non-déterministe qui associe à tout élément de $Q \times V_t$ une partie de Q possédant zéro ou un élément (exactement un si c'est

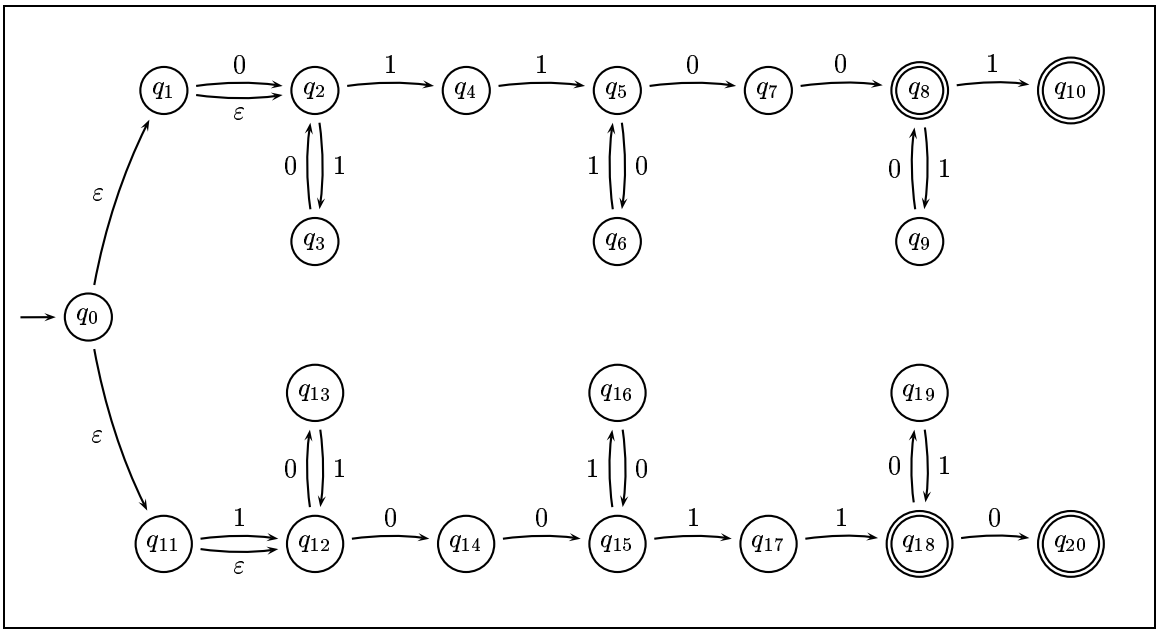


FIG. 2.3 – Un automate non-déterministe avec transitions vides.

un automate complet). On pourra dire d'un automate déterministe qu'il est incomplet s'il peut se bloquer, c'est-à-dire s'il existe un état q et une lettre a tels que $T(q, a) = \emptyset$.

Un automate avec transitions vides est un automate non-déterministe où certaines transitions peuvent être étiquetées par ε, qui dénotera l'absence de lettre lue lors de la transition.

Définition 2.5 *Un* automate non-déterministe A avec transitions vides *est un triplet* (V_t, Q, T) *où*

1. V_t *est le* vocabulaire *de l'automate ;*
2. Q *est l'ensemble fini des* états *de l'automate ;*
3. $T : Q \times (V_t \cup \varepsilon) \to \mathcal{P}(Q)$ *est la* fonction de transition *de l'automate.*

On notera $q \xrightarrow{\alpha} q'$ *pour* $q' \in T(q, \alpha)$, *avec* $\alpha \in V_t$ *ou* $\alpha = \varepsilon$.

Par exemple, la figure 2.3 représente un automate non-déterministe avec transitions vides reconnaissant le langage sur l'alphabet $\{0, 1\}$ contenant exactement une occurrence de chacun des mots 00 et 11.

Notons que la notation $q \xrightarrow{\alpha} q'$ devient ambigue, puisqu'elle prend maintenant deux significations différentes suivant que α est considérée comme une lettre (ou comme le symbole ε) et il y aura alors une transition unique, ou comme un mot (de longueur 1 ou 0), et il pourra alors y avoir un nombre arbitraire de transitions (dont au plus une sera non-vide). Lever l'ambiguité nécessite d'indiquer le nombre exact de transitions effectuées.

La notion de calcul est la même pour ces deux nouvelles variétés d'automates que pour les automates déterministes, ainsi que la notion de reconnaissance. Le non-déterministe a toutefois une conséquence essentielle : il peut y avoir plusieurs calculs issus de q_0 qui lisent un mot w donné, dont certains peuvent terminer en état acceptant et d'autres pas. Si tous les calculs échouent, il n'y aura pas d'autre moyen que de les explorer tous avant de savoir que le mot w n'est pas reconnu. La bonne notion n'est donc pas celle de calcul, mais d'arbre de calcul associé à un mot u lu par l'automate :

Définition 2.6 *Étant donné un automate non déterministe* $\mathcal{A}$, *l'arbre de calcul de racine* $q \in Q$ *engendré par la lecture du mot* $u \in V_t^*$ *est un arbre doublement étiquetté défini par récurrence sur* u *:*

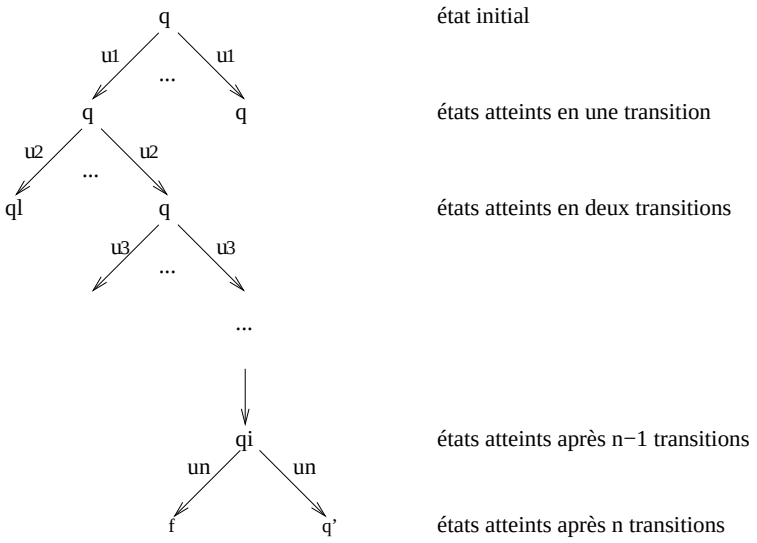


FIG. 2.4 – Arbre des calculs d'un automate non déterministe.

Si $u = \varepsilon$, alors l'arbre est réduit à sa racine q ;

Si $u = av$, la racine q possède des transitions sortantes étiquettées par $a \in V_t$ vers différents arbres de calcul engendrés par la lecture de v et dont les racines sont étiquettées par les états de $T(q, a)$.

Un arbre de calcul est représenté à la figure 2.4.

Les calculs d'un automate avec transitions vides autorisent le passage par un nombre quelconque de transitions vides au cours de l'exécution. Le nombre d'états parcourus ne sera donc plus égal au nombre de lettres du mot d'entrée mais pourra être beaucoup plus grand. L'arbre de calcul d'un automate avec transitions vides s'avère du coup être un outil moins intéresant que pour les automates non-déterministes sans transitions vides, nous ne le définirons pas.

Si les automates non-déterministes (avec des transitions vides éventuelles) sont un outil de modélisation fondamental en pratique,ils sont au contraire d'une manipulation informatique plus complexe causée par leur non-déterminisme. Mais le principe est que l'utilisateur est roi.

2.3 Décision du vide d'un automate

Qu'il soit ou non déterministe, un automate peut posséder des états superflus, en ce sens qu'ils peuvent être tout simplement retirés sans changer le langage reconnu.

Définition 2.7 *Étant donné un automate (déterministe ou pas) $A = (V_t, Q, q_0, F, T)$, on dira que l'état $q \in Q$ est*

- accessible, *s'il existe un mot w tel que $q_0 \xrightarrow{w}{}^* q$;*
- productif, *s'il existe un mot w tel que $q \xrightarrow{w}{}^* f \in F$;*
- utile, *s'il est à la fois accessible et productif.*

Un automate est dit réduit *si tous ses états sont utiles.*

Afin d'éliminer les états inutiles d'un automate, il faut en éliminer successivement les états inaccessibles et les états improductifs, opération appellée *nettoyage*. L'automate obtenu sera un automate réduit équivalent à l'automate de départ, il aura pour transitions celles qui relient des états utiles de l'automate de départ. Reste à calculer les états productifs, puis les états accessibles. Pour cela, on

observe qu'un état q est productif ssi $q \in F$ ou bien il existe un état productif q' et une lettre $a \in (V_t \cup \varepsilon)$ tels que $q' \in T(q, a)$. On en déduit un algorithme simple qui procède, en temps linéaire, jusqu'à stabilisation par ajouts successifs d'états productifs à un ensemble réduit initialement à F. De même, un état q est accessible ssi $q = q_0$ ou bien il existe un état accessible q' et une lettre $a \in (V_t \cup \varepsilon)$ tels que $q \in T(q', a)$. On en déduit un algorithme similaire qui procède, toujours en temps linéaire, jusqu'à stabilisation par ajouts successifs d'états accessibles à un ensemble réduit initialement à $\{q_0\}$.

Le nettoyage nous permet donc de décider en temps linéaire si le langage reconnu par un automate est vide, fini ou infini :

Lemme 2.8 *Le langage d'un automate $\mathcal{A}$ est non vide si et seulement si l'un au moins des états de F est accessible.*

Lemme 2.9 *Le langage d'un automate $\mathcal{A}$ sans transition vide est fini si et seulement son automate réduit est sans cycle, et infini s'il possède un cycle.*

Théorème 2.10 *Le vide, la finitude et l'infinitude du langage reconnu par un automate sont décidables en temps linéaire.*

Notons que le problème du plein du langage reconnu par un automate fini déterministe (savoir s'il reconnait exactement V^*) a la même complexité que le problème du vide, puisqu'ils sont interréductibles par échange des états acceptants et non-acceptants. Cela n'est pas vrai des automates non-déterministes (aucun algorithme polynomial n'est connu dans ce cas pour le résoudre le problème du plein).

2.4 Élimination des transitions vides d'un automate

Nous voulons maintenant montrer que le langage reconnu par un automate avec transitions vides peut également l'être par un automate non-déterministe sans transitions vides. Cette opération va nécéssiter l'addition de nouvelles transitions dans l'automate :

Définition 2.11 *Soit $A = (V_t, Q, q_0, F, T)$ un automate non-déterministe avec transitions vides. On définit $\mathcal{NDet}(A)$ comme l'automate $(V_t, Q, \{q_0\}, F_{\mathcal{NDet}(A)}, T_{\mathcal{NDet}(A)})$ où*
$q \xrightarrow{a}_{\mathcal{NDet}(A)} q'$ *ssi* $q \xrightarrow{\varepsilon}_A q'' \xrightarrow{a}_A q'$
$F_{\mathcal{NDet}(A)} = \{q \mid q \xrightarrow{\varepsilon}_A f \in F\}$

Dans la définition ci-dessus, la lettre a est considérée comme une lettre, alors que le symbole ε est considéré comme un mot de longueur nulle (aussi bien dans $q_0 \xrightarrow{\varepsilon}_A q''$ que dans $q \xrightarrow{\varepsilon}_A f \in F$). Notons que cette construction n'augmente pas la complexité de l'automate définie comme son nombre d'états.

Théorème 2.12 *Soit A un automate non-déterministe avec transitions vides. Alors $\mathcal{NDet}(A)^F_{q_0}$ est un automate non-déterministe qui reconnaît le même langage que $A^F_{q_0}$.*

La preuve est simple, et procède par découpage d'un calcul allant de q_0 à $q_n = f \in F$ dans l'automate A en tronçons de la forme $q_i \xrightarrow{\varepsilon \in V_t}{}^*_A \xrightarrow{a \in V_t}_A q_{i+1}$ suivis d'un dernier tronçon de la forme $q_{n-1} \xrightarrow{\varepsilon \in V_t}{}^*_A f \in F$.

Appliquée à l'automate de la figure 2.3, la construction précédente nous donne l'automate non-déterministe de la figure 2.5. Les états q_1 et q_{11} ont été supprimés car non accessibles.

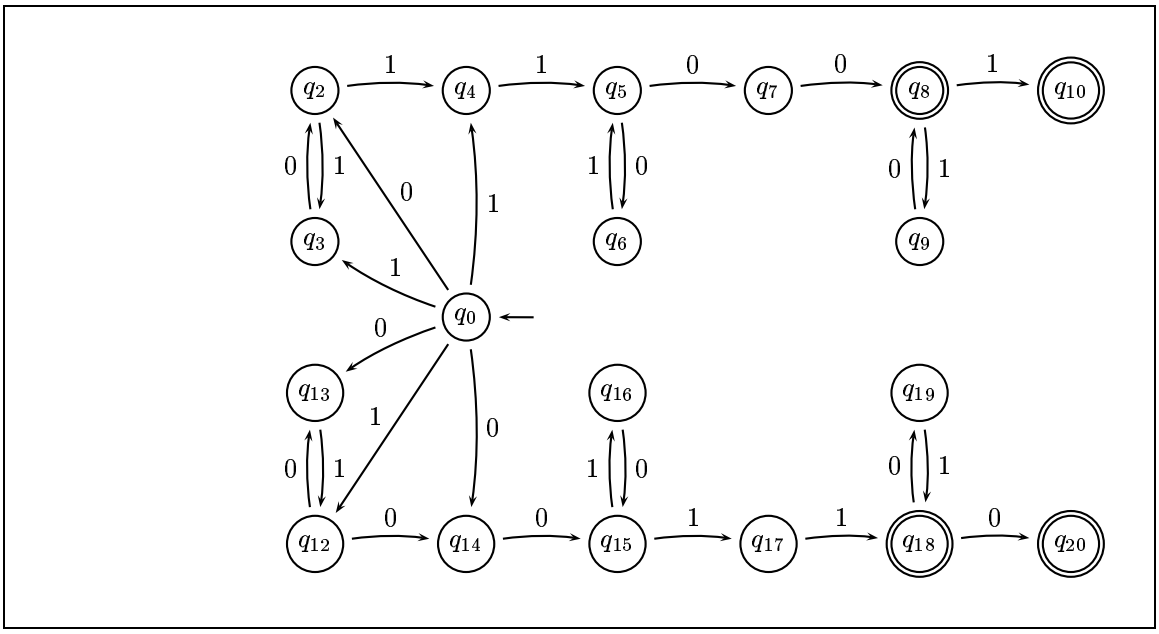


FIG. 2.5 – Automate non-déterministe sans transitions vides.

2.5 Déterminisation d'un automate non-déterministe

Nous allons maintenant déterminiser un automate non-déterministe (sans transitions vides) en ajoutant de nouveaux états : si Q est l'ensemble des états d'un automate non-déterministe, l'ensemble des états de l'automate déterministe associé sera $\mathcal{P}(Q)$, l'ensemble des parties de Q.

Définition 2.13 *Soit $A = (V_t, Q, T)$ un automate non-déterministe. On définit $\mathcal{D}et(A)$ comme l'automate $(V_t, \mathcal{P}(Q), \{q_0\}, F_{det}, T_{det})$ où $T_{det}(K, a) = \bigcup_{q \in K} T(q, a)$.*

Théorème 2.14 *Soit A un automate non-déterministe. Alors $\mathcal{D}et(A)^{\{K \in \mathcal{P}(Q) \mid K \cap F \neq \emptyset\}}_{\{q_0\}}$ est déterministe et reconnaît le même langage que $A^F_{q_0}$.*

Si la construction peut sembler complexe au néophyte, la preuve est par contre très simple, et basée sur l'idée que s'il est possible dans l'automate non-déterministe d'atteindre les états $q_1, \ldots, q_n$ depuis l'état q en lisant la lettre a, alors il sera possible dans l'automate déterminisé d'atteindre un état contenant la partie $\{q_1, \ldots, q_n\}$ depuis tout état contenant l'état $\{q\}$ en lisant cette même lettre a. En fait, l'idée de la preuve est parfaitement décrite par la notion d'arbre de preuve représentée à la figure 2.4.

L'automate déterminisé correspondant à l'automate de la figure 2.5 est représenté sur la figure 2.6, où seulement les états accessibles sont montrés. Les états sont étiquetés par l'ensemble des noms des états de l'automate non-déterministe qui le constituent.

2.6 Minimisation d'un automate déterministe

Dans tout ce paragraphe, il est fondamental de supposer les automates *déterministes complets*, même si leur représentation en graphe correspond parfois à un automate incomplet par souci de clarté des dessins. On supposera également que tous les états des automates manipulés sont accessibles.

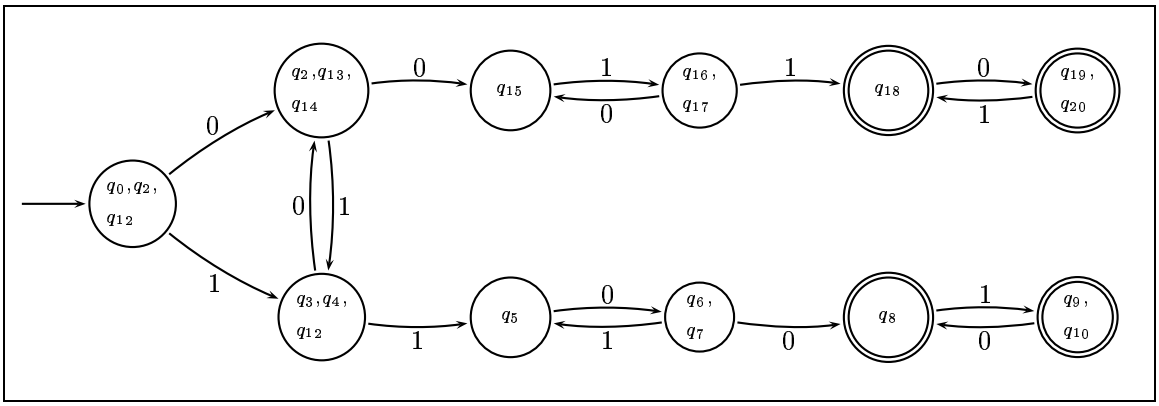


FIG. 2.6 – Automate déterminisé.

La déterminisation d'un automate non-déterministe fait exploser le nombre d'états, mais tous les états de $\mathcal{P}(Q)$ ne sont pas utiles à la construction du déterminisé d'un automate dont Q est l'ensemble des états. De plus, il se trouve que l'automate construit est en fait souvent plus compliqué que nécessaire. En particulier, il contient génralement des états inaccessibles. L'élimination des états inaccessibles n'est cependant pas toujours suffisante pour obtenir l'automate le plus simple possible : si deux états d'un automate reconnaissent le même langage, on peut *réduire* l'automate en les confondant. Le problème, bien sûr, va être de déterminer si deux états q et q' reconnaissent le même langage, de manière à partitionner l'ensemble des états afin de confondre les états d'une même classe.

2.6.1 Automate minimal

La première question, bien sûr est celle de l'existence et de l'unicité d'un automate minimal pour tout langage reconnaissable. C'est l'objet du développement qui suit.

Définition 2.15 *On dit qu'une relation d'équivalence $\simeq$ sur les mots d'un vocabulaire V est*

- d'index fini *si elle partitionne l'ensemble des mots en un nombre fini de classes,*
- une congruence droite *si* $\forall u, v \in V^* \; \forall a \in V \; u \simeq v \Longrightarrow ua \simeq va$.

Lemme 2.16 *Soit $\sim$ une congruence droite d'index fini sur V_t^* dont on note Q l'ensemble des classes d'équivalence, et $F \subseteq Q$. Alors $\mathcal{A}_\sim = (V_t, Q, [\varepsilon], F, T)$ avec $T([u], a) = [ua]$ est un automate fini déterministe complet qui reconnait le langage des mots appartenant à une classe de F.*

Preuve: Comme $\sim$ est d'index fini, Q est fini. Comme $\sim$ est une congruence droite, la définition de T ne dépend pas du choix de u dans $[u]$, donc T est bien une fonction totale. Donc $\mathcal{A}$ est un automate fini déterministe complet.

Reste à montrer qu'il reconnait le langages des mots appartenant à une classe de F. Pour cela, nous montrons par récurrence sur le mot v la propriété :

$$\forall v \in V_t^* \; \forall [u] \in Q \; [u] \xrightarrow{v}_A [uv]$$

Si $v = \varepsilon$, la propriété est trivialement vraie.
Si $v = aw$, alors $[u] \xrightarrow{a}_A [ua]$ par définiton de T, et $[ua] \xrightarrow{w}_A [uaw]$ par hypothèse de récurrence, d'où la propriété.

Appliquant maintenant cette propriété pour $u = \varepsilon$, on obtient $\forall v \in V_t^* \; [\varepsilon] \xrightarrow{v}_A [v]$, et donc v est reconnu ssi il appartient à une classe de F. □

Définition 2.17 *Étant donné un langage L sur V_t et un mot $u \in V_t^*$, on appelle :*
contexte à droite *de u dans v l'ensemble $D_L = \{v \mid uv \in L\}$;*
congruence syntaxique *de L l'équivalence $u \sim_L v$ ssi $D_L(u) = D_L(v)$.*

Lemme 2.18 *La congruence syntaxique d'un langage quelconque est une congruence droite.*

Preuve: Soit $u \sim_L v$. Pour tout $w, w' \in V^*$, $uww' \in L$ ssi $uww' \in L$ et donc $uw \sim_L vw$, prouvant que $\sim_L$ est une congruence droite. □

Définition 2.19 *Soit L reconnaissable par un automate déterministe complet $\mathcal{A}$, et $\sim_\mathcal{A}$* l'équivalence de transition *engendrée sur les mots de V définie par $u \sim_\mathcal{A} v$ ssi $T(q_0, u) = T(q_0, v)$.*

Lemme 2.20 *L'équivalence de transition $\sim_\mathcal{B}$ d'un automate $\mathcal{B}$ est une congruence droite d'index fini telle que $\mathcal{A}_{\sim_\mathcal{B}} = \mathcal{B}$.*

Preuve: Soit $u \sim_A v$. Alors $T(q_0, u) = T(q_0, v)$ et donc pour tout $w \in V^*$, $T(q_0, uw) = T(T(q_0, u), w)) = T(T(q_0, v), w)) = T(q_0, vw)$, et par définition, $uw \sim_A vw$. Comme les classes sont en correspondance biunivoque avec les états, la congruence est d'index fini et $\mathcal{A}_{\sim_\mathcal{A}} = \mathcal{A}$. □

Théorème 2.21 (Myhill-Nerode) *Tout langage reconnaissable L sur un alphabet V est reconnu par l'automate $A_{\sim_L}$ qui se trouve être l'automate minimal (en nombre d'état) reconnaissant L.*

Preuve: On montre que la congruence syntaxique est d'index fini en prouvant que $\sim_\mathcal{A}$ est un raffinement de $\sim_L$, c'est-à-dire que $u \sim_\mathcal{A} v \Longrightarrow u \simeq_L v$. Cela aura pour corrollaire que l'automate associé à la congruence syntaxique par les Lemmes 2.18 et 2.16 reconnait L et est minimal.

Soit $u \sim_\mathcal{A} v$, et comme $\sim_\mathcal{A}$ est une congruence droite, pour tout $w \in V^*$, $uw \sim_\mathcal{A} vw$, et donc $uw \in L$ ssi $vw \in L$ puisque L est une union de classes d'équivalence de $\sim_\mathcal{A}$, et par définition de $\sim_L$, on en déduit que $u \sim_L v$. $\sim_\mathcal{A}$ a donc au moins autant de classes que $\simeq_L$, ce qui prouve que cette dernière est d'index fini et même minimal. □

Il nous reste maintenant à calculer cet automate minimal unique (à renommage près des états).

2.6.2 Calcul de l'automate minimal

L'idée est de calculer l'automate minimal d'un langage reconnaissable à partir d'un automate arbitraire le reconnaissant. Pour cela, plutôt que de raisonner en terme de congruence sur les mots du langage, nous allons considérer les classes de congruence elles-même, en ce qu'elles forment une partition des états de l'automate qui a la propriété clé de *compatibilité* :

Définition 2.22 *Étant donné un automate déterministe complet $\mathcal{A} = (V_t, Q, q_0, F, T)$, une partition $\mathcal{P}$ de Q est* compatible *ssi*

1. *elle est un rafinement de la partition $\{F, Q \setminus F\}$:*

$$\forall C \in \mathcal{P}\ C \subseteq F \text{ or } C \subseteq (Q \setminus F)$$

2. *toute classe de la partition est* compatible *:*

$$\forall C \in \mathcal{P}\ \forall q, q' \in C\ \forall a \in V_t\ \forall C'\ T(q, a) \in C' \text{ ssi } T(q', a) \in C'$$

Les partitions compatibles ont bien sûr un lien étroit avec les congruences droites.

Définition 2.23 *Étant donnée une partition $\mathcal{P}$ de Q,* l'équivalence de partition *sur les mots associée à $\mathcal{P}$ est définie par $u \sim_\mathcal{P} v$ ssi u et v appartiennent à une même classe de $\mathcal{P}$.*

Lemme 2.24 *Soit $\mathcal{P}$ une partition compatible des états d'un automate $\mathcal{A} = (V_t, Q, q_0, F, T)$. Alors $\sim_{\mathcal{P}}$ est une congruence droite d'index fini et l'automate*

$$\mathcal{A}_{\mathcal{P}} = (V_t, \mathcal{P}, C_0, \{C \in \mathcal{P} \mid C \subseteq F\}, T_q)$$

C_0 est la classe qui contient l'état q_0

$T_q(C, a) = C'$ *ssi* $\exists q \in C\ \exists q' \in C'$ *tel que* $T(q, a) = q'$

est un automate fini déterministe qui reconnait le langage $\mathcal{L}(\mathcal{A})$.

Preuve: La propriété de congruence droite découle directemnt de la compatibilité de la partition. La congruence est d'index fini puisque son nombre de classes est majoré par la taille de Q. Enfin, l'automate $\mathcal{A}_{\mathcal{P}}$ est tout simplement l'automate $\mathcal{A}_{\sim_{\mathcal{P}}}$ dont les états acceptants sont les classes associées à la partition de F : il reconnait donc le langage $L(\mathcal{A})$ d'après le Lemme 2.16. □

Nous sommes donc ramenés à trouver la partition compatible de Q dont le nombre de classes est minimum. Le problème est que la détermination a-priori d'une telle partition n'est pas simple, et peut difficillement être obtenue par un algorithme qui confond les états déterminés comme équivalents. Nous allons au contraire procéder *à l'envers* : l'idée est de partir d'une partition des états en deux classes, les états acceptants d'une part, et les autres, puisqu'une classe compatible ne peut contenir à la fois des acceptants et des non-acceptants. On va ensuite affiner cette partition chaque fois que nécessaire, c'est-à-dire lorsque une classe C ne sera pas compatible. Ce sera le cas lorsqu'une même lettre a provoque la transition de $p \in C$ vers p' et de $q \in C$ vers q', p' et q' n'appartenant pas à une même classe, donc ne reconnaissant pas le même langage. On doit alors partitionner C en sous classes compatibles, ce qui fera apparaître au moins deux nouvelles classes, C' contenant p et C'' contenant q. Ce partitionnement va rendre incompatibles des classes auparavent compatibles, et il va donc falloir tester toutes les classes à l'exception de celles issues du partitionnement de C. L'algorithme s'arrête (nécessairement) lorsque toutes les classes sont compatibles.

Étant donnée une partition P des états d'un automate, la fonction suivante recherche une classe de P qui n'est pas compatible. Dans le cas où toutes les classes sont compatibles la fonction retourne $\emptyset$. Puisque tous les classes dans les partitions construites par l'algorithme sont non-vides, le résultat $\emptyset$ signale le cas d'échec de la recherche.

```
type classe sur Q = ensemble sur Q
type partition sur Q = ensemble sur classe sur Q
fonction non_compatible(A : automate sur (V_t, Q),
                        P : partition sur Q) : classe sur Q
% Recherche d'une classe non compatible de P
 si il existe C ∈ P, p ∈ C, q ∈ C et a ∈ V_t tels que
   transition(A, p, a) ∈ C' et transition(A, q, a) ∈ C'' avec C' ≠ C''
   retourner C
 sinon
   retourner ∅
 fin si
fin fonction
```

La fonction suivante calcule alors, à partir d'une partition P donnée, le *raffinement compatible le plus grossier* de P, c'est-à-dire la sous-partition compatible de P ayant le plus petit nombre possible de classes (on peut montrer qu'elle est unique) :

```
fonction partition(A : automate sur (V_t,Q),
                   P : partition sur Q) : partition sur Q
% Calcul d'un raffinement compatible de la partition P
 soit C = non_compatible(A, P)
 si C = ∅
   retourner P
 sinon
   soit P' la partition la plus grossière de C telle que
```

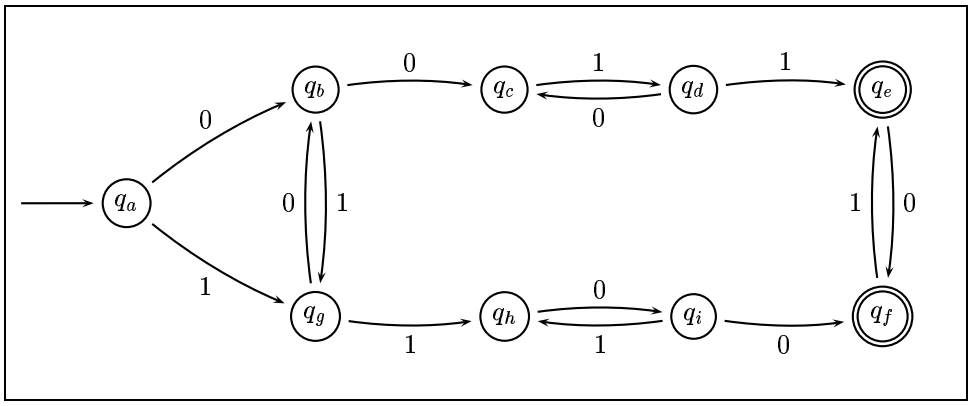


FIG. 2.7 – Automate déterminisé et minimisé.

pour tout $C' \in P'$, pour tout $C'' \in P$, pour tous $p, q \in C'$
pour tout $a \in V_t$:
transition$(A, p, a) \in C''$ ssi *transition*$(A, q, a) \in C''$
retourner *partition*$(A, (P - \{C\}) \cup P')$
fin si
fin fonction

La fonction suivante calcule l'automate minimisé.

fonction *minimisation*(A : automate sur (V_t, Q)) : automate sur $(V_t$, classe sur $Q)$
% *Calcul du minimisé de A*
% *Calcul de la plus grossière partition compatible*
soit $F = \{q \in Q \mid$ *est_acceptant*$(A, q)\}$
$P =$ *partition*$(A, \{Q - F, F\})$
% *Calcul de l'état initial de l'automate minimal*
soit I_{min} la classe de P telle que *etat_initial*$(A) \in I_{min}$
% *Calcul des états acceptants de l'automate minimal*
pour chaque $C \in P$:
soit $F_{min}(C) =$ true **ssi** $C \cap F \neq \emptyset$
fin pour
% *Calcul de la fonction de transition de l'automate minimal*
pour chaque $(C, a) \in P \times V_t$ **faire**
soit $q \in C$ (un représentant quelconque)
soit C' la classe de P telle que *transition*$(A, q, a) \in C'$
$T_{min}(C, a) = C'$
fin pour
retourner $(I_{min}, F_{min}, T_{min})$
fin fonction

Le minimisé de l'automate de la figure 2.6 est représenté sur la figure 2.7.

Théorème 2.25 *Soit $A = (V_T, Q, q_0, F, T)$ un automate déterministe et complet. L'automate résultat de la fonction minimisation est l'automate déterministe complet avec un nombre minimal d'états acceptant le même langage que A.*

2.7 Propriétés de pompage et de clôture des langages reconnaissables

Le pompage est une opération sur les mots qui laisse invariant un langage reconnaissable :

Théorème 2.26 *Pour tout langage reconnaissable L, il existe une constante (dite de pompage) N telle que tout mot m de longueur au moins N peut s'écrire sous la forme d'un produit de concaténation $u \cdot v \cdot w$ tel que :*
(i) $v \neq \varepsilon$
(ii) $|u \cdot v| < N$
(iii) $\forall n \in \mathbb{N},\ u \cdot v^n \cdot w \in L$

Preuve: La preuve montre que l'on peut prendre pour N le nombre d'états d'un automate reconnaissant L. En effet, la reconnaissance d'un mot fait intervenir un d'état de plus que sa longueur, et donc un état q au moins est répété dans la reconnaissance. le mot reconnu jusqu'à q est u, le mot reconnu sur le chemin de q à la première répétition de q est v est le reste est w. Notons la satisfaction e la condition (i). Parmi les états q possibles, on choisit le premier à se répeter, ce qui assure (ii). La propriété (iii) est évidente, elle consiste à parcourir la boucle sur q un nombre arbitraire (positif ou nul) de fois. □ □

Ce théorème sert à montrer que certains langages ne sont pas reconnaissables :

Soit $L = \{a^N b^n\}$. Supposons L reconnaissable, et soit N sa constante de pompage. Pompons le mot $m = a^N b^N$. Comme uv est dans a^N, v est formé d'un nombre non nul de a, et il suffit donc de prendre $n = 0$ pour obtenir une contradiction.

Certains langages non reconnaissables vérifient le théorème de pompage ci-dessus, et la méthode ci-dessus ne s'applique donc pas dans ces cas-là. On peut résoudre ce problème de deux manières, en renforçant le théorème de pompage, ou en transformant le langage de départ par une transformation qui conserve la reconnaissabilité (mais pas la propriété de pompage). Comme ces transformations jouent un rôle fondamental par ailleurs, nous allons les voir en détail.

Soit f une opération d'arité n sur les mots. Étant donnés les alphabets $V_1, \ldots, V_n$ et les langages L_1 sur $V_1, \ldots, L_n$ sur V_n, on définit le langage $f(L_1, \ldots, L_n) = \bigcup_{u_i \in L_i} f(u_1, \ldots, u_n)$ sur l'alphabet $V_1 \cup \ldots \cup V_n$

On dit que les langages reconnaissables sont clos par une opération f si $f(L_1, \ldots, L_n)$ est reconnaissable lorsque les langages $L_1, \ldots, L_n$ le sont.

Nous nous intéressons avant tout aux opérations Booléennes sur les langages. Les résultats sont énoncés sans peuve, qui peuvent être faites en exercice. Les langages considérés seront bien sûr définis par des automates les reconnaissant, de sorte que les opérations peuvent être vues comme agissant directement sur les automates.

On commence par la clôture par complémentaire :

Théorème 2.27 *Soient $A = (V_t, Q, q_0, F, T)$ un automate déterministe complet. Alors $A = (V_t, Q, q_0, Q \setminus F, T)$ reconnait le langage complémentaire de $\mathcal{L}ang(A)$.*

Il est fondamental dans cet énoncé que l'automate de départ soit complet. Passons maintenant à l'union et à l'intersection :

Définition 2.28 *Étant donnés deux automates déterministes $A_1 = (V_t, Q_1, T_1)$ et $A_2 = (V_t, Q_2, T_2)$ sur le même alphabet V_t, l'automate $A = (V_t, Q_1 \times Q_2, T)$ où $T((q_1, q_2), a) = \{(T(q_1, a), T(q_2, a)\}$ est appellé automate produit cartésien de A_1 par A_2.*

Notons que l'automate produit cartésien a pour effet de synchroniser les transitions, donc les calculs des automates de départ.

Théorème 2.29 *Soient $A = (V_t, Q, T)$ et $A' = (V_t, Q', T')$ deux automates déterministes sur le même alphabet V_t, et B leur automate produit cartésien. Alors, l'automate $B^{F\times F'}_{(q_0,q'_0)}$ reconnait le langage $\mathcal{L}ang(A^F_{q_0}) \cap \mathcal{L}ang(A'^{F'}_{q'_0})$, et si les automates de départ sont complets, l'automate $B^{(F\times Q')\cup(F'\times Q)}_{(q_0,q'_0)}$ reconnait le langage $\mathcal{L}ang(A^F_{q_0}) \cup \mathcal{L}ang(A'^{F'}_{q'_0})$.*

La construction ci-dessus se généralise sans difficulté pour reconnaitre l'intersection des langages reconnus par deux automates non-déterministes sans transitions vides A et A' ne possèdant pas le même alphabet : il suffit pour cela de définir T sous la forme plus générale (mais équivalente pour les automates déterministes complets) $T((q_1, q_2), a) = \{(q'_1, q'_2) \mid q'_1 \in T(q_1, a)$ et $q'_2 \in T(q_2, a)\}$ qui fournit un automate incomplet sur l'alphabet union (mais complet sur l'alphabet intersection) qui répond à la question.

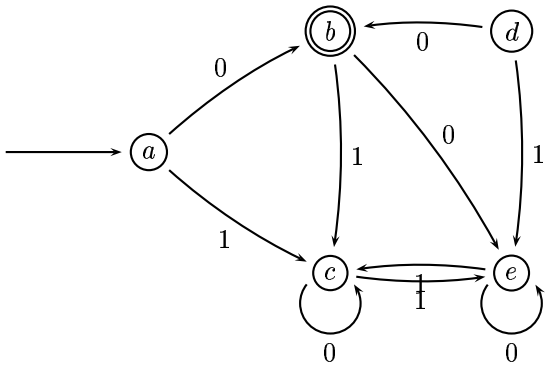
Par contre, il faut faire attention en ce qui concerne l'union des langages reconnus, car il faut disposer d'automates qui soient complets sur l'alphabet union. Il suffit pour cela de rajouter une poubelle de manière à ce qu'un automate ne se bloque pas pendant que l'autre est en train de reconnaitre. Cette construction fonctionne également pour des automates non déterministes.

Nous verrons dans le chapitre 3 une construction qui généralise les deux constructions précédentes : le produit synchronisé d'automates.

Les langages reconnaissables possèdent de très nombreuses propriétés de clôture, en particulier par substitution (des lettres dans les mots d'un langage par des mots pris arbitrairement dans un langage régulier associé à chaque lettre, le cas particulier où chaque lettre est remplacée par un mot étant appellé homomorphisme), par homomorphisme inverse, par quotient à droite, par shuffle, etc.

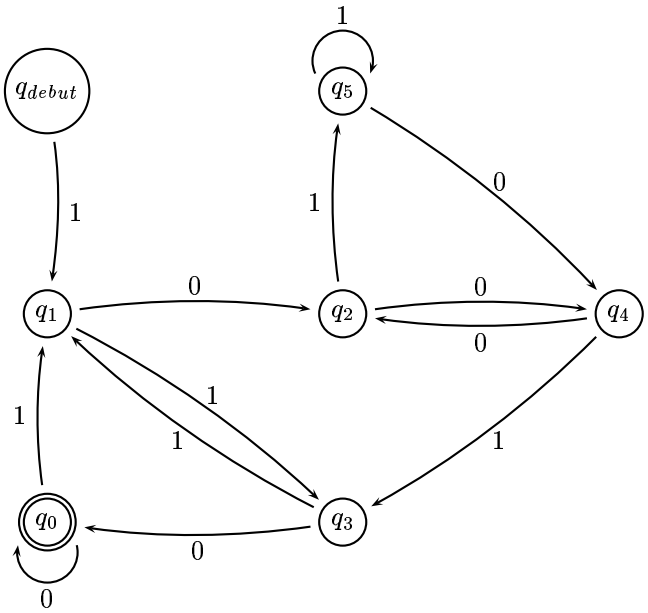
2.8 Exercices

Exercice 2.1 *On considère l'automate suivant reconnaissant le langage L_0 :*



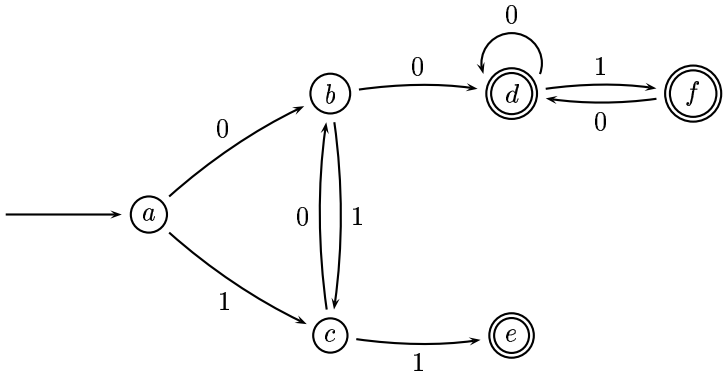
On demande de nettoyer cet automate, puis de donner le langage qu'il reconnait.

Exercice 2.2 *On considère l'automate suivant reconnaissant le langage L_0 :*



1. *Verifiez que les mots* 110, 1100, 10010 *sont reconnus par cet automate.*
2. *Caractériser le language L_0 reconnu par cet automate ?*
3. *Si w est un mot, on note $mir(w)$ le mot miroir de w : si $w = a_1.a_2.\ldots.a_n$ alors $mir(w) = a_n.\ldots.a_2.a_1$.*
   Et on definit le langage miroir comme : $mir(L) = \{mir(w), w \in L\}$.
   Donner un automate pour le langage $mir(L_0)$.

Exercice 2.3 *Construire l'automate minimal associé à l'automate suivant :*



Exercice 2.4 *Soit L_n l'ensemble des mots sur $\{a, b\}$ de longueur au moins n dont la $n$ième lettre avant la fin est un b.*

Donnez un petit automate non-déterministe, puis donnez l'automate déterministe minimal reconnaissant L_4.

Combien y-a-t-il d'états dans le plus petit automate non-déterministe (on ne cherchera pas à montrer que c'est le plus petit) et dans l'automate déterministe minimal reconnaissant L_n ? Comparez.

Exercice 2.5 *Montrez que les langages reconnaissables sur l'alphabet $\{1\}$ sont les unions finies de langages de la forme*

$$1^n \cdot (1^m)^*$$

On pourra donner la forme générale d'un automate déterministe sur un alphabet à une lettre.

En déduire que $\{1^{n^2} \mid n \geq 0\}$ n'est pas reconnaissable.

Exercice 2.6

1. *Soit ϕ le morphisme défini par $\phi(a) = a.b$, $\phi(b) = a$, $\phi(c) = b.a$ et $\phi(d) = bbb$. Décrire $\phi(abcd)$, $\phi((ab)^*)$, $\phi^{-1}(aba)$, $\phi^{-1}((bba)^*)$, $\phi^{-1}((bb)^*)$*
2. *Soit L et M deux langages et soit ϕ un homomorphisme. Montrer ou infirmer (en exhibant un contre-exemple) les propriétés suivantes :*

$$\begin{array}{l}
1)\phi(L \cup M) = \phi(L) \cup \phi(M) \\
2)\phi(L.M) = \phi(L).\phi(M) \\
3)\phi(L \cap M) = \phi(L) \cap \phi(M) \\
4)\phi^{-1}(L \cup M) = \phi^{-1}(L) \cup \phi^{-1}(M) \\
5)\phi^{-1}(L.M) = \phi^{-1}(L).\phi^{-1}(M) \\
6)\phi^{-1}(L \cap M) = \phi^{-1}(L) \cap \phi^{-1}(M) \\
7)\phi^{-1}(\phi(L)) = L
\end{array}$$

3. *En utilisant le fait que $\{a^n b^n\}$ n'est pas rationnel, montrer que les langages suivants ne sont pas rationnels :*
   - $L_1 = \{a^i b^j c^k \mid i + j = k \geq 0\}$
   - $L_2 = \{a^n b^n c^n \mid n \geq 0\}$
   - $L_3 = \{w \in (a + b)^* \mid |w|_a = |w|_b\}$
   - $L_4 = \{(ab)^{2n}(cd)^{2n} \mid n \geq 0\}$
   - $L_5 = \{a^n b a^n \mid n \geq 0\}$
   - $L_6 = \{w \in (a + b)^* \mid w$ *est un palindrome* $\}$
   - $L_7 = \{a^i b^j a^{i+j} | i, j \geq 0\}$

Exercice 2.7 *On se propose de définir un autre algorithme de minimisation des automates finis déterministes. Soit L un langage reconnu par un automate fini non-déterministe A.*

1. *Soit A' un automate fini déterministe reconnaissant le langage $mir(L)$.*

Soient maintenant B l'automate obtenu à partir de A' en éliminant les états inutiles, B' l'automate obtenu à partir de B par la construction précédente, et C l'automate obtenu à partir de B' en éliminant les états inutiles.

2. *Appliquer ces constructions à l'automate L_0 de l'exercice 2.2.*
3. *Appliquer ces constructions à l'automate de l'exercice 2.3.*
4. *Montrer que l'automate C est l'automate minimal reconnaissant le langage $\mathcal{L}(A)$.*

Exercice 2.8 *Soit L un langage reconnaissable. Montrer que les langages suivants sont reconnaissables :*

1. $demi(L) = \{u \in V^* \mid uu \in L\}$.
2. $demi(L) = \{u \in V^* \mid \exists v \in V^* \text{ such that } \mid u \mid = \mid v \mid \text{ and } uv \in L\}$.
3. $tiers(L) = \{u \in V^* \mid \exists v, w \in V^* \text{ such that } \mid v \mid = \mid w \mid = \mid u \mid \text{ and } uvw \in L\}$.
4. $demi(L) = \{u \in V^* \mid \exists v \in V^* \text{ such that } \mid v \mid = (\mid v \mid)^2 \text{ and } uv \in L\}$.

Exercice 2.9 *Montrer que le langage $L = \{a^n b^{2n} \mid n \geq 0\}$ n'est pas reconnaissable.*

Exercice 2.10 *On appelle* chien de garde *un systeme qui recoit les signaux $\{e, f, r, s\}$ et qui est spécifié ainsi :*

- *Initialement le système est en état de repos et ignore les messages recus.*
- *À l'arrivée du signal e (éveil), le système passe à l'état de veille.*
- *Quand le système est en veille, tout signal s le fait passer dans un état d'alerte.*
- *Le signal f (fin) le fait toujours passer dans l'état de repos.*
- *Le système reste en état d'alerte tant que le signal r (repos) n'est pas émis.*

Donner un automate réalisant un tel chien de garde.

Exercice 2.11 *Trois missionnaires et trois cannibales veulent traverser une rivière pleine de piranhas. Il y a une seule pirogue qui peut transporter deux personnes au maximum. Un missionnaire ne peut pas utiliser la pirogue tout seul (mais deux le peuvent). Dès qu'il y a, à quelque endroit, plus de cannibales que de missionnaires, les cannibales vont manger les missionnaires.*

- *Est-ce qu'il y a une possibilité pour les six personnes de traverser la rivière sans être mangées ?*
  Indication : utiliser un automate où les états sont les différentes répartitions possibles des missionnaires et des cannibales d'un côté et de l'autre de la rivière, et les transitions représentent les traversées de la pirogue.
- *Si l'on considère le mot des actions, que penser d'une solution miroir ? Justifier.*

Exercice 2.12 *Un barman, un client et un client arbitre jouent au jeu suivant :*

Le barman met un bandeau sur les yeux qui le rend aveugle, et il met des gants de boxe qui l'empêchent de "sentir" si un verre est à l'endroit ou à l'envers.

Devant le barman, se trouve un plateau tournant sur lequel sont placés quatre verres en carré. Ces verres peuvent être à l'envers ou à l'endroit. La configuration initiale est inconnue du barman.

Les joueurs alternent au plus 10 coups chacun comme suit : le client fait tourner le plateau d'un certain nombre de quarts de tour, puis le barman retourne au choix un ou deux verres parmi les 4. Si les verres sont alors tous dans le même sens, le barman gagne.

1. *On se place du point de vue du client. Donnez un automate dont les états sont les différentes configurations du plateau, les lettres les coups joués par le barman et les transitions les évolutions possibles des configurations.*
2. *Donnez un automate non déterministe (avec éventuellement plusieurs états d'entrée) qui donne toutes les séquences de coups du barman pour lesquelles le client peut gagner.*
3. *Donnez un automate qui donne les coups qui assurent au barman de gagner quel que soit le comportement du client.*

4. *Jouez-vous de l'argent contre le barman ?*
5. *Et si au lieu de 4 verres, il y a 3 verres en triangle ou bien 5 verres en pentagone ?*

Exercice 2.13 *Émile, Firmin et Gérard sont trois ouvriers. Ils se trouvent avec leur caisse à outils au sommet d'un immeuble en construction. Il n'y a pas d'escalier pour redescendre et ils ne disposent que d'un monte-charge rudimentaire formé de deux paniers reliés par un câble passant sur une poulie. Un panier ne peut contenir qu'au plus deux personnes ou une personne et la caisse à outils. La descente se fait naturellement, le panier le plus lourd l'emportant sans qu'il soit possible ni à ceux qui sont dans le panier ni aux autres d'aider cette descente. Enfin, si le panier qui descend a sur l'autre un excédent de poids de plus de 10 kg, toute personne qui s'y trouverait se blesserait à l'arrivée. En revanche, la caisse à outils supporte sans dommage l'atterrissage. Si Émile pèse 85 kg, Firmin 50 kg, Gérard 40 kg et la caisse à outils 30 kg, au départ un panier étant au sommet et l'autre au sol, combien de manœuvres au minimum des paniers devront effectuer les trois ouvriers pour rejoindre sains et saufs le sol avec la caisse à outil et comment ?*

Chapitre 3

Modélisation de systèmes réactifs par des automates

Ce chapitre présente plusieurs modélisation de complexité croissante qui montrent la puissance d'expression des automates et introduisent des constructions classiques visant à une certaine économie de description.

3.1 Automates à propriétés

Nous nous proposons dans ce paragraphe de modéliser un digicode (et lui seul, sans la porte pour l'instant) qui déclenche l'ouverture d'une porte lorsque le code 36A35 a été tapé sur son clavier, comme représenté à la figure 3.1. Dans ce modèle, l'arrivée à l'état 6 est supposée provoquer l'ouverture de la porte, le système retournant ensuite dans son état initial, action modélisée par une transition vide que l'on a figurée explicitement pour plus de clarté.

Faisons deux remarques :

- Dans le but d'alléger la spécification de l'automate (et son dessin), on s'autorise des transitions étiquettées par des expressions, qui sont soit des ensembles finis de lettres de l'alphabet, soit des complémentaires d'ensembles finis de lettres.

- l'automate est bien sûr donné incomplet, toujours dans le but d'en alléger la description.

Une question fondamentale concerne l'ensemble des états atteints : comprend-il un état interdit ? Dans l'exemple du digicode, l'ouverture de la porte (que nous n'avons pas encore modélisée), peut-elle être déclenchée avec un digicode erroné ? Pour répondre à ce type de question, il est important de considérer les exécutions possibles du digicode, qui sont représentées à la figure 3.2.

Notons que cet arbre est infini, mais nous ne considérons bien sûr (pour l'instant) que les exécutions finies représentées dans cet arbre infini.

On va maintenant associer à tout état une propriété logique qui a pour rôle de matérialiser la connaissance que l'on a du système à ce point d'un calcul en cours. Pour cela, on va se donner un ensemble de propriétés élémentaires à partir desquelles on devra exprimer les propriétés du système. On va par exemple considérer les propriétés suivantes :

- P_i : on vient d'appuyer sur la touche $i \in [0..9, A, B]$;
- Q_i : l'état précédent dans l'exécution est l'état i.

À chaque état on associe la formule propositionnelle, batie sur les propriétés élémentaires considérées comme des symboles propositionnels, décrite à la figure 3.3.

Considérons une exécution où la porte s'ouvre, c'est-à-dire, dans notre modèle, une exécution qui atteint l'état 6.

La dernière lettre tapée était un 5 (P_5), et on venait de l'état 5 (Q_5) ;

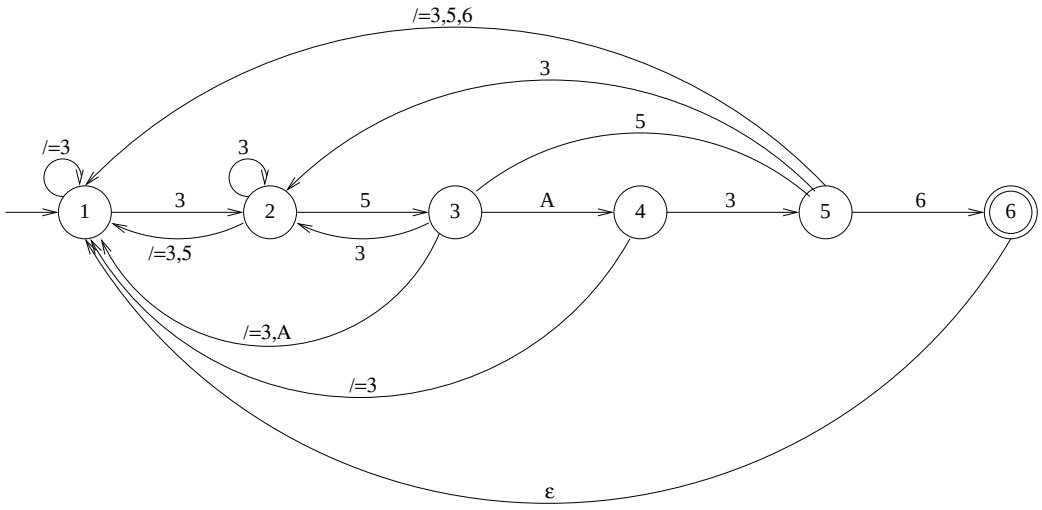


FIG. 3.1 – Digicode à utilisations multiples.

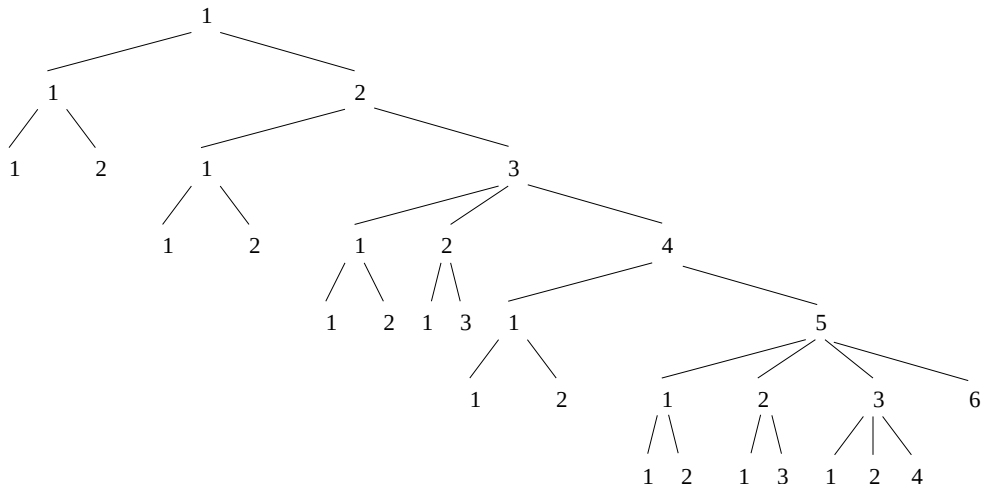


FIG. 3.2 – Arbre des exécutions du digicode à utilisation multiples.

État	Propriété
1	$False$
2	$P_3 \wedge (Q_1 \vee Q_3 \vee Q_5)$
3	$P_6 \wedge (Q_2 \vee Q_5)$
4	$P_A \wedge Q_3$
5	$P_3 \wedge Q_4$
6	$P_5 \wedge Q_5$

FIG. 3.3 – Propriétés des états du digicode.

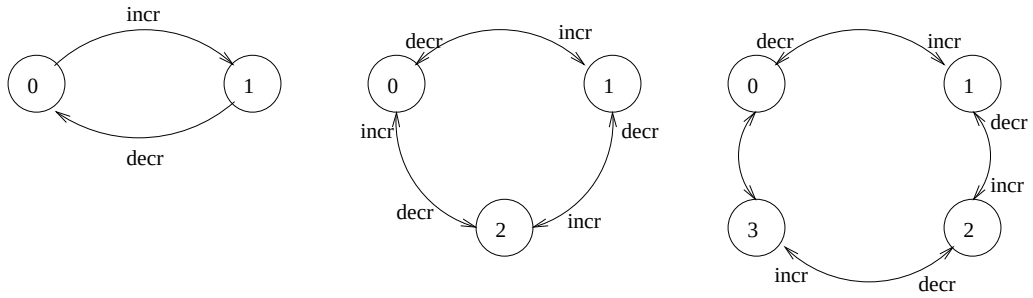


FIG. 3.4 – Compteurs modulo deux, trois et quatre.

En 5, la dernière lettre tapée était un 3 (P_3), et on venait de l'état 4 ;
En 4, la dernière lettre tapée était un A (P_A), et on venait de l'état 3 ;
En 3, la dernière lettre tapée était un 6 (P_6), et on venait de l'état 2 ou de l'état 5 ;
En 2 comme en 5, la dernière lettre tapée était un 3 (P_3) ;
Dans tous les cas, si la porte s'ouvre, c'est que le mot tapé sur le clavier se termine par le code d'ouverture. C'est ce que l'on appelle une propriété de sureté.

Cet exemple suggère une nouvelle définition des automates dans laquelle il n'y a pas d'états acceptants (l'arrivée en l'état 6 produit une action sur la porte, ce que l'on modélisera ultérieurement) car la suite de caractères tapée sur le clavier est implicitement sans fin. Par contre, les états seront associés à des propriétés.

Définition 3.1 *Un* automate déterministe $\mathcal{A}$ *est un sextuplet* $(V_t, Q, q_0, T, \mathcal{P}rop, l)$ *où*

1. V_t *est le* vocabulaire *de l'automate ;*
2. Q *est l'ensemble des* états *de l'automate ;*
3. q_0 *est l'état initial ;*
4. $T : Q \times V_t \to Q$, *est la* fonction de transition *de l'automate.*
5. $\mathcal{P}rop$ *est l'ensemble des* propriétés *de l'automate ;*
6. $l : Q \to \mathcal{P}(\mathcal{P}rop)$ *est un* étiquettage *des états de* Q *par des sous-ensembles de propriétés de* $\mathcal{P}rop$.

Les sous-ensembles de propriétés de $\mathcal{P}rop$ doivent etre interprétées comme des conjonctions. On aurait pu prendre la formulation plus générale $l : Q \to \mathcal{P}(\mathcal{P}(\mathcal{P}rop))$, où $\mathcal{P}(\mathcal{P}(\mathcal{P}rop))$ est interprété comme une conjonction de disjonctions de propriétés élémentaires, sachant que toute formule propositionnelle construite sur le langage $\mathcal{P}rop$ peut se mettre sous une telle forme dite clausale. Cette formulation permet de tenir compte de l'exemple précédent. Ce n'est en fait pas nécessaire, car nous disposerons bientôt pour cela des formules logiques temporelles qui sont encore plus générales.

Notons également que l'on peut se passer très simplement de la fonction d'étiquettage en définissant directement l'ensemble des états comme un sous-ensemble de l'ensemble des parties de $\mathcal{P}rop$. Cela permet d'écrire $p \in q$ au lieu de $p \in l(q)$, ce qui est bien sûr plus simple. Nous adopterons souvent cette simplification par la suite.

3.2 Produit synchronisé d'automates

Le produit synchronisé d'automates est un mécanisme visant à modulariser la description de systèmes complexes en enrichissant le langage de base. Considérons les compteurs modulo 2, 3 et 4 représentés à la figure 3.4.

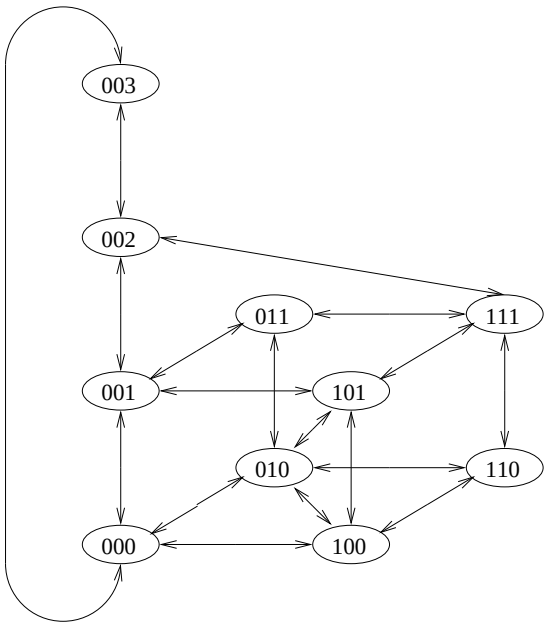


FIG. 3.5 – Partie du produit de trois compteurs modulo deux, trois et quatre.

L'automate produit synchronisé de ces trois compteurs est représenté (partiellement) à la figure 3.5. Il possède $2 \times 3 \times 4 = 24$ états. Dans chaque état, chaque compteur peut s'incrémenter, se décrémenter ou rester inchangé, ce qui fait $3^3 - 1 = 26$ possibilités. Il y aura donc $24 \times 26 = 624$ transitions dans le système de transition obtenu.

Cette construction sert en particulier à définir l'intersection de deux langages : dans ce cas là, ne sont retenues que les transitions qui font toutes $incr$ en même temps ou toutes $decr$ en même temps, ce qui engendre une grande économie du nombre d'états. Mais on pourrait synchroniser différemment, pour d'autres besoins : par exemple, synchroniser les deux premiers compteurs lorsqu'ils augmentent en interdisant au troisième d'évoluer, et les deux derniers lorsqu'ils diminuent en interdisant au premier d'évoluer. La prise en compte de telles possibilités conduit à la définition du produit synchronisé d'automates, ou il va être possible pour un sous-ensemble des automates considérés d'*attendre* que les autres automates atteignent une certaine configuration avant de déclencher un ensemble de transitions les impliquant éventuellement tous. On dira qu'ils se *synchronisent*.

Définition 3.2 *Étant donnés n automates non-déterministes $A_i = (V_i, Q_i, T_i, \mathcal{P}rop_i, l_i)$ pour $i \in [1..n]$, une* étiquette d'attente $- \notin \vee_i V_i$ *et un ensemble de synchronisation $Sync \subseteq \Pi_{1 \leq i \leq n}(V_i \cup \{-\})$, on appelle produit synchronisé des automates A_i par la synchronisation $Sync$ l'automate*

$$
\begin{aligned}
\Pi_{Sync} A_i &= (V, Q, T, \mathcal{P}rop, l) \\
V &= (V_1 \cup \{-\}) \times \ldots \times (V_n \cup \{-\}) \\
Q &= Q_1 \times \ldots \times Q_n \\
T((q_1, \ldots, q_n), (a_1, \ldots, a_n)) &= \{(q'_1, \ldots, q'_n) \mid q'_i \in T'_i(q_i, a_i)\} \\
T'_i(q_i, a_i \neq) &= T_i(q_i, a_i) \\
T'_i(q_i, -) &= \{q_i\} \\
\mathcal{P}rop &= \bigcup_{i \in [1..n]} \mathcal{P}rop_i
\end{aligned}
$$

$$l((q_1, \ldots, q_n)) \quad = \quad \bigvee_{i \in [1..n]} l(q_i)$$

Le produit de synchronisation d'automates est souvent noté $A_1||A_2||\ldots||A_n$, la contrainte de synchronisation étant supposée connue sans ambiguité. Dans la pratique, on n'est en général pas intéressé par la transition dans laquelle aucun automate ne progresse, et donc $-^n \notin sync$. Notons que l'étiquette $-$ agit exactement comme une transition vide sur l'état en attente, mais il n'est pas toujours possible de confondre une attente avec une transition vide.

Il est suggéré au lecteur de modifier cette définition en éliminant la fonction d'étiquettage.

L'automate produit cartésien défini dans la section 2.7 n'est rien d'autre qu'un produit synchronisé de 2 automates sur le même alphabet V avec la contrainte de synchronisation $sync = \{(a, a) \mid a \in V\}$. C'est la forme particulière de cette contrainte de synchronisation qui autorise l'utilisation du vocabulaire union des vocabulaires de départ au lieu du vocabulaire produit.

Dans l'exemple précédent des compteurs, la synchronisation possible annoncée se décrit par $sync = \{(incr, incr, -), (-, decr, decr)\}$. En pratique, il nous arrivera fréquemment de renommer les étiquettes du produit, opération dont la formalisation est laissée au lecteur.

On pourrait croire que le produit synchronisé n'est qu'un banal outil de description d'automates plus complexes. En fait, son pouvoir d'expression est très important puisque l'automate fini équivalent à un produit synchronisé est exponentiellement plus complexe que la description de départ. En effet, si les automates de départ on un nombre d'états respectivement égal à $k_1, \ldots, k_p$, le nombre d'états du produit est égal à $k_1 \times \ldots \times k_p$ qui est exponentiel en la variable p.

Si le nombre d'état d'un produit synchronisé est rapidement gigantesque, la contrainte de synchronisation a pour résultat que certain d'entre eux ne sont pas atteignables. Calculer efficacement l'ensemble des états atteignables est un problème bien sûr fondamental, mais difficile.

Dans notre exemple, tous les sommets sont atteignables, car pour i fixé, la décrémentation de j et k produit un cycle hamiltonien dans le plan $X = i$, du fait que les valeurs initiales de j et k sont premieres entre elles, et que l'on peut par ailleurs parcourir toutes les valeurs de i par incrémentation.

On appellera *graphe d'atteignabilité* l'automate obtenu à partir de l'automate produit lorsqu'on se restreint aux seuls états atteignables. Ce graphe joue un rôle central en vérification, car la sureté d'un système s'exprime comme la non-atteignabilité de certains états. Son calcul (efficace) est donc un problème essentiel.

Les constructions qui suivent font toutes peu ou prou appel à des produits synchronisés d'automates. Cela signifie que les concepts importants et l'algorithmique à développer sont esentiellement ceux des produits synchronisés. La vérification que les constructions à venir sont des cas particuliers de produits synchronisés sera généralement laissée au lecteur.

3.3 Automates à variables d'état et transitions gardées

Reprenons l'exemple du digicode. On souhaite maintenant contrôler le nombre d'erreurs d'un utilisateur de manière à diminuer le risque d'une ouverture accidentelle de la porte par un utilisateur essayant des codes au hasard. On va pour cela introduire un compteur qui va s'incrémenter à chaque erreur d'un utilisateur, et des gardes qui ne vont autoriser les transitions que lorsque qu'une certaine condition est satisfaite. Les transitions non gardées auront par convention la garde "True". Cette nouvelle modélisation est représentée à la figure 3.6, où l'on a noté par (M) la mise à jour du compteur qui est alors incrémenté d'une unité ($cpt := cpt + 1$).

On pourrait bien sûr ajouter des transitions en erreur en cas de dépassement du nombre d'erreurs autorisées. On pourrait également contrôler toutes les transitions par la garde $cpt \leq 3$ et mettre à jour le compteur de manière systématique en cas d'erreur dans le but d'obtenir un digicode plus réaliste. Cet exercice est recommandé au lecteur.

L'automate obtenu peut être "déplié" de manière à faire apparaître les comportements possibles du système, c'est-à-dire ceux qui satisfont les gardes. L'automate montré à la figure 3.7 est obtenu par dépliement de l'automate représentant un digicode plus réaliste où toutes les transitions sont gardées,

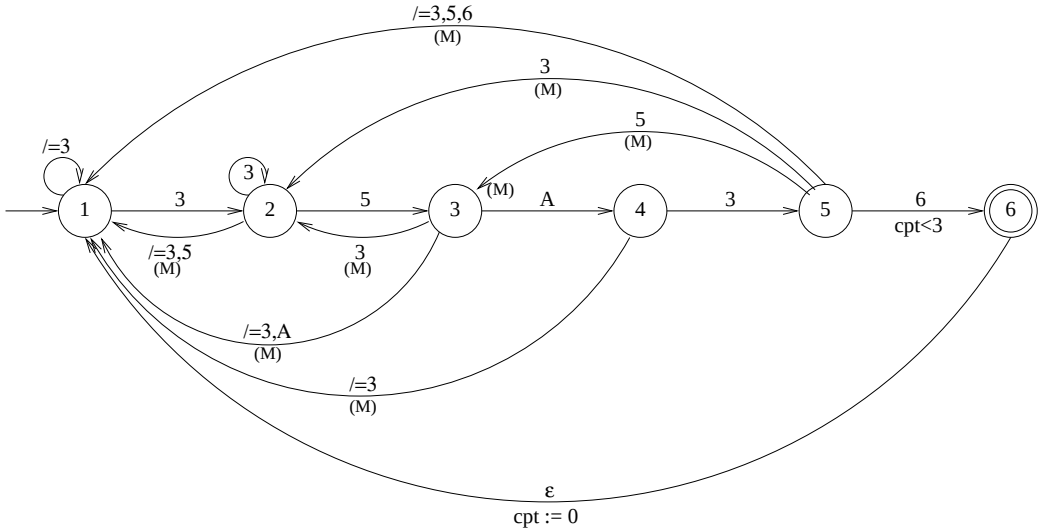


FIG. 3.6 – Digicode à utilisation multiples et contrôle d'erreurs.

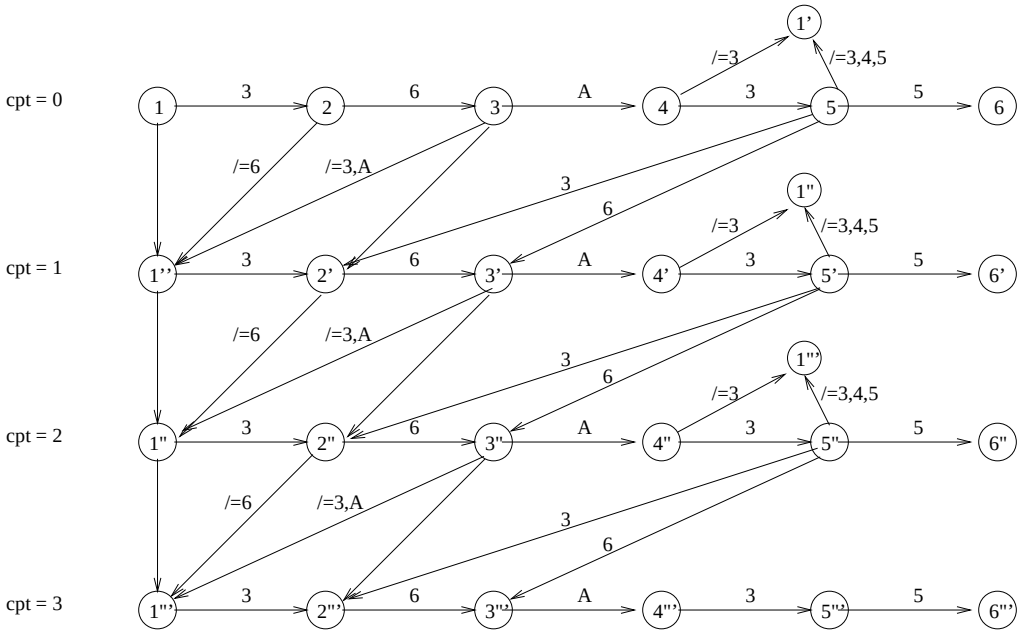


FIG. 3.7 – Système de transitions associé au digicode à utilisation multiples et contrôle d'erreurs.

et toutes les erreurs de frappe sur le clavier provoquent une incrémentation du compteur d'erreurs. Un tel automate est appellé *système de transition*, et ses états sont dits *globaux*. Si l'automate comporte n variables $X_1, \ldots, X_n$ prenant leurs valeurs dans des domaines $D_1, \ldots, D_n$, les états globaux seront ceux du produit cartésien $Q \times D_1 \times \ldots \times D_n$ tels que la garde associée à un état $q \in Q$ est vraie pour les valeurs $d_1, \ldots, d_n$ des variables dans l'état q.

Dans l'exemple considéré, l'état $1'$ est l'état $(1, cpt = 1)$. L'état 1 est appellé état de contrôle de l'état global 1'. Le fait qu'il n'y ait pas de transition sortante de l'état $4'''$ provient de l'absence de

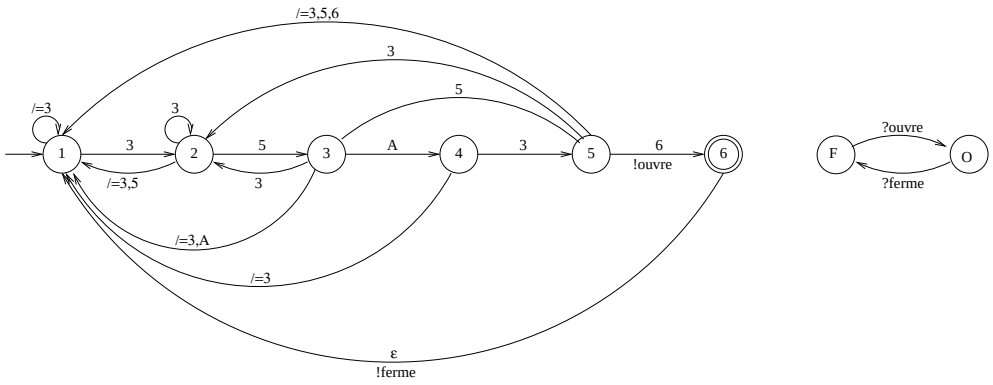


FIG. 3.8 – Le digicode comme couple clavier-porte.

transitions erreurs lorsque les gardes ne sont pas satisfaites. On pourrait bien sûr les rajouter.

La définition formelle d'un automate gardé et du système de transition associé est un exercice recommandé au lecteur. Tout aussi rfecommandé est la construction du système de transition associé à l'automate de la figure 3.6. Qu'observez-vous ?

3.4 Synchronisation par messages

Il s'agit d'un cas de produit synchronisé très important en pratique. On distingue parmi les étiquettes (lettres du vocabulaire),

- l'envoi d'un message m, noté $!m$;

- la réception du message m, notée $?m$.

Dans le produit synchronisé, on n'autorisera que les transitions où toute émission du message m est accompagnée de la réception de m (et vice-versa). Cette technique est illustrée avec la modélisation conjointe du digicode -dans sa version simplifiée- et de la porte qu'il contrôle à la figure 3.8. Notons que l'on peut avoir plusieurs étiquettes pour une même transition : dans notre exemple, une étiuette corresondant à la frappe d'une lettre sur le clavier, l'autre à la commande de porte. Cet exemple montre bien que le digicode est une système composé d'un clavier et d'une porte communiquant par messages.

On souhaite maintenant bien sûr vérifier que la porte ne s'ouvre que lorssque le code correct a été tapé sur le clavier. cela implique bien sûr un fonctionnement correct du modèle de digicode, de la porte, et de la communication entre les deux. Cette vérification (intuitive) est laissée au lecteur.

3.5 Synchronisation par variables partagées

Il s'agit là encore d'un enrichissement du langage de base qui peut se coder sous la forme d'un produit synchronisé. L'idée est qu'une même variable peut être partagée par plusieurs automates. On va utiliser cette technique pour modéliser un gestionnaire d'impressions.

On utilisera la lettre A pour figurer l'attente, la lettre I pour l'impression, et la lettre R pour le repos. On a les propriétés suivantes :

Aa : l'utilisateur a attend. Il a fait une requête non encore traitée ;

Ia : l'imprimante imprime le document demandé par a ;

Ra : l'utilisateur a est au repos : aucune requête n'est en attente ni en cours d'impression.

On souhaite s'assurer que :

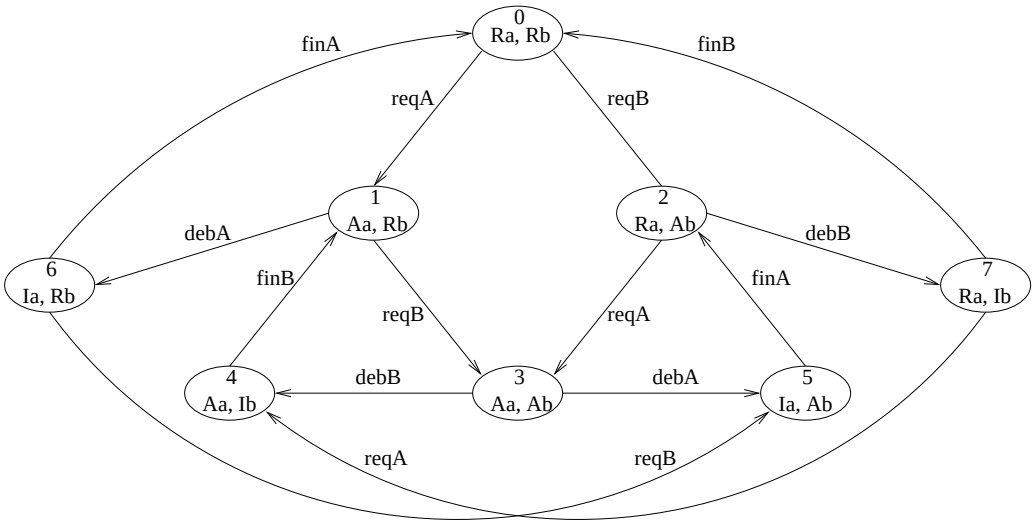


FIG. 3.9 – Gestionnaire d'impression.

FIG. 3.10 – Gestionnaire d'impression équitable.

- toute impression est précédée d'une requête appropriée : dans toute exécution, tout état qui valide I_A est précédé d'un état qui valide A_A ;

- toute demande d'impression finit par être satisfaite, ce qui n'est pas le cas de la modélisation proposée : le gestionnaire n'est pas *équitable*, il est possible de passer son temps à satisfaire les requêtes de l'utilisateur A sans jamais s'occupper de celles de l'utilisateur B.

Pour remédier à ce problème, on va décrire le gestionnaire comme un produit de 2 automates, un par utilisateur, se partageant une variable $tour$. Ce gestionnaire est représenté à la figure 3.10.

Ce produit est équivalent à un unique automate représenté à la figure 3.11 :

Cette modélisation n'est toujours pas satisfaisante, puisqu'elle ne permet pas aux utilisateurs d'imprimer deux fois de suite. On fait donc une dernière modification de notre modèle, décrit à la figure 3.12.

L'automate produit ainsi obtenu a pour états des quintuplets $(a, b, d_A, d_B, tour)$, ce qui nous fait donc $4 \times 4 \times 2 \times 2 \times 2 = 124$ états. On peut montrer qu'aucun état de la forme $(4, 4, -, -, ?)$ n'est atteignable, donc deux utilisateurs ne peuvent imprimer en même temps. On peut aussi montrer que cet automate est équitable, il répond donc au problème posé.

Cet exemple suggère que les variables partagées sont elles-même décrites par des automates, les états étant l'ensemble des valeurs qu'elles prennent.

Il est conseillé au lecteur de construire l'automate produit équivalent à la modélisation de la figure 3.12 et de formaliser les constructions que nous venons de décrire sous forme de produits synchronisés.

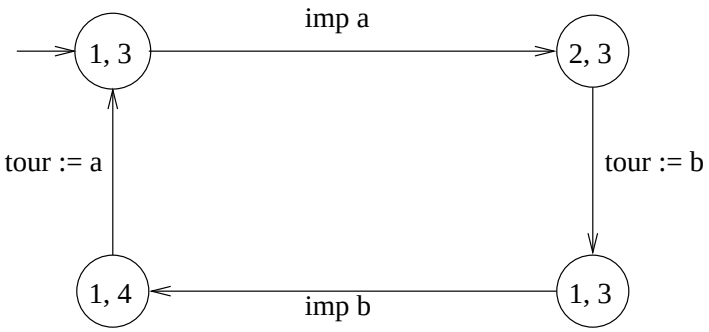


FIG. 3.11 – Automate produit équivalent au gestionnaire d'impression équitable.

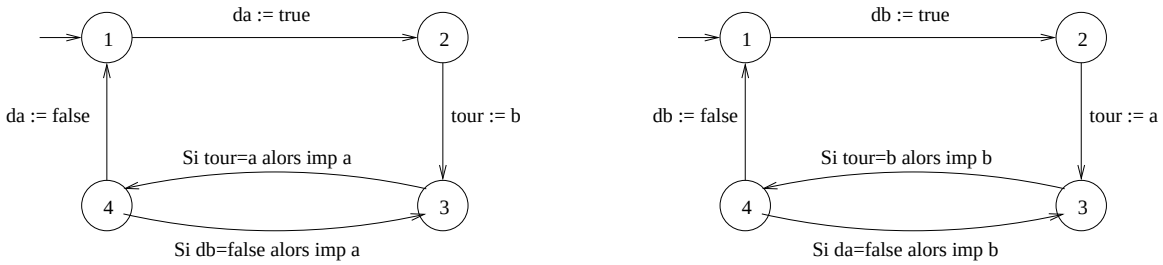


FIG. 3.12 – Gestionnaire d'impressions multiples équitable.

3.6 Conclusion

Il est apparu tout au long de ce chapitre que le produit de synchronisation était l'outil méthodologique essentiel de la modélisation, et que les nouvelles constructions que l'on peut vouloir ajouter pour faciliter la description pratique des systèmes réels n'étaient en fait que des cas particuliers d'automates synchronisés. Ces constructions sont néanmoins essentielles à une modélisation facile.

L'intérêt du produit de synchronisation n'est pas uniquement méthodologique, il est également théorique. Il s'avère en effet moins complexe (du point de vue de la théorie de la complexité) de vérifier une propriété pour un produit synchronisé que pour l'automate produit équivalent. Cela est dû à l'explosion en nombre d'états qui se produit lorsque l'on construit l'automate produit à partir d'un ensemble d'automates synchronisé.

3.7 Exercices

Exercice 3.1 *On se propose de modéliser un ascenseur à trois étages (dont le rez-de-chaussée). On décomposera l'ascenseur en*

- une cabine, qui monte ou descend suivant l'étage où elle se trouve en fonction des ordres du contrôleur ;

- une porte par étage, qui s'ouvre ou se ferme suivant les ordres du contrôleur ;

- et un contrôleur qui envoie des instructions d'ouverture/fermeture aux portes et de montée/descente à la cabine.

1. *On prendra en compte la possibilité de descendre 1 étage à la fois, ou bien 2 étages sans arrêt lorsque l'on est à l'étage 2, et de même de monter 2 étages d'un coup lorsqu'on est au*

rez-de-chaussée.

2. *On se propose maintenant de décrire un modèle plus réaliste dans lequel chaque étage possède un bouton d'appel, et la cabine possède trois boutons pour indiquer la destination des utilisateurs, un par étage à atteindre. On supposera que les usagers arrivent à une cadence suffisament faible pour qu'il ne soit jamais nécessaire de mémoriser une demande.*

   On devra à nouveau prendre en compte la possibilité de monter ou descendre 2 étages d'un coup à la condition suivante :

   (a) *Il n'y a pas d'appel à l'étage intermédiaire ;*

   (b) *Il n'y a pas d'appel à l'étage intermédiaire pour un étage situé dans la direction de déplacement de la cabine.*

3. *Que faudrait-il faire pour prendre en compte des arrivées nombreuses d'usagers voulant utiliser l'ascenseur ?*

Exercice 3.2 *Modéliser une* alarme haute *qui déclenche une alarme chaque fois que le message attendu d'étiquette msg arrive dans un intervalle de temps supérieur à 5 unités de temps.*

Modéliser une alarme basse *qui déclenche une alarme chaque fois que le message attendu d'étiquette msg arrive dans un intervalle de temps inférieur à 5 unités de temps.*

Exercice 3.3 *Exprimer sous forme d'un produit synchronisé un ensemble d'automates communiquant par variables partagées.*

Exercice 3.4 *La café des philosophes n'a qu'une table, et il n'accepte que des philosophes comme clients, à qui il sert invariablement des pates. La règle veut que chaque philosophe arrive avec sa fourchette, qu'il place à sa droite sur la table. Un philosophe peut aussi quitter la table avec sa fourchette, à condition qu'elle soit libre. Pour manger, il lui faut utiliser sa fourchette, et celle de son voisin de gauche. Il mange lorsqu'il a faim -à condition d'avoir les fourchettes-, sinon il pense.*

Décrire sous forme de produits synchronisés les situations suivantes :

1. *4 philosophes sont assis à une table de 4.*
2. *2 philosophes sont assis à une table de 8, et 6 philosophes attendent leur tour.*
3. *En ce jour d'affluence, 2 philosophes sont installés et 100 autres attendent leur tour. Tout philosophe qui a mangé doit quitter la table dès que possible, à condition qu'il reste au moins un philosophe à table ou que la file d'attente soit vide.*
4. *Après avoir mangé, un philosphe doit maintenant laver l'assiette et les deux fourchettes qu'il a utilisées avant de pouvoir partir.*
5. *Les philosophes en attente sont maintenant gérés en file d'attente.*

On se demande, dans chacun des casétudiés, s'il y a risque qu'un plisophe meurre de faim.

Chapitre 4

Logique Temporelle

La logique dite temporelle est une logique qui ne parle pas du temps, c'est sa première caractéristique. La seconde est qu'il y a plusieurs logiques temporelles, même s'il en existe une version généralisée qui capture les autres.

Dans l'exemple de l'ascenseur développé à la première question de l'exercice 3.1, dont la solution est représentée à la figure 4.1 dans laquelle on a fait figurer explicitement l'ensemble de synchronisation de l'automate produit équivalent à l'automate avec messages de synchronisation, on s'intéresse à plusieurs types de propriétés :

- Tout appel de l'ascenseur doit finir par être satisfait, c'est une propriété dite d'*équité* ;
- La porte de l'ascenseur est ouverte à l'étage n si et seulement si l'ascenseur est en train de desservir l'étage n, c'est une propriété dite de *sureté* ;
- L'ascenseur ne traverse jamais un étage pour lequel existe un appel sans le satisfaire, c'est une propriété de *rendez-vous* ;

Notons que notre modèle d'ascenseur ne distingue pas la commande de descente d'un seul étage de celle de descente de deux étages, il décide donc de manière non-déterministe.

Ces propriétés portent sur le comportement dynamique du système. On pourrait les exprimer en logique du premier ordre, à l'aide d'une variable t interprétée comme étant le temps :

$$\forall t\, \forall n\, (App(n,t) \Longrightarrow \exists t' > t\, Dess(n,t))$$

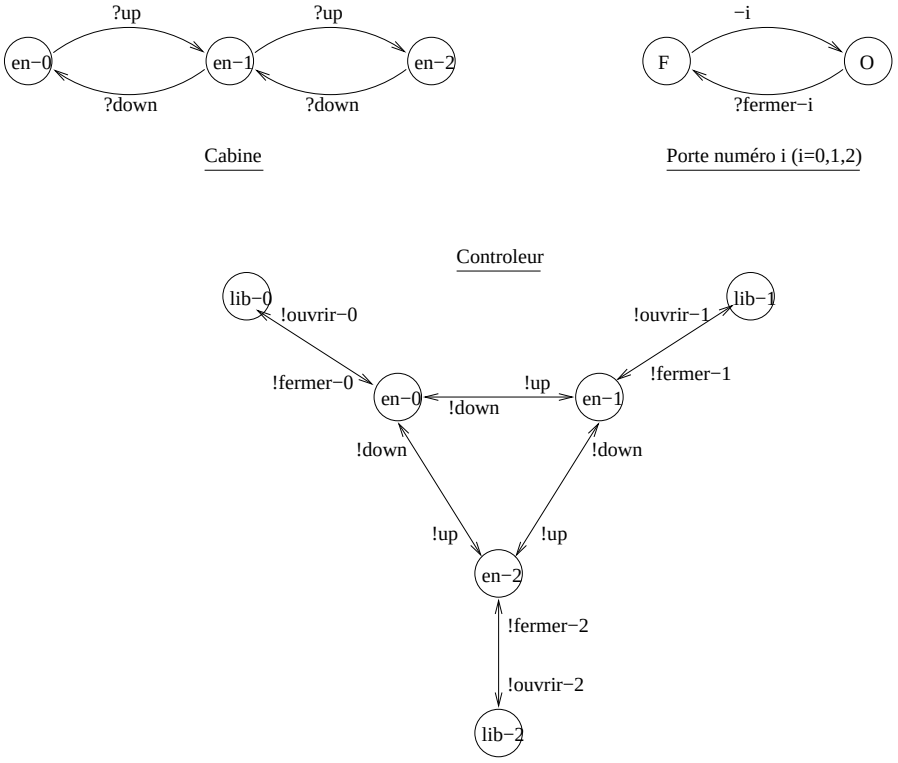
$$\forall t \forall n\, (\neg Ferme(n,t) \Longleftrightarrow (Dess(n,t) \wedge \forall n' \neq n\, Ferme(n',t))$$

$$\forall t\, \forall n\, (App(n,t) \Longrightarrow (\exists t' (t \leq t') \wedge H(t') = n \wedge \forall t'' (t \leq t'' < t'') \Longrightarrow H(t'') \neq n)$$

en notant $App(n,t)$ le prédicat figurant l'appel de l'étage n à l'instant t, $Dess(n,t)$ le prédicat figurant la desserte de l'étage n à l'instant t, $H(t)$ la position de l'ascenseur à l'instant t, et $Ferme(n)$ le prédicat indiquant que la porte à l'étage n est fermée.

Mais la logique du premier ordre a deux désavantages : la lourdeur des spécifications, et l'indécidabilité des énoncés à cause de la quantification universelle (sur le temps, dans l'exemple ci-dessus).

La logique temporelle est beaucoup plus simple, mieux adaptée au problème, et décidable pour les fragments considérés. En fait, le temps en est absent, remplacé par un ordre qui en symbolise l'écoulement tout en faisant abstraction de la durée. C'est une logique parfaitement adaptée à l'énoncé de propriétés portant sur les exécutions d'un système en l'absence de contraintes de temps (ou plus généralement de contraintes portant sur des variables prenant leur valeur dans un ensemble dense). En fait, la logique temporelle peut se traduire dans un fragment décidable de la logique classique dont la caratérisation n'est pas simple.



L'ensemble de synchronisation obtenu pour le produit Porte0xPorte1xPorte2xCabinexControleur est le suivant:

Sync ={ (?ouvrir−0,−,−,−,!ouvrir−0), (?fermer−0,−,−,−,!fermer−0),
(?ouvrir−1,−,−,−,!ouvrir−1), (?fermer−1,−,−,−,!fermer−1),
(?ouvrir−2,−,−,−,!ouvrir−2), (?fermer−2,−,−,−,!fermer−2),
(−,−,−,?down,!down), (−,−,?up,!up)}

FIG. 4.1 – Ascenseur à trois étages.

4.1 La logique temporelle CTL*

4.1.1 Syntaxe

C'est une logique temporelle propositionnelle très générale qui a pour but d'exprimer des propriétés concernant les exécutions d'un automate $\mathcal{A} = (V, Q, q_0, T, \mathcal{P}rop, l)$, et dont

1. les propriétés atomiques sont celles de $\mathcal{P}rop$;
2. les combinateurs logiques sont ceux de la logique propositionnelle : Si ϕ, ϕ_1 et ϕ_2 sont des formules, alors $\neg\phi$, $\phi_1 \wedge \phi_2$ et $\phi_1 \vee \phi_2$ sont des formules, $\neg\phi$ étant dite négative, les autres formules étant positives ;
3. les combinateurs temporels permettent de construire des expressions vérifiées dans certains états de l'automate le long d'une exécution donnée ou le long de certaines exécutions de l'automate. Si ϕ est une formule temporelle, alors
   - $X\phi$ (dire $next\ \phi$) est une formule temporelle, qui indique que la formule ϕ est vérifiée dans l'état qui suit l'état courant dans l'exécution courante ;
   - $\phi_1 U \phi_2$ (dire $\phi_1\ until\ \phi_2$) est une formule temporelle qui indique que ϕ_1 est vérifiée dans tout état atteint dans le futur à partir de l'état courant et précédent l'état à partir duquel ϕ_2 devient vérifiée dans l'exécution courante ;

   Ces formules s'interprètent donc *le long d'une exécution donnée*. Deux quantificateurs permettent de parler d'exécutions différentes :
   - $A\phi$ (dire $for\ all\ path\ \phi$) est une formule temporelle qui indique que la formule ϕ est vérifiée le long de toute exécution issue de l'état courant ;
   - $E\phi$ (dire $for\ some\ path\ \phi$) est une formule temporelle qui indique que la formule ϕ est vérifiée le long d'au moins une exécution issue de l'état courant.

Le langage ci-dessus est souvent enrichi de nouvelles constructions utiles en pratique qui s'expriment à l'aide des précédentes :

- $F\phi = true\ U\ \phi$ (dire $eventually\ \phi$) est une formule temporelle qui indique que la formule ϕ est vérifiée dans au moins un état atteint dans le futur à partir de l'état courant dans l'exécution courante ; F est souvent notée par le symbole diamant $\Diamond$;
- $G\phi = \neg F \neg\phi$ (dire $henceforth\ \phi$ -désormais enfrançais) est une formule temporelle qui indique que la formule ϕ est vérifiée dans tout état atteint dans le futur à partir de l'état courant dans l'exécution courante ; G est souvent notée par le symbole carré $\Box$;
- $\phi_1 W \phi_2$ (dire $\phi_1\ until\ possibly\ \phi_2$) est vérifiée dans tout état atteint dans le futur à partir de l'état courant et précédent l'état s'il existe à partir duquel ϕ_2 devient vérifiée dans l'exécution courante. Le combinateur W est aussi appellé "Weak Until" ;
- $\overset{\infty}{F}\ \phi$, abbréviation de $GF\phi$, est vérifiée une infinité de fois dans le futur à partir de l'état courant dans l'exécution courante ;
- et $\overset{\infty}{G}\ \phi$, abbréviation de $FG\phi$, qui est vérifiée dans tout état futur à partir d'un certain état de l'exécution courante atteint à partir de l'état courant.

On peut s'étonner que la logique temporelle ne parle que du futur, et pas du passé. On peut en fait l'enrichir avec de nouveux combinateurs parlant des calculs passés, ces combinateurs ne s'exprimant pas en fonction des combinateurs X et U.

4.1.2 Sémantique

Les modèles d'une formule temporelle sont des automates avec des propriétés dans les états, et sans vocabulaire, de telle sorte que la fonction de transition T est une application de Q dans $\mathcal{P}(Q)$, également appellés *structures de Kripke*. L'automate $\mathcal{A} = (V, Q, q_0, T, \mathcal{P}rop, l)$ étant donné, on écrira

$$\mathcal{A}, u, i \models \phi$$

$\mathcal{A}, u, i$	$\models$	p	ssi	$p \in u(i)$
$\mathcal{A}, u, i$	$\models$	$\neg\phi$	ssi	$\mathcal{A}, u, i \not\models \phi$
$\mathcal{A}, u, i$	$\models$	$\phi \wedge \psi$	ssi	$\mathcal{A}, u, i \models \phi$ et $\mathcal{A}, u, i \models \psi$
$\mathcal{A}, u, i$	$\models$	$\phi \vee \psi$	ssi	$\mathcal{A}, u, i \models \phi$ ou $\mathcal{A}, u, i \models \phi$
$\mathcal{A}, u, i$	$\models$	$X\ \phi$	ssi	$i <\mid u \mid$ et $\mathcal{A}, u, i+1 \models \phi$
$\mathcal{A}, u, i$	$\models$	$F\phi$	ssi	$\exists j < \omega$ avec $i \leq j \leq \mid u \mid$ tel que $u, j \models \phi$
$\mathcal{A}, u, i$	$\models$	$G\phi$	ssi	$\forall j < \omega$ tel que $i \leq j \leq \mid u \mid$ alors $u, j \models \phi$
$\mathcal{A}, u, i$	$\models$	$\phi U \psi$	ssi	$\exists j < \omega$ avec $i \leq j \leq \mid u \mid$ tel que $u, j \models \psi$ et $\forall k, i \leq k < j,\ u, j \models \phi$
$\mathcal{A}, u, i$	$\models$	$E\ \phi$	ssi	$\exists v$ prolongeant u tel que $v, i \models \phi$
$\mathcal{A}, u, i$	$\models$	$A\ \phi$	ssi	$\forall v$ prolongeant u alors $v, i \models \phi$
$\mathcal{A}$	$\models$	ϕ	ssi	pour toute exécution u, on a $\mathcal{A},\ u, 0 \models \phi,$ c'est-à-dire
$\mathcal{A}$	$\models$	ϕ	ssi	$\mathcal{A}, -, 0 \models A\phi$

FIG. 4.2 – Sémantique des formules temporelles.

pour exprimer la validité de la formule ϕ dans l'état $i < \omega$ (on dit aussi à l'instant i) du calcul (on dit aussi l'exécution) u de l'automate $\mathcal{A}$. En pratique, $\mathcal{A}$ est souvent omis. Le problème, étant donné un automate, de décider la validité d'une formule vis à vis de cet automate est souvent appellé *model-checking* ou, en français, *vérification*. Il est très différent de la satisfiabilité d'une formule, problème plus complexe pour lequel il s'agit de décider s'il existe un automate $\mathcal{A}$ qui rende la formule valide.

Une exécution est une suite finie ou infinie d'états, c'est-à-dire une application de $[0..n]$ dans Q pour un certain ordinal $n \leq \omega$. On note $\mid u \mid = n$ la longueur de la suite u. Notons que la valeur $u(0)$ est donnée, et que $u(i+1) \in T(u(i))$. En général, on prendra $u(0) = q_0$. La possibilité d'avoir à la fois des calculs finis et infinis complique un peu les choses, car il faudra en permanence s'assurer que la notation $u(i)$ a un sens.

L'interprétation des formules de logique temporelle est donnée à la figure 4.2, conformément à la signification intuitive déja donnée de leur syntaxe.

Notons que la définition de until n'impose pas à la formule ϕ de devenir fausse lorsque ψ devient vrai. Des définitions alternatives sont possibles, la présente étant acceptée comme la plus commode.

Définition 4.1 *On dit que $\mathcal{A}$ est un modèle de ϕ si $\mathcal{A} \models \phi$.*

En CTL* , le temps est "built-in", à l'inverse de la logique du premier ordre. On le manipule avec un nombre réduit de combinateurs bien choisis, et les actions ont lieu à des instants $0, 1, 2, \ldots, n, \ldots$ multiples d'un temps élémentaire. Ce n'est donc pas un modèle *asynchrone*, qui suppose un temps dense, mais un modèle *synchrone*.

Notons que $\mathcal{A}, u, i \models \neg\phi$ ssi $\mathcal{A}, u, i \not\models \phi$ par définition, mais $\mathcal{A} \not\models \phi$ n'implique pas $\mathcal{A} \models \neg\phi$, car il peut exister deux exécutions u et v telles que $u, 0 \models \phi$ et $v, 0 \models \neg\phi$.

Un exercice utile consistera à *calculer* les définitions de $\mathcal{A}, u, i \models \overset{\infty}{F}\ \phi$ et $\mathcal{A}, u, i \models \overset{\infty}{G}\ \phi$ à partir de celles de $F\phi$ et $G\phi$.

4.1.3 Exemple

Voyons maintenant un exemple, description (très) approchée du climat suivant les saisons, figurée dans l'automate $\mathcal{A}_{temps}$ de la figure 4.3. Chaque état comporte deux informations, son nom, puis une propriété écrite en dessous parmi l'ensemble {chaud, doux, froid, caniculaire, glacial}. Une

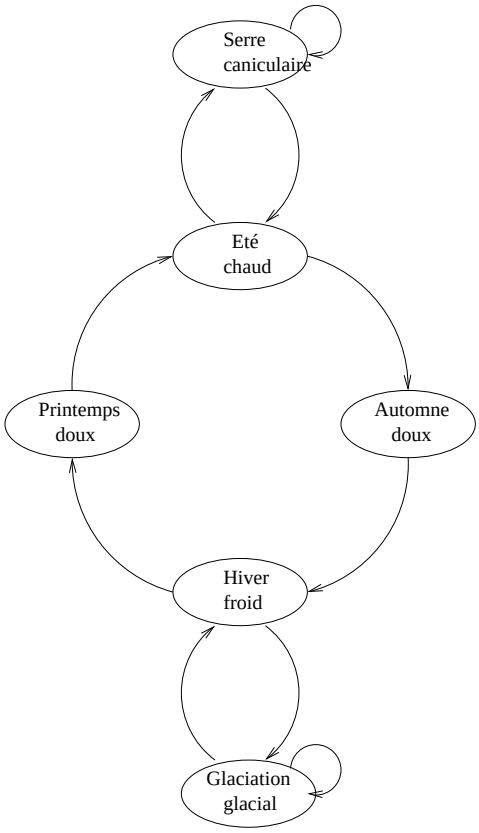


FIG. 4.3 – Le temps suivant les saisons.

propriété qui est clairement vérifiée par cette modélisation est qu'il y a un risque potentiel de climat glacial définitif dans le futur, ce qui s'exprime par la propriété :

$$E\ F\ G\ \text{glacial} \quad \text{ou encore} \quad A\ E\ F\ E\ G\ \text{glacial}$$

On pourrait également exprimer le risque potentiel de climat "semi-extrême", défini comme la disparition des inter-saisons :

$$E\ F\ G\ \neg\text{doux} \quad \text{ou encore} \quad A\ E\ F\ E\ G\ \neg\text{doux}$$

Une autre propriété de notre modèle est qu'il existe une possibilité que le climat recommence de fonctionner comme aujourd'hui, en alternant les saisons, après avoir subi une période extrême :

$$E\ F\ ((glacial \vee caniculaire) \wedge FG\ (\neg\text{doux} \Longrightarrow X\ \text{doux}))$$

Montrons maintenant que notre modèle valide la possibilité que les saisons continuent à alterner, en supposant que nous sommes en hiver, propriété exprimée par la formule

$$E\ G\ \neg\text{doux} \Longrightarrow X\ \text{doux}$$

La définition de la sémantique des formules nous donne les vérifications successives :

$$\begin{array}{rcll} \mathcal{A}_{temps} & \models & E\,G\,\neg\text{doux} \Longrightarrow X\ \text{doux} & \\ \mathcal{A}_{temps}, u, \text{hiver} & \models & E\,G\,\neg\text{doux} \Longrightarrow X\ \text{doux} & \\ \mathcal{A}_{temps}, v_{hiver}, \text{hiver} & \models & G\,\neg\text{doux} \Longrightarrow X\ \text{doux} & \text{où } v_s \text{ est le cycle saisonnier issu de l'état s} \end{array}$$

calcul qui se décompose en une infinité de calculs de l'une des 4 formes

$$\begin{array}{rcl} \mathcal{A}_{temps}, v_{hiver}, \text{hiver} & \models & \neg\text{doux} \Longrightarrow X\ \text{doux} \\ \mathcal{A}_{temps}, v_{printemps}, \text{printemps} & \models & \neg\text{doux} \Longrightarrow X\ \text{doux} \\ \mathcal{A}_{temps}, v_{ete}, \text{été} & \models & \neg\text{doux} \Longrightarrow X\ \text{doux} \\ \mathcal{A}_{temps}, v_{automne}, \text{automne} & \models & \neg\text{doux} \Longrightarrow X\ \text{doux} \end{array}$$

Le premier de ces calculs se décompose en la disjonction des deux calculs qui suivent :

$$\begin{array}{rcl} \mathcal{A}_{temps}, v_{hiver}, \text{hiver} & \models & X\ \text{doux} \\ \mathcal{A}_{temps}, v_{hiver}, \text{hiver} & \models & \neg\neg\text{doux} \end{array}$$

dont le premier conduit au résultat

$$\mathcal{A}_{temps}, v_{printemps}, \text{printemps} \models \text{doux}$$

permettant d'abandonner le second.

Le second calcul se décompose en une disjonction similaire :

$$\begin{array}{rcl} \mathcal{A}_{temps}, v_{printemps}, \text{printemps} & \models & X\ \text{doux} \\ \mathcal{A}_{temps}, v_{printemps}, \text{printemps} & \models & \neg\neg\text{doux} \end{array}$$

et le second calcul conduit au résultat :

$$\begin{array}{rcl} \mathcal{A}_{temps}, v_{printemps}, \text{printemps} & \not\models & \neg\text{doux} \\ \mathcal{A}_{temps}, v_{printemps}, \text{printemps} & \models & \text{doux} \end{array}$$

permettant d'abandonner le premier.

Les deux autres calculs sont similaires, ce qui termine notre preuve, dans laquelle nous avons utilisé la notation $\mathcal{A}_{temps}, u, u(i) \models \phi$ au lieu de $\mathcal{A}_{temps}, u, i \models \phi$.

Nous avons bien sûr triché, puisque nous n'avons vérifié la popriété que pour un tronçon de chemin de longueur 4. Mais il est bien clair que cela est suffisant, puisque nous avons en fait montré la propriété pour tout sommet rencontré le long du chemin. La justification formelle de ce racourci découlera des chapitres qui suivent. En fait, vérifier une formule de CTL* est complexe, on le fera pour deux sous-classes particulières de CTL* après un détour qui nous conduira aux automates de Büchi sur les mots infinis, chapitre qui sera lui-même précédé de rappels sur les classes de complexité polynomiales en temps et en espace.

Exercice : exprimer les propriétés d'équité, sureté et rendez-vous de l'ascenseur. Sont-elles vérifiées ?

4.2 Choix d'une logique temporelle

CTL* est une logique très expressive, qui suffit amplement aux besoins pratiques (sous l'hypothèse de synchronisme des actions). En fait, on peut montrer que tout combinateur dont la sémantique se définit de manière analogue à X, F, G, U (c'est-à-dire, comme une formule du premier ordre batie sur le seul prédicat $\leq$) s'exprime en fonction des seuls combinateurs X et U. C'est en particulier le cas des combinateurs $F, G, \overset{\infty}{F}, \overset{\infty}{G}, W$ que nous avons déja introduits. C'est le cas d'autres qui seront étudiés en exercice. Notre logique temporelle est donc redondante, et l'on peut se demander pourquoi avoir une syntaxe redondante ? Il y a deux réponses à cette question :

- certaines propriétés s'expriment plus naturellement avec des combinateurs "de haut niveau", et les formules sont plus lisibles et plus compactes ;

- la redondance peut disparaître si l'on considère des fragments particuliers de CTL* , comme le fragment où les formules sont baties sans U, mais avec X, F, G. Ces fragments sont souvent apparus avant la formulation de CTL* , et ont des propriétés de décision intéressantes.

Par la suite, nous allons considérer deux fragments particuliers importants de CTL* , pour lesquels la complexité du problème de vérification $\mathcal{A} \models \phi$ se fait respectivement en temps linéaire (en les tailles de ϕ et de $\mathcal{A}$) et PSPACE (en la taille de ϕ). On s'intéressera bien sûr à des fragments

suffisament expressifs pour permettre d'exprimer les propriétés importantes classiques, comme :

- l'invariance : $AG\phi$;

- l'inévitabilité une infinité de fois : $A \overset{\infty}{F} \phi$ ou $E \overset{\infty}{F} \phi$ (formules équivalentes à $AGEF\phi$ et $EGEF\phi$ respectivement) ;

- la potentialité de vérifier une formule : $AGEF\phi$

- etc.

Notons que la complexité du problème de vérification pour CTL* est également en $PSPACE$, nous y reviendrons.

4.2.1 CTL

En CTL, chaque occurrence d'un combinateur temporel doit être immédiatement sous la portée d'un quantificateur de chemin.

Par exemple, AGp et $AGEFp$ sont des formules de CTL, mais $AGFp$ et $\neg A \neg Gp$ ne sont pas des formules de CTL.

Les quatre combinateurs de CTL sont donc tout simplement $EX, AX, E\_U\_$ et $A\_U\_$. Il n'est en effet pas nécessaire de disposer de F et G, puisque $AF\phi$ et $EF\phi$ ne sont autres que $A\,true\,U\phi$ et $E\,true\,U\phi$; de même, $AG\phi$ et $EG\phi$ s'expriment comme $\neg EF\neg\phi$ et $\neg AF\neg\phi$; si ϕ est une formule de CTL, il en sera donc de même de toutes ces expressions.

Notons qu'il est nécessaire de disposer à la fois de A et E, car $\neg E\neg\phi$ n'est pas en général une formule de CTL même si ϕ l'est. CTL s'avère néanmoins assez expressive pour un bon nombre de besoins pratiques, notons en particulier les expressions représentées à la figure 4.4.

Le dernier exemple de la figure 4.4 montre une faiblesse de CTL : certaines propriétés ne s'expriment pas de manière très naturelle. Par exemple, la propriété exprimée par la formule de CTL* $EpU(qUr)$ s'exprime par la formule logiquement équivalente de CTL $EpU(EqUr)$. On est ainsi souvent amené à insérer des quantificateurs de chemins inutiles dans les formules.

Pour les propriétés ϕ de CTL* qui ne sont pas exprimables en CTL, on cherchera à les approximer par une formule ϕ' telle que

$$\begin{array}{rcll} \mathcal{A} \models_{CTL} \phi' & \Longrightarrow & \mathcal{A} \models_{\text{CTL}^*} \phi & \text{(renforcement)} \\ \mathcal{A} \models_{CTL} \neg\phi' & \Longrightarrow & \mathcal{A} \models_{\text{CTL}^*} \neg\phi & \text{(affaiblissement)} \end{array}$$

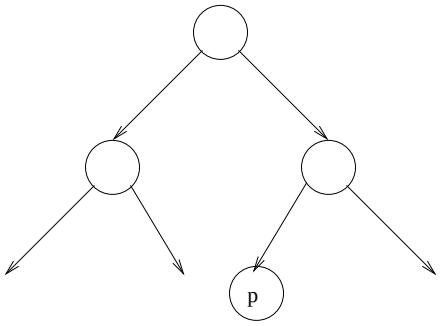
Par exemple, pour $\phi = EGFp$, on pourra prendre la propriété plus forte $\phi' = EGp$ ou la propriété plus faible $\neg AF\neg EFp$. Les automates de la figure 4.5 ont pour but de montrer que ces propriétés ne coincident pas avec ϕ.

La grande faiblesse de CTL est de ne pas pouvoir faire référence à une même exécution lorsque l'on emboite plusieurs combinateurs temporels, comme dans $EGFp$. Cette obligation de toujours quantifier sur les futurs posibles à partir de chaque état atteint dans une exécution donnée fait que la signification de la formule ne dépend que de l'état courant, on appelle ces formules des formules d'état, on y reviendra dans le chapitre 7.

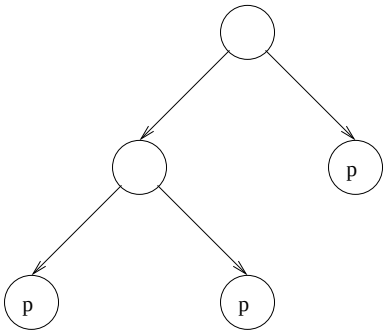
4.2.2 LTL

En LTL (on dit aussi PLTL), on autorise X et U sans restriction, mais on interdit A et E. LTL permet de faire référence à toutes les exécutions, mais pas à la manière dont elles sont organisées en arbre. On parle pour cette raison de logique du temps linéaire, d'où son nom.

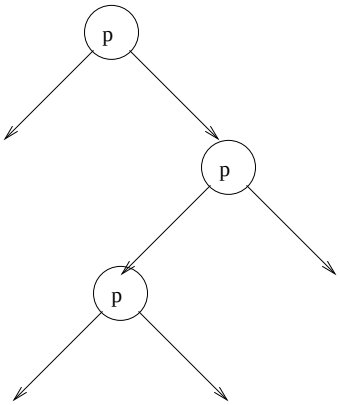
La propriété "ϕ est toujours potentiellement vérifiée" qui s'exprime en CTL* par la formule (de CTL) $AGEF\phi$ n'est plus exprimable en LTL. D'une manière générale, LTL n'est pas adaptée à l'expression des propriétés d'équité. Néanmoins, et en dépit de la complexité plus élevée de la vérification en LTL, on la préferre souvent à CTL car la pluspart des propriétés usuelles sont plus facilement exprimables.



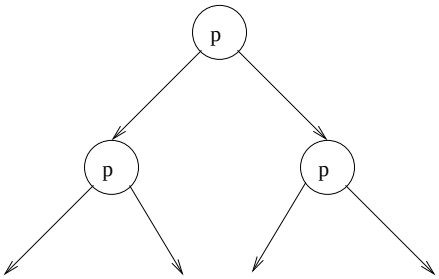
E F p



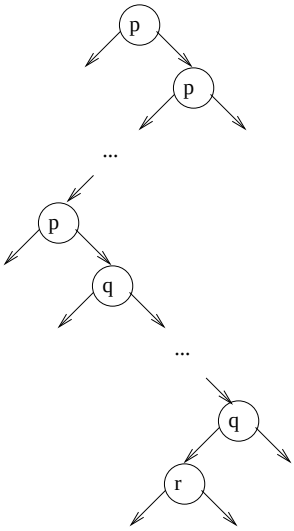
A F p



E G p



A G p



E p U E q U r

FIG. 4.4 – Expressivité de CTL.

FIG. 4.5 – Affaiblissement et renforcement.

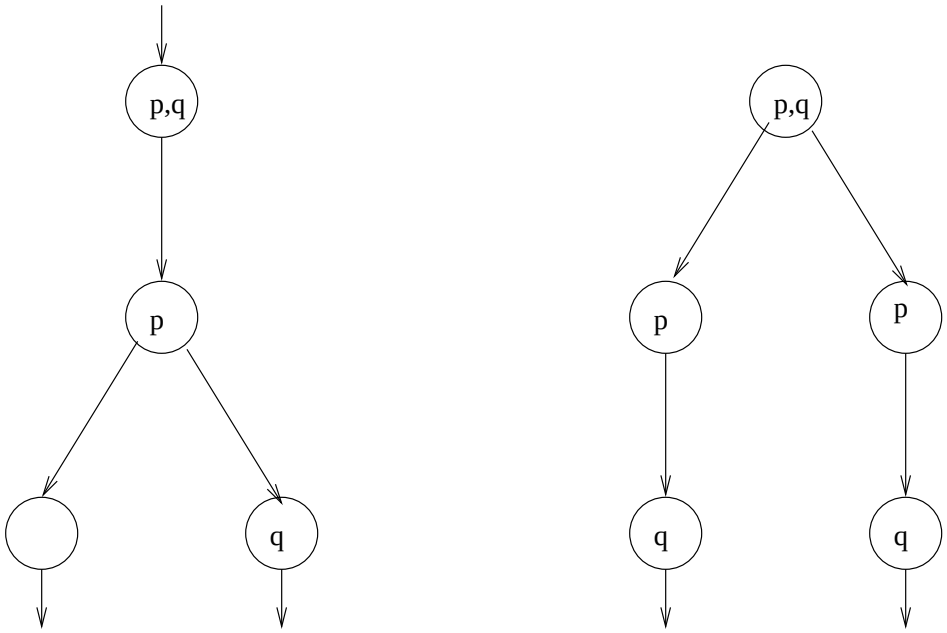


FIG. 4.6 – Automates non distinguables par des formules de LTL.

Des automates comme ceux de la figure 4.6 qui ont le même ensemble de chemins ne pourront donc pas être distingués en LTL. Dans le cas précédent, on peut bien sûr distinguer ces automates en CTL par la formule $EX(p \wedge AXq)$ qui est fausse du premier mais vraie du second.

4.3 Logique temporelle et automates

Les formules de logique temporelle servent à décrire les popriétés des calculs effectués par des automates. On peut donc se demander si, étant donné un automate $\mathcal{A}$, il existe une formule temporelle qui caractérise exactement les calculs de $\mathcal{A}$? Par exemple, les automates de la figure **??** admettent une caratérisation évidente par les formules respectives $A(p \iff Xp)$ et $A((pU\neg vp) \vee Gp)$. Ce

calcul semble pouvoir se faire de manière systématique, mais comment ? Le lecteur est invité à se poser cette question dès maintenant.

4.4 Exercices

Exercice 4.1 *On notera $\phi \equiv \psi$ si ϕ et ψ ont les mêmes modèles. On demande de vérifier la véracité des énoncés suivants :*

1. $X\ \neg\phi \equiv \neg X\phi$
2. $X\ F\phi \equiv F\ X\phi$
3. $X\ G\phi \equiv G\ X\phi$
4. $X(\phi \wedge \psi) \equiv (X\phi \wedge X\psi)$
5. $X(\phi \vee \psi) \equiv (X\phi \vee X\psi)$
6. $(X\phi)U\psi \equiv \psi \vee X(\phi\, U(\phi \wedge \psi))$
7. $\neg F\phi \equiv G\neg\phi$
8. $\neg G\phi \equiv F\neg\phi$
9. $\neg A\phi \equiv E\neg\phi$
10. $\neg E\phi \equiv A\neg\phi\ A\ G\ F\ p \equiv A\ G\ A\ Fp$
11. $A\ F\ G\ p \equiv A\ F\ A\ Gp$
12. $A\ F\ G\ p \equiv A\ F\ E\ Gp$ *item* $E\ G\ F\ p \equiv E\ G\ E\ Fp$
13. $E\ F\ G\ p \equiv E\ F\ E\ Gp$
14. $E\ F\ G\ p \equiv E\ F\ A\ Gp$

Exercice 4.2 *On demande de montrer que le combinateur temporel U n'est pas associatif.*

Exercice 4.3 *On se propose dans cet exercice d'examiner la signification des connecteurs temporels.*

1. *Exprimer la sémantique de la formule $false\, U\phi$;*
2. *Exprimer la sémantique de la formule $true\, U\phi$;*
3. *Exprimer la sémantique de la formule $\neg(false\, U\neg\phi)$;*
4. *Exprimer les connecteurs temporels F, G et W à l'aide des connecteurs X, U de manière à ce que la traduction respecte la sémantique des formules donnée dans le tableau 4.2.*

On se donne maintenant un nouveau connecteurs until, noté $\overset{\infty}{U}$ dont la sémantique est définie par :

$$u, i \models \phi\ \overset{\infty}{U}\ \psi \text{ ssi } \exists j < \omega \text{ avec } i \leq j \leq |\, u\, | \text{ tel que } \begin{cases} \forall k\ j \leq k \leq |\, u\, | & u, j \models \phi \\ \forall k\ i \leq k < j & u, j \models \phi \end{cases}$$

5. *Exprimer le connecteur $\overset{\infty}{U}$ en fonction des connecteurs X, U ;*
6. *Exprimer les connecteurs F, G en fonction des connecteurs $X, \overset{\infty}{U}$;*
7. *Peux-t-on exprimer le connecteur U en fonction des connecteurs $X, \overset{\infty}{U}$?*

Exercice 4.4 *On se propose de vérifier le bon fonctionnement de l'automate de la figure 4.1 en énonçant les propriétes voulues en logique temporelle avant de les évaluer sur l'automate. Soient*
P1 : la porte de l'étage i ne peut s'ouvrir si la cabine n'est pas à l'étage i ;
P2 : la cabine ne peut se déplacer si l'une des portes est ouverte.

1. *Donner un ensemble de propriétés élémentaires permettant d'énoncer ces deux propriétés.*

2. *Étiqueter les états par des sous-ensembles de l'ensemble des propriétés.*
3. *Donner l'énoncé de P1 en logique temporelle ;*
4. *Donner l'énoncé de P2 en logique temporelle ;*
5. *Vérifier P1 sur l'automate ;*
6. *Vérifier P2 sur l'automate ;*

Exercice 4.5 *On se propose d'examiner l'écriture de propriétés en CTL. Soit ϕ une formule de CTL. On demande d'exprimer*

1. *l'équité (un évènement se produit dans le futur quelque soit l'évolution du système) ;*
2. *le rendez-vous : quelque soit l'évolution du système, une certaine propriété se produira ;*
3. *la causalité : quelque soit l'évolution du système, si une certaine propriété devient vraie un jour, alors une autre propriété sera nécessairement vraie dans le futur ;*
4. *l'inévitabilité de ϕ une infinité de fois le long d'un chemin ;*
5. *l'inévitabilité de ϕ une infinité de fois le long de tout chemin ;*

Enfin, on demande d'exhiber une formule de CTL* non exprimable en CTL.

Exercice 4.6 *On se propose d'examiner l'écriture de propriétés en LTL. Soit ϕ une formule de LTL. On demande d'exprimer*

1. *l'équité ;*
2. *le rendez-vous ;*
3. *la causalité ;*
4. *l'inévitabilité de ϕ une infinité de fois le long d'un chemin ;*
5. *l'inévitabilité de ϕ une infinité de fois le long de tout chemin.*

Enfin, on demande d'exhiber une formule de CTL* non exprimable en LTL.

Chapitre 5

Complexité en temps et en espace

On définit la complexité des langages relativement à un modèle de calcul particulier, les machines de Turing. Cela peut sembler limiter la portée des constructions, mais on s'aperçoit vite que ce modèle résiste à de nombreuses variations, et que tous les modèles de la calculabilité connus donnent peu ou prou les mêmes notions de complexité à un facteur polynomial près : ce qui est linéaire dans un modèle peut très bien devenir quadratique dans un autre !

On peut se demander si la notion de langage est bien celle qui convient, puisque l'on est en général intéressé par des *problèmes*, et non des langages. En fait, tout problème peut se coder sous la forme d'un langage sur un alphabet fini, la précision est d'importance, par un codage appoprié des données et résultats du problème. On pourra donc parler indifféremment de langage ou de problème.

5.1 Machines de Turing

Conformément à la figure 5.1, les machines de Turing considérées sont dotées :

1. d'une bande de lecture bi-infinie, sur laquelle la donnée, mot sur le vocabulaire V_t, sera écrite entourée à l'infini de caractères blancs, considéré comme notre caractère spécial noté _ n'appartenant pas à V_t ;
2. d'un nombre fini de bandes de travail qui peuvent stocker des mots entourés à l'infini de blancs sur les vocabulaires respectifs $V_1, \ldots, V_n$ ne contenant pas le caractère blanc ;
3. d'un contrôle matérialisé par un ensemble fini Q d'états ;
4. d'un état initial q_0 ;
5. d'un ensemble $F \subseteq Q$ d'états acceptants ;
6. et d'une fonction de transition T, qui est une application de $Q \times V_t \times V_1 \times \ldots \times V_n$ dans
   - $Q \times \{L, R\} \times \{L, R\} \times \ldots \times \{L, R\}$ si la machine est déterministe,
   - et $\mathcal{P}(Q \times \{L, R\} \times \{L, R\} \times \ldots \times \{L, R\})$ si la machine est non déterministe.

Lors de chaque transition, la machine de Turing lit un mot sur chaque bande, éffectue une transition d'état accompagnée du déplacement du lecteur de chaque bande conformément à la fonction T. La notion de reconnaissance est ici un peu différente de celle que l'on a vue pour les automates : il suffit d'accéder à un état acceptant sans qu'il soit nécessaire d'avoir lu le mot à reconnaitre en entier. La reconnaissance d'un mot arbitraire peut donc s'éffectuer en zéro transition, il suffit de déclarer que q_0 est acceptant. Cette machine, bien sûr, ne vérifie même pas que le mot est formé de lettres appartenant à l'alphabet autorisé pour la bande entrée.

On peut bien sûr prendre une autre définition où l'on force la lecture des données, mais ce modèle présente quelques inconvénients du point de vue de la théorie de la complexité, en particulier l'absence de complexité sub-linéaire.

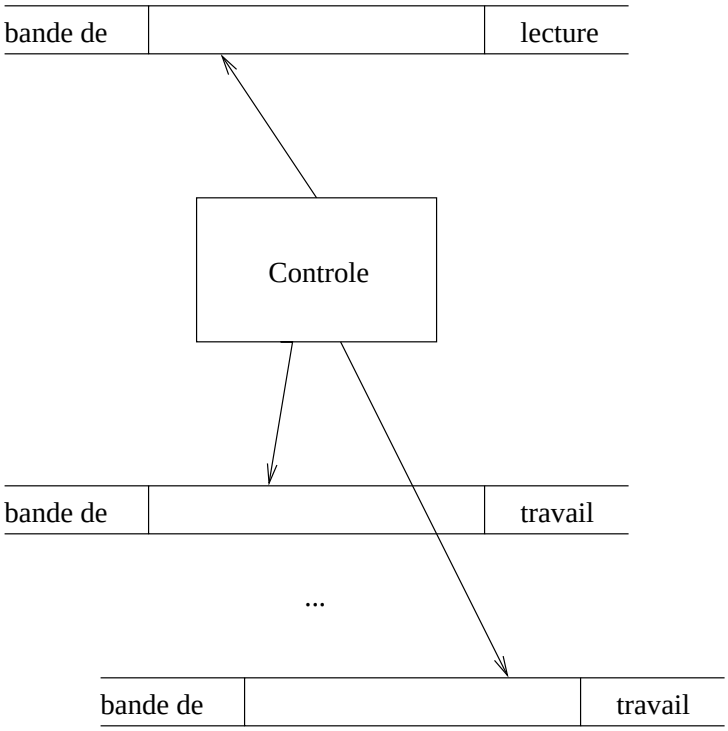


FIG. 5.1 – La machine de Turing.

5.2 Classes de complexité

On dira qu'une machine de Turing M est bornée en temps par la fonction $T(n)$ si elle ne fait pas plus de $T(n)$ transitions pour une donnée de taille n. On s'intéressera tout particulièrement au cas où T est un polynôme. Dans ce cas, on dira que M travaille en espace polynomial.

On dira qu'une machine de Turing M est bornée en espace par la fonction $S(n)$ si elle ne balaie pas plus de $S(n)$ cellules sur chaque bande pour une donnée de taille n. On s'intéressera tout particulièrement au cas où S est un polynôme. Dans ce cas, on dira que M travaille en temps polynomial.

Ces notions servent à classifier les langages suivant le temps -ou l'espace- nécéssaire pour reconnaitre les mots qui en font partie. On dira ainsi qu'un langage L appartient à la classe $DTIME(T(n))$ s'il existe une machine de Turing déterministe M qui reconnait un mot $u \in L$ en temps au plus $T(\mid u \mid)$, et à la classe $NTIME(T(n))$ s'il existe une machine de Turing non déterministe M qui reconnait un mot $u \in L$ en temps au plus $T(\mid u \mid)$. La définition des classes $DSPACE(S(n))$ et $NSPACE(S(n))$ est similaire.

Finallement, on dira qu'un langage L appartient à la classe P (respectivement NP) s'il existe un polynôme $P(n)$ tel que L soit reconnu en temps $DTIME(P(n)$ (respectivement, $NTIME(P(n))$). La définition des classes $PSPACE$ et $NPSPACE$ est similaire.

De nombreuses autre classes jouent un rôle important, en particulier les classes logarithmiques en espace, $LOGSPACE$ qui est déterministe, et $NLOGSPSACE$ qui est non déterministe -l'espace autorisé est une puissance quelconque du logarithme de la taille de l'entrée-, ainsi que les classes exponentielles en temps $EXPTIME$ et $NEXPTIME$. Enfin, la classe de complexité élémentaire (en temps) caractérise les langages dont la reconnaissance est bornée en temps par une hauteur arbitraire d'exponentielles emboitées, comme n^{n^n}, de hauteur 3. On ne parle dans ce cas que de complexité déterministe. Cette classes interviendra par la suite.

Le nombre de bandes utilisées n'a guère d'importance pour les classes de complexité

$P, NP, PSPACE, NPSPACE$: on peut le réduire à 2 en passant de $DTIME(T(n))$ à $DTIME(T(n)logT(n))$, et même à un en passant de $DTIME(T(n))$ à $DTIME(T(n)^2)$.

Le tableau qui suit récapitule les classes de complexité que nous venons de décrire.

	MT déterministe	MT non déterministe
Temps $T(n)$	$DTIME(T(n))$	$NTIME(T(n))$
Temps polynomial	P	NP
Temps exponentiel	EXP	$NEXP$
Temps élémentaire		
Temps non élémentaire		
Espace $S(n)$	$DSPACE(S(n))$	$NSPACE(S(n))$
Espace logarithmique	$LOGSPACE$	$NLOGSPACE$
Espace polynomial	$PSPACE$	$NPSPACE$

Une dernière classe importante est $CoNP$. On dit qu'un langage $L \subseteq V^*$ est dans $CoNP$ ssi son complémentaire $\overline{L}$ est dans NP. Cette définition est plus subtile qu'il n'y parait à première vue : pour reconnaitre $u \in L$ avec la machine non déterministe M reconnaissant $\overline{L}$, il est nécéssaire de rejeter u, c'est-à-dire d'essayer toutes les façons possibles de reconnaitre u avec M pour vérifier qu'elles échouent toutes. Peut-on reconnaitre u avec une machine de Turing non-déterministe M ? On n'en sait rien, le problème est ouvert.

5.3 Relations entre mesures de complexité

Tout d'abord, notons que

$$DTIME(T(n)) \subseteq DSPACE(T(n))$$

puisque il faut au moins n transition pour visiter n cases dds bandes mémoires.

Pour donner une "réciproque", il faut d'abord s'assurer que borner l'espace implique un temps borné. Cela revient à montrer que si la machine M travaille en espace borné, alors il existe une machine de Turing M' qui reconnait le même langage, et travaille en espace et en temps borné à la fois. Plus précisément, on peut montrer que

$$DSPACE(S(n)) \subseteq DTIME(c^{S(n)})$$

où la constante c dépend de la machine M. On montre une relation similaire entre temps déterministe et non déterministe :

$$NTIME(T(n)) \subseteq DTIME(c^{T(n)})$$

où, à nouveau, la constante c dépend de la machine M. Par contre, la relation entre espace déterministe et non déterministe est bien différente, c'est le théorème de Savitch :

$$NSPACE(S(n)) \subseteq DSPACE(S(n)^2)$$

ce qui implique que

$$NPSPACE = PSPACE$$

et

$$NLOGSPACE = LOGSPACE.$$

Les inégalités connues entre classes de complexité sont les suivantes :

$$DSPACE(log\ n) \subseteq P \subseteq \{NP,\ CoNP\} \subseteq PSPACE,$$

$$DSPACE(log\ n) \subset PSPACE,$$

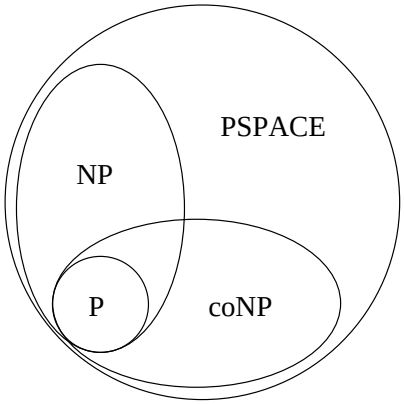


FIG. 5.2 – Inclusions (au sens large) essentielles.

$$LOGSPACE \subseteq PSPACE$$

et

$$P \neq LOGSPACE.$$

On voit que la pluspart de ces inégalités ne sont pas strictes ; par ailleurs, on ne sait pas si l'une des classes P et $LOGSPACE$ est incluse dans l'autre. La théorie de la complexité s'avère plus riche de questions que de réponses. La question de savoir si $P = NP$ ou pas et considérée comme l'une des grandes questions actuelles non seulement de l'informatique, mais de l'ensemble des mathématiques.

Les inclusions les plus importantes sont représentées à la figure 5.2. Bien entendu, d'après les inclsions de classes de complexité rappellées précédemment, toutes les classes de la figure sont incluses dans la classe exponentielle en temps.

5.4 Langages complets

De nombreux langages sont dans NP pour lesquels aucune solution polynomiale n'est connue. Le but de ce paragraphe et d'identifier parmi eux ceux qui sont difficiles, en ce sens que si l'un d'eux devait être polynomial, alors les classes P et NP coincideraient. La notion qui permet de comparer la difficulté de deux langages différents fait l'objet du paragraphe qui suit. Elle a une portée générale, qui dépasse le problème de comparer P et NP.

5.4.1 Réductions et complétude

Définition 5.1 *Le langage L' est réductible au langage L en temps polynomial, on dit aussi* polynomialement réductible en temps, *par la machine de Turing $M : L' \mapsto L$ si :*

1. *M fonctionne en temps polynomialement borné en fonction de la taille de son entrée ;*
2. *$M(x) \in L$ ssi $x \in L'$.*

Définition 5.2 *Le langage L' est réductible au langage L en espace logarithmique, on dit aussi* logarithmiquement réductible en espace, *par la machine de Turing $M : L' \mapsto L$ si :*

1. *M fonctionne en espace logarithmiquement borné en fonction de la taille de son entrée ;*
2. *$M(x) \in L$ ssi $x \in L'$.*

Lemme 5.3 *Supposons que L' soit polynomialement réductible en temps à L. Alors,*

1. $L \in NP \Longrightarrow L' \in NP$;
2. $L \in P \Longrightarrow L' \in P$.

Lemme 5.4 *Supposons que L' soit logarithmiquement réductible en espace à L. Alors,*

1. $L \in P \Longrightarrow L' \in P$;
2. $L \in LOGSPACE \Longrightarrow L' \in LOGSPACE$.

Ce sont les réductions qui vont nous permettre de comparer les langages entre eux :

Définition 5.5 *Soit $\mathcal{C}$ une classe de complexité. On dit que L est complet pour $\mathcal{C}$ vis-à-vis de la réduction polynomiale en temps (respectivement la réduction logarithmique en espace) si*

1. *$L \in \mathcal{C}$ ($\mathcal{C}$-difficulté) ;*
2. *tout langage $L' \in \mathcal{C}$ est polynomialement réductible en temps (respectivement, logarithmiquement réductible en espace) à L*

5.4.2 NP-complétude

On connait un grand nombre de problèmes $NP - complets$, c'est le cas de la pluspart de problèmes de nature combinatoire. On connait même des problèmes qui sont dans NP mais dont on sait pas s'ils sont dans P, ni s'ils sont complets. C'est le cas du test de primalité d'un entier.

Le problème paradigmatique de la classe NP est SAT : il s'agit de savoir si une formule propositionnelle donnée est satisfiable ou pas.

On considère le langage des formules logiques propositionnelles, c'est-à-dire baties à l'aide d'un ensemble donné $\mathcal{P}rop$ de symboles (ou variables) propositionnels pouvant pendre les deux valeurs $\{0,1\}$ et des connecteurs logiques $true, false, \wedge, \vee, \neg$. Une formule propositionnelle est dite satisfiable s'il existe une application de $\mathcal{P}rop$ dans $\{0,1\}$ telle qu'elle s'évalue en 1, $true, false$ étant des constantes logiques s'évaluant respectivement en 0 et 1.

Le problème SAT consiste à déterminer l'existence d'une telle application pour une formule donnée, ϕ, arbitraire.

On peut représenter le problème SAT comme le langage L_{SAT} construit comme suit : s'il y a n variables, la ième sera représentée par le mot formé de la lettre x suivie du code binaire de i. Notre alphabet sera donc $\{\wedge, \vee, \neg, (,), x, 0, 1\}$, permettant de coder une formule de longueur n par un mot de longueur (l'arrondi supérieur de) $n\ log\ n$. En effet, il y a au plus $n/2$ variables différentes dans une formule de taille n, et donc le code de chacune d'entre elles ne nécéssite pas plus de (l'arrondi supérieur de) $1 + log\ n$ lettres du vocabulaire. Comme nous allons utiliser une réduction en espace logarithmique, on va pouvoir faire comme si une formule de taille n était codée par un mot de même taille n, car $log(n\ log\ n) \leq log\ n^2 = 2\ log\ n$, et que toutes ls quantitées sont à une constante multiplicative près.

Lemme 5.6 *SAT est dans NP.*

Preuve: Pour cela, il suffit de construire de manière non-déterministe des valeurs de vérité pour toutes les variables, puis d'évaluer la formule, ce qui se fait en temps linéaire en la taille de la formule. □

D'une manière générale, les preuves d'appartenance à NP se décomposent en deux phases :

1. la première consiste à construire un arbre de choix de hauteur polynomiale, qui figure l'ensemble de tous les cas devant être examinés, en temps (non-déterministe) proportionnel à sa hauteur en utilisant le non-déterminisme des machine de Turing. Dans notre exemple, il s'agit de construire toutes les possibilités d'affecter des valeurs de vérité pour les variables ;

2. pour chaque choix, un calcul polynomial avec une machine déterministe. Lors de cette seconde phase, il s'agit de *vérifier* une propriété ; dans notre exemple, il s'agit de vérifier que les valeurs de vérité affectées aux variables rendent la formule vraie. C'est là une caractéristique essentielle des problèmes NP.

Il suffit pour terminer de collecter les résultats aux feuilles afin de vérifier s'il existe un choix d'affectation de valeurs de vérité aux variables qui rende la formule vraie, ce qui est fait par la définition de l'acceptation pour les machines non-déterministes.

Théorème 5.7 *SAT est NP-complet.*

Preuve: Il nous reste à montrer que tout problème de la classe NP est réductible à L_{SAT} en espace logarithmique en la taille de la donnée. Pour cela, étant donnée une machine de Turing non déterministe M -dont nous supposerons qu'elle possède une bande unique infinie à droite seulement pour plus de facilité- qui reconnait son entrée $x = x_1 \dots x_n$ de taille n en temps borné par un polynôme $p(n)$, nous allons définir un algorithme qui travaille en espace logarithmique pour construire une formule propositionnelle M_x qui est satisfiable ssi M accepte x. Cet algorithme pourra être facilement décrit sous la forme d'une machine de Turing prenant en entrée une description du calcul de M pour sa donnée x, et écrivant la formule M_x sur sa bande de sortie, et utilisant par ailleurs des bandes de travail pour éffectuer ses calculs.

Soit $\#m_0\#m_1 \dots \#m_{p(n)}$ la description d'un calcul de M formé de $p(n)$ transitions. Si la machine M accepte avant la $p(n)$ème transition, on répètera une transition vide sur l'état acceptant -ce qui ne change pas le langage reconnu de M ni le polynôme $p(n)$- de manière à ce qu'il y ait exactement $p(n)$ transitions et donc $(p(n)+1)^2$ caractères au total. Les $p(n)+1$ mots m_i de longueur $p(n)$ décrivent la configuration de la machine M après exactement i transitions. Ils sont de la forme $\alpha_i(q_i, a_i, b_i, D_i, q_{i+1})\beta_i$, où

- q_i désigne l'état courant, α_i est le mot écrit sur la bande depuis l'extrémité gauche jusqu'à la tête de lecture non comprise,
- a_i est le caractère lu,
- b_i est le caractère écrit lors de la transition à venir (choisi arbitrairement pour $i = p(n)$),
- D_i est la direction de déplacement de la tête de lecture lors de la transition à venir (choisie arbitrairement pour $i = p(n)$),
- q_{i+1} désigne l'état atteint dans la transition à venir (choisi arbitrairement pour $i = p(n)$),
- β_i est le mot écrit sur la bande à partir de la tête de lecture non-comprise jusqu'au premier blanc rencontré, et complété si nécéssaire par des blancs de manière à ce que le mot $\alpha_i a_i \beta_i$ ait exactement $p(n)$ caractères.

Chaque mot $\#\alpha(q, a, b, D, q')\beta$ est appellé une *description instantanée*, où les quintuplets (q, a, b, D, q') sont considérés comme les lettres d'un alphabet fini particulier W dont le sous-ensemble formé des quintuplets $(f \in F, a, b, D)$ est noté W_F. On utilisera la notation pointée de la programmation par objets pour récupérer les composantes des éléments de W. Notons par ailleurs que β peut être une séquence (éventuellement vide) formée de blancs uniquement si la tête de lecture pointe sur un blanc.

Á chaque caractère X qui peut figurer dans la descrition d'un calcul, on associe une variable propositionnelle $c_{i,j,X}$ qui servira à indiquer si le $(j+1)$ème caractère (on commence avec $j = 0$) de la ième description instantanée est un X. L'expression M_x cherchée doit assurer les propriétés suivantes :

1. Mot : les variables $c_{i,j,X}$ décrivent un mot, et donc l'une d'elle exactement est vraie pour chaque paire (i, j) ;
2. Init : m_0 décrit l'état initial de la machine M avec i sur la bande ;
3. Final : $m_{p(n)}$ possède un état acceptant ;
4. Trans : pour chaque $i \in [0, p(n)]$, m_{i+1} découle de m_i par une transition de la machine non déterministe M.

La formule M_x cherchée est donc la conjonction des formules décrivant ces quatre propriétés. Décrivons les dans notre langage logique :

$$\text{Mot} = \bigwedge_i \bigwedge_j \left(\bigvee_X c_{i,j,X} \wedge \neg \left(\bigvee_{X \neq Y} (c_{i,j,X} \wedge c_{i,j,Y}) \right) \right)$$

$$\text{Init} = c_{0,0,\#} \wedge \bigvee_{X \in W} c(0,1,X) \wedge \bigwedge_{j \in [1,n]} c_{0,j,x_j} \wedge \bigwedge_{j \in [n+1,p(n)]} c_{0,j,\_}$$

$$\text{Final} = \bigwedge_j \bigwedge_{X \in W_F} c_{p(n),j,X}$$

$$\text{Trans} = \bigwedge_{i \in [0,p(n)-1]} \bigwedge_{j \in [0,p(n)]} \bigvee_{\substack{W,X,Y,Z \text{ such} \\ \text{that} f(W,X,Y,Z)}} c_{i,j-2,W} \wedge c_{i,j-1,X} \wedge c_{i,j,Y} \wedge c_{i+1,j,Z}$$

où le prédicat de filtrage $f(W,X,Y,Z)$ décrit la possibilité pour le caractère Z d'apparaitre en position j de la $(i+1)$ème description instantanée, sachant que les caractères W, X et Y apparaissent aux positions respectives j_1, j et $j+1$ de la ième description instantanée. Il est important de noter que ce prédicat doit simplement être calculé à partir de la donnée de la machine M, sans qu'il soit nécessaire de le construire. Il sert en effet à éliminer les calculs impossibles de la formule. Il faudrait au contraire le construire dans notre langage si nous avions défini

$$\text{Trans} = \bigwedge_{i \in [0,p(n)-1]} \bigwedge_{j \in [0,p(n)]} \bigvee_{W,X,Y,Z} f(W,X,Y,Z) \Longrightarrow c_{i,j-2,W} \wedge c_{i,j-1,X} \wedge c_{i,j,Y} \wedge c_{i+1,j,Z}$$

Il faut maintenant montrer qu'il est possible de construire la formule M_x, ce qui inclue le calcul du prédicat f, en espace logarithmique. Cela résulte du fait que la taille de M_x est de l'ordre de $p(n)^2$, et qu'il suffit d'un espace logarithmique pour compter jusqu'à une valeur qui s'exprime comme un polynôme de n.

Il reste enfin à montrer que la machine M accepte la donnée x si et seulement si la formule M_x est satisfiable. Comme tout a été fait pour que ce soit vrai, la preuve en est laisée au lecteur. □

Les problèmes NP-complets sont très nombreux, comme la programmation linéaire en nombres entiers, le problème du sac à dos, le problème du voyageur de commerce, ou comme la recherche dans un graphe d'un circuit hamiltonien, d'un chemin de longueur minimale, d'un coloriage en 4 couleurs. La recherche d'un chemin eulérien, par contre est polynomiale.

On peut se demander où est la frontière entre le polynomial et le polynomial non-déterministe. Appelons $n-CNF$, la restriction du problème SAT au cas où la formule propositionnelle est une conjonction de clauses comprenant toutes le même nombre n de littéraux. On peut montrer que $2-CNF$ est polynomial, alors que $3-CNF$ est déja NP-complet.

Les problèmes NP-complets ont longtemps été considérés comme intractable. Qu'un problème soit NP-complet n'empêche pas de le résoudre, en général, du moins ses instances de taille pas trop grande. Il existe souvent de nombreuses sous-classes qui sont polynomiales, comme $2-CNF$ dans le cas de SAT. La difficulté est généralement concentrée sur des problèmes très particuliers, qui font apparaître des phénomènes de seuils pour certains paramètres : loin du seuil, le problème est polynomial ; près du seuil, il devient difficile, mais il peut alors être possible de trouver une solution approximative en utilisant des algorithmes probabilistes.

5.4.3 PSPACE-complétude

La classe $PSPACE$ est tout aussi mystérieuse. Le problème $PSPACE$-complet paradigmatique de cette classe est un autre problème de logique, QBF pour Quantified Boolean Formulae, pour

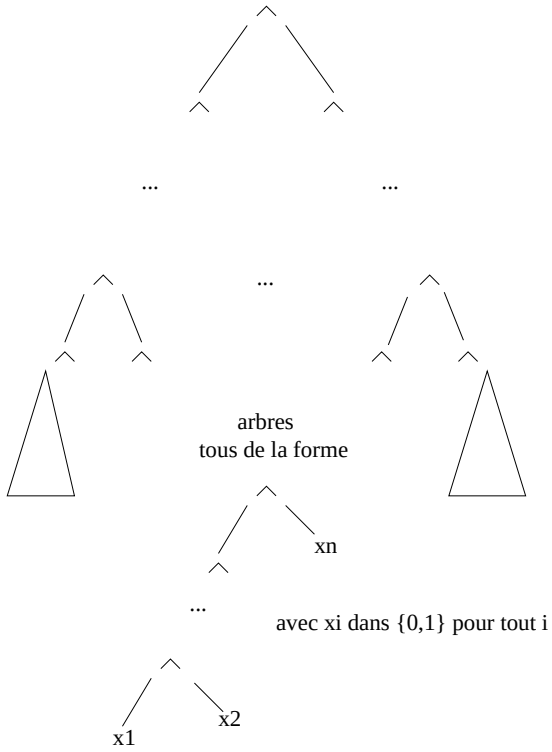


FIG. 5.3 – La formule transformée de taille exponentielle.

lequel il s'agit de décider si une formule propositionnelle quantifiée est satisfiable ou pas. Le langage est donc cette fois bâti à partir d'un nombre fini de variables propositionnelles, des connecteurs logiques habituels, et des quantificateurs universel et existentiel. C'est l'alternance des quantificateurs qui va nous propulser dans une classe dont on pense qu'elle et intrinsèquemnt plus complexe que la classe NP. On dit qu'une variable est libre dans une formule si elle n'est pas dans la portée d'un quantificateur.

Lemme 5.8 *QBF est dans NP.*

Preuve: La preuve utilise l'élimination des quantificateurs sur un ensemble fini, ici l'ensemble des valeurs de vérité. On a en effet les équivalences logiques :

$$\forall x \phi(x) \equiv \phi(0) \wedge \phi(1)$$

$$\exists x \phi(x) \equiv \phi(0) \vee \phi(1)$$

Le problème est qu'étant donnée une formule de taille n, la formule transformée peut être de taille exponentielle en n :

$$\forall^n \; x_1 x_2 \ldots x_n \text{ est de taille } 4n-1$$

mais sa transformée représentée à la figure 5.3 est de taille $2^{n-1} - 1 + 2^n \times (n-1) \; > \; (n-1)^2 \; 2^n$.

On va donc parcourir l'arbre ci-dessus sans l'engendrer, en définissant une procédure d'évaluation récursive $EVAL$ qui utilise un espace de tavail polynomial dans la pile d'exécution pour évaluer ϕ :

1. $\phi = true$: EVAL retourne la valeur $true$;

2. $\phi = \phi_1 \wedge \phi_2$: EVAL retourne la conjonction des résultats obtenus récursivement pour ϕ_1 et ϕ_2 ;
3. $\phi = \phi_1 \vee \phi_2$: EVAL retourne la disjonction des résultats obtenus récursivement pour ϕ_1 et ϕ_2 ;
4. $\phi = \neg\psi$: EVAL retourne la négation du résultat obtenu récursivement pour ψ ;
5. $\phi = \exists x\psi$: EVAL construit les formules ψ_0 et ψ_1 en remplaçant respectivement la variable x par 0 et 1 dans ψ, puis retourne la disjonction des résultats obtenus récursivement pour ψ_0 et ψ_1 ;
6. $\phi = \forall x\psi$: EVAL construit les formules ψ_0 et ψ_1 comme précédemment, puis retourne la conjonction des résultats obtenus récursivement pour ψ_0 et ψ_1.

Comme le nombre de connecteurs ou quantificateurs est au plus égal à la taille n de la formule, la profondeur de récursion est au plus n. La taille des données à stocker dans la pile d'exécution étant linéaire en n, on en déduit que l'espace occuppé dans la pile (pour laquelle on utilisera par exemple une bande spécifique de la machine) est quadratique. Donc QBF est dans $PSPACE$. □

Les preuves d'appartenance à $PSAPCE$ patissent du fait que $NPSPACE = PSPACE$. On pourrait en déduire que le même schéma va à nouveau fonctionner, en construisant un arbre de choix en espace polynomial non-déterministe. En fait, cela n'est pas le cas. Les problèmes $PSPACE$, comme le problème QBF, ne se prètent pas à la décomposition en une première phase d'énumération non-déterministe qui contient toute la complexité suivie d'une phase de vérification polynomiale. Vérifier qu'un ensemble de valeurs de vérité est une solution d'un problème de QBF est tout ausi difficile que de les trouver.

Théorème 5.9 *QBF est PSPACE-complet.*

Preuve: Reste à réduire tout problème de $PSPACE$ à QBF par une réduction que l'on choisit polynomiale en temps. La réduction est similaire à la réduction précédente pour NP, en plus complexe. □

Nous terminons par un problème qui intervient de manière cruciale en vérification : l'accessibilité dans un graphe consiste, étant donnés un graphe G et deux sommets a et b, à déterminer s'il existe un chemin allant de a à b.

Théorème 5.10 *L'accessibilité dans un graphe est $NSPACE(log\ n)$-complet pour les réductions en espace logarithmique.*

Preuve: Le codage du problème comme un langage sur un alphabet fini est laissé au lecteur. Notons toutefois que le codage d'un noeud du graphe prendra $log\ n$ bits s'il y a n sommets.

On montre tout d'abord que le problème est dans la classe $NSPACE(log\ n)$. Pour cela, on va engendrer tous les chemins grâce au non-déterminisme, de sorte que la mémoire occuppée soit de taille logarithmique en n pour chacun d'eux. Pour cela, on va à chaque étape (il y aura au plus n étapes) engendrer tous les sommets et vérifier que l'on construit bien un chemin d'une part, et si le sommet engendré est le sommet voulu d'autre part. Cela suppose de mémoriser pour chaque chemin partiel le sommet qui vient d'être engendré plus le sommet précédent, ce qui se fait en espace logarithmique. On ne conserve donc pas le chemin.

Il reste à réduire tout problème de $NSPACE(log\ n)$ au problème d'accessibilité par une machine de Turing déterministe fonctionnant en espace logarithmique. La réduction est esquissée, les détails sont laissés au lecteur.

Soit M une machine de Türing non-déterministe fonctionnant en espace logarithmique de la taille de son entrée, et x une donnée de taille n. La description de la machine à un moment donné se fait avec $log\ n$ bits, puisqu'elle fonctionne en espace logarithmique. Nous allons contruire un graphe M_x qui va coder le fonctionnement de la machine et aura un chemin du premier noeud au dernier ssi

la machine M reconnait x. Les noeuds du graphe seront les descriptions instantanées de la machine plus un dernier pour l'acceptance. Le premier noeud est la description initiale. il y aura un arc d'un noeud à un autre si la transition correspondante est posible. Ce graphe répond bien à la question, et il faut montrer qu'on peut l'engendrer avec une machine qui travaille en espace logarithmique. Cela est du au fait que chaque description instantanée est codée sur approximativement $log\ n$ bits. □

Cette preuve est intéressante car elle mêle la génération aléatoire incrémentale d'un objet (ici, un chemin dans un graphe) avec le test que cette génération construit bien l'objet en question. Cela est nécéssité par le besoin de réduire la taille occuppée à l'incrément de l'objet, l'objet lui-même prenant trop de place.

5.5 Exercices

Exercice 5.1 *Montrer que le problème de satisfiabilité pour CNF est $NP-complet$.*

Exercice 5.2 *Montrer que le problème de satisfiabilité pour $3-CNF$ est $NP-complet$.*

Pour montrer la complétude, on réduira le problème SAT au problème $3-CNF$. On pourra dans un premier temps se débarasser des négations internes en les faisant descendre aux feuilles en utilisant pour ce faire un espace de travail logarithmique. Dans un second temps, il faudra créer la formule $3-CNF$ à partir de la formule obtenue à l'issue de la première passe. La seconde passe consiste à engendrer les disjonctions de la clause les unes après les autres en montrant qu'il suffit de n clauses et d'un espace logarithmique pour les construire une par une.

Exercice 5.3 *Montrer que le problème de satisfiabilité pour $2-CNF$ est dans P. Est-il complet pour P ?*

Exercice 5.4 *On appelle* couverture *d'un graphe G un sous ensemble A de ses sommets tel que pour toute arète a, b, $a \in A$ ou $b \in A$.*

Montrer que le problème, étant donné un graphe G et un nombre k, de déterminer si G a une couverture de taille k est NP-complet.

Exercice 5.5 *Montrer que le problème du coloriage d'un graphe avec un nombre de couleurs minimal est NP-complet.*

Exercice 5.6 *On appelle* circuit *hamiltonien d'un graphe G un circuit passant une fois et une seule par tout sommet de G.*

Montrer que le problème, étant donné un graphe G, de déterminer s'il possède un circuit hamiltonien est NP-complet.

Exercice 5.7 *On appelle* circuit *eulerien d'un graphe G un circuit passant une fois et une seule par tout arc de G.*

Montrer que le problème, étant donné un graphe G, de déterminer s'il possède un circuit eulerien est dans P.

Exercice 5.8 *Montrer que le problème, étant donné un entier n, de déterminer s'il est premier est dans NP. En fait, on sait depuis l'été 2002 que ce problème est polynomial (de degré élevé, ce qui fait que l'algorithme polynomial connu n'est pas compétitif avec les algorithmes exponentiels antérieurs pour des nombres de taille raisonnable, tels que ceux utilisés en cryptographie).*

Chapitre 6

Automates de mots infinis

Nous allons maintenant nous intéresser aux automates de mots infinis, automates de Büchi pour commencer, automates de Büchi alternants plus tard. Ces automates ne diffèrent des automates de mots finis que par la condition d'acceptation.

6.1 Mots infinis et automates de Büchi

Définition 6.1 *Un* mot *sur le vocabulaire V_t est une application w de $\underline{n}$ dans V_t, où $\underline{n}$ désigne un ordinal au plus égal à ω, le premier ordinal limite. $\mid w \mid = \underline{n}$ est la longueur du mot. Le mot w est fini si $\underline{n} < \omega$ et infini sinon . On note par V_t^* l'ensemble des mots finis, par V_t^ω l'ensemble des mots infinis, et par V_t^∞ l'ensemble des mots finis ou infinis.*

Définition 6.2 *Étant donné un automate non-déterministe $\mathcal{A} = (V_t, Q, T)$, on appelle* calcul *de longueur $\underline{n} \leq \omega$ issu d'un état $q_0 \in Q$ toute application u de $\underline{n}$ dans V_t telle que*

(i) $u(0) = q_0$;

(ii) $\forall i < \underline{n}\ u(i) = q_{i+1} \Longrightarrow q_{i+1} \in T(q_i, a_i)$ for some $a_i \in V_t$.

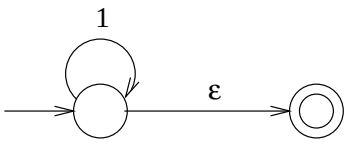
On notera

- *$q_0 \xrightarrow{w}{}^* q_n$ pour un calcul fini de longueur n, auquel cas q_n est l'*état atteint *en ayant lu le mot $w = a_1 \cdots a_n$;*
- *$q_0 \xrightarrow{w}{}^\omega$ pour un calcul infini (de longueur ω), auquel cas $\{q \mid \{i \mid e(i) = q\}$ est de cardinal infini$\}$ est l'*ensemble des états atteints *en ayant lu le mot infini $w = a_1 \cdots a_\omega$.*

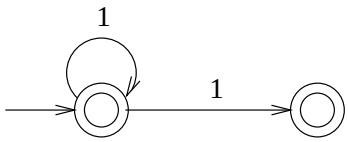
Dans les deux cas, on appelle limite *du calcul e, notée $lim(e)$, l'ensemble non vide des états atteints dans le calcul, et on dit que l'automate $\mathcal{A} = (V_t, Q, q_0, F, T)$ reconnait le mot fini ou infini w s'il existe un calcul e issu de l'état initial q_0 qui lit le mot w et tel que $lim(e) \cap F \neq \emptyset$. On notera par $L(\mathcal{A})$ le langage des mots finis reconnus par l'automate $\mathcal{A}$ et par $L^\omega(\mathcal{A})$ le langage des mots infinis reconnus par l'automate $\mathcal{A}$ et l'on dira que $L^\omega(\mathcal{A})$ est* Büchi-reconnaissable. *On notera par $L^\infty(\mathcal{A})$ le langage $L^\omega(\mathcal{A}) \cup L(\mathcal{A})$ des mots finis ou infinis.*

Notons que notre définition fait que l'on ne peut pas reconnaitre un langage composé uniquement de mots infinis, puisque tout préfixe fini d'un mot infini reconnu sera reconnu pourvu que son calcul se termine en état acceptant. La définition de la reconnaissance pour les mots infinis assure que l'ensemble de ces préfixes est infini. A contrario, il se peut que l'automate ne reconnaisse aucun mot infini alors qu'il reconnait des mots finis. Par la suite, nous ne nous préoccuperons que des mots infinis (dans $L^\omega(\mathcal{A})$) reconnus par un automate fini grâce à la condition d'acceptation de Büchi.

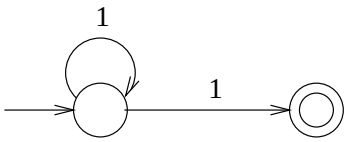
On peut bien sûr définir des automates de Büchi avec transitions vides, la définition précise en est laissée au lecteur. Notons que l'automate pourra éventuellement faire une infinité de transitions vides. L'élimination des transitions vides dans le cas des automates de Büchi est donc d'autant plus



Automate avec transitions vides



Automate équivalent sur les mots finis



Automate équivalent sur les mots infinis

FIG. 6.1 – Élimination des transitions vides d'un automate de Büchi non-déterministe.

importante. Notons aussi, cela est important, que l'on peut toujours compléter un automate de Büchi sans changer le langage reconnu de façon à ce qu'il ne s'arrête pas avant d'avoir entièrement lu sa donnée. La transformation est la même que pour les automates de mots finis. On fera par la suite systématiquement l'hypothèse que les automates de Büchi considérés sont complets.

6.2 Déterminisation

On commence par les automates avec transitions vides :

Théorème 6.3 *Pour tout automate de Büchi $\mathcal{A}$ avec transitions vides, il existe un automate de Büchi $\mathcal{A}'$ sans transitions vides qui reconnait le même langage de mots infinis.*

Preuve: On élimine les transitions vides en temps quadratique, après avoir résolu une petite difficulté : la transformation faite dans le cas des automates de mots finis ne marche pas : un état non acceptant q qui permet d'accéder à un état acceptant f par des transitions vides sans que f soit pour autant situé sur un circuit ne peut devenir acceptant sans changer le langage de mots infinis reconnus par l'automate. L'examen de la preuve faite dans le cas fini montre qu'il suffit de conserver le même ensemble d'états acceptants que l'automate $\mathcal{A}$. □ □

Un exemple d'automate qui illustre ces difficultés est donné en haut de la figure 6.1. L'automate de gauche reconnait les mêmes mots finis, mais pas les même mots infinis. Celui de droite reconnait les mêmes mots infinis, mais pas les même mots finis. Un autre exemple d'éliminations de transition vides est montré à la figure 6.4.

La parallèle avec les automates de mots finis s'arrête là :

Théorème 6.4 *Les automates de Büchi non-déterministes reconnaissent strictement plus de langages que les automates de Büchi déterministes.*

Preuve: Soit $L = \{(0+1)^* 1^\omega\}$.

Un automate de Büchi non-déterministe reconnaissant L est donné à la figure 6.2.

Supposons maintenant que l'on puisse reconnaitre ce même langage L par un automate de Büchi déterministe dont Q est l' ensemble des états, et analysons un calcul reconnaissant le mot $m_0 = 1^\omega$.

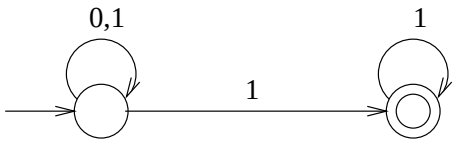


FIG. 6.2 – Automate de Büchi non-déterministe reconnaissant $(0+1)^*1^\omega$.

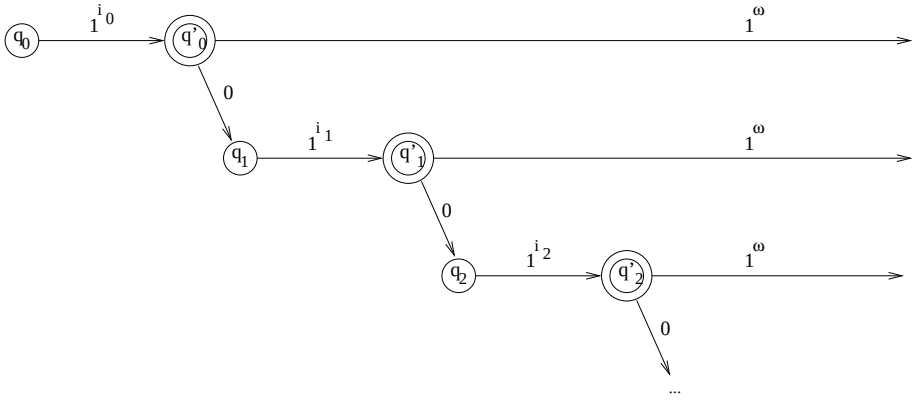


FIG. 6.3 – Calculs de l'automate de Büchi déterministe supposé reconnaitre $(0+1)^*1^\omega$.

Ce calcul passe forcément une infinité de fois par un état de F. Notons q'_0 le premier passage par un état de F dans ce calcul, après avoir reconnu le mot 1^{i_0}.

Analysons maintenant le mot $m_1 = 1^{i_0}01^\omega$. Comme l'automate est déterministe, le calcul reconnaissant m_1 va coincider avec le précédent jusqu'à atteindre l'état q'_0, puis il va prendre la transition sortante étiquettée par 0 qui existe puisque le mot $1^{i_0}01^\omega$ appartient au langage de l'automate, et se retrouver en l'état q_1 à partir duquel il devra à nouveau reconnaitre le mot 1^ω. Soit q'_1 le premier état acceptant rencontré à partir de q_1 dans ce calcul.

On peut recommencer ce raisonnement autant de fois qu'il y a d'états acceptants plus un, conformément à la figure 6.3. Si l'on note de manière générale q'_j le premier état acceptant rencontré à partir de q_j dans le calcul reconnaissant le mot $1^{i_0}01^{i_1}0\ldots1^{i_j}01^\omega$, qui passe nécessairement par l'état q'_j grâce au déterminisme, on en déduit que pour $j \geq \mid Q \mid$ la suite $q'_0, \ldots, q'_j$ comporte deux occurences au moins d'un même état acceptant, et donc l'automate comporte une boucle sur cet état acceptant, d'où l'on conclue la reconnaissance d'un mot infini de la forme $1^{i_0}01^{i_1}0\ldots1^{i_k}(01^{i_{k+1}}0\ldots1^{i_{k+l}})^\omega$ donc comportant une infinité de 0 et n'appartenant ipso facto pas à L □

6.3 Décision du vide

La technique est similaire à celle utilisée pour les automates finis. Toutefois, la notion d'état productif diffère quelque peu :

Définition 6.5 *Étant donné un automate (déterministe ou pas) $A = (V_t, Q, q_0, F, T)$, on dira que l'état $q \in Q$ est*

- accessible, *s'il existe un mot w tel que $q_0 \xrightarrow{w}{}^* q$;*
- productif, *s'il existe un mot w tel que $q \xrightarrow{w}{}^* f \xrightarrow{w\neq\varepsilon}{}^+ f \in F$;*
- utile, *s'il est à la fois accessible et productif.*

Un automate est dit réduit *si tous ses états sont utiles.*

La différence est donc que les états acceptants doivent être situés sur un cycle pour être eux-même productifs.

Lemme 6.6 *Le langage d'un automate de Büchi $\mathcal{A}$ est non vide si et seulement si l'un au moins des états acceptants accessibles est productif.*

Théorème 6.7 *Le vide du langage reconnu par un automate de Büchi est décidable en temps linéaire et est $NSPACE(log\ n)$-complet.*

Preuve: Le langage de mots infinis reconnu par un automate $\mathcal{A} = (V_t, Q, q_0, F, T)$ est non vide ssi il existe une exécution de la forme $q_0 \xrightarrow{v}{}^* q_f \xrightarrow{w}{}^* q_f \xrightarrow{w}{}^* q_f \ldots$, qui exprime le fait que q_f est visité infiniment souvent. Donc q_f est accessible depuis q_0 et depuis q_f. Un tel chemin peut être aisément trouvé en espace non-déterministe logarithmique : on devine grâce au non déterminisme un chemin de longueur au plus $2\times \mid Q \mid$ issu de q_0, ainsi qu'un état $f \in F$, et l'on teste que ce chemin est de la forme cherchée. Pour cela, on conserve en permanence l'état courant du chemin ainsi que l'état f. On compare cet état courant avec f, et l'on décide le non-vide dès que la double comparaison a réussi.

Un raisonnement un tout petit peu plus élaboré nous montre que le problème du vide et le problème de l'accessibilité dans un graphe sont en fait interréductibles, et donc le problème du vide est $NSPACE(log\ n)$-complet. □

La finitude diffère du cas standard elle-aussi. Le lecteur est invité à y réfléchir.

Un autre problème important lié au premier est celui du plein d'un automate :

Théorème 6.8 *Le plein d'un automate de Büchi est décidable en temps*

- *linéaire et est $NSPACE(log\ n)$-complet pour les automates déterministes ;*
- *exponentiel et est $PSPACE$-complet pour les automates non-déterministes.*

Preuve: Le cas des automates déterministes ne pose pas de problème, puisque le problème du plein d'un automate déterministe est équivalent au problème du vide de son complémentaire, et que ce dernier s'obtient par une transformation en espace logarithmique qui fait l'objet de la preuve du théorème 6.9.

Pour les non-déterministes, la preuve d'appartenance à PSPACE est laissée au lecteur.

Reste à montrer la PSPACE-complétude dans ce cas, par réduction de QBF au problème du plein. □

6.4 Propriétés de clôture

Ces propriétés sont fondamentales pour les applications au problème de décision de la satisfiabilité des expressions logiques de CTL* . En effet, le principe sera de montrer par récurrence sur la structure des formules que les solutions (vues comme des mots finis ou infinis) sont reconnaissables par des automates finis. Ainsi, la négation correspondra à la complémentation, la quantification universelle à une intersection, et l'existentielle à une union.

Théorème 6.9 *La classe des langages reconnaissables par des automates de Büchi est close par union, intersection et complémentation.*

Preuve:

- Union : on fait la même construction que pour les automates finis. Étant donnés deux automates finis complets sur le même alphabet V_t, $\mathcal{A}_1 = (V_t, Q_1, q_0^1, F_1, T_1)$ et $\mathcal{A}_2 = (V_t, Q_2, q_0^2, F_2, T_2)$ reconnaissant respectivement les langages $L_\omega(\mathcal{A}_1)$ et $L_\omega(\mathcal{A}_2)$, on construit l'automate

$$\mathcal{A}_{1\cup 2} = (V_t, Q_1 \times Q_2, (q_0^1, q_0^2), F_1 \times Q_2 \cup Q_1 \times F_2, T_{1\times 2})$$

où

$$T_{1\times 2}((q_1,q_2),a) = \{(q_1',q_2') \mid q_1' \in T_1(q_1,a), q_2' \in T_2(q_2,a)\}.$$

Reste à montrer que $\mathcal{A}_{1\cup 2}$ reconnait le bon langage de mots infinis. Soit w un mot infini sur V_t. $w \in L^\omega(\mathcal{A}_1 \cup \mathcal{A}_2)$ si et seulement si il existe un calcul e lisant w tel que $lim(e) \cap (F_1 \times Q_2 \cup Q_1 \times F_2) \neq \emptyset$ si et seulement si la projection sur Q_1 de $lim(e)$ intersecte F_1 ou bien la projection sur Q_2 de $lim(e)$ intersecte F_2, ce qui est équivalent à $w \in L^\omega(\mathcal{A}_1) \cup L^\omega(\mathcal{A}_2)$.
Rappellons que la correction du raisonnement ci-dessus suppose que les automates $\mathcal{A}_1$ et $\mathcal{A}_2$ soient complets.

- Intersection : la construction classique, qui ne diffère de celle pour l'union que par la définition des états acceptants (qui devient $F \times F'$) ne convient pas. Cette définition aurait en effet pour résultat que les états acceptants de chaque automate seraient traversés infiniment souvent *en même temps*, la contrainte de simultanéité ainsi introduite aboutissant à reconnaître un sous-ensemble (peut-être strict) du langage intersection. Pour l'éliminer, nous allons mémoriser à l'aide d'une variable $tour$ prenant ses valeurs dans l'ensemble $\{1,2\}$, quel automate n'a pas encore visité un état acceptant depuis le dernier passage de l'autre automate par un état acceptant. Les états vont devenir des triplets de la forme $(q_1,q_2,tour)$, donnant la définition suivante pour l'automate intersection :

$$\mathcal{A}_{1\cap 2} = (V_t, Q_1 \times Q_2 \times \{1,2\}, (q_0^1, q_0^2, 1), F_1 \times Q_2 \times \{1,2\}, T_{1\cap 2})$$

où

$$T_{1\cap 2}((q_1,q_2,tour),a) = \{(q_1',q_2',f(tour,q_1,q_2)) \mid q_1' \in T(q_1,a), q_2' \in T(q_2,a)\}$$

$$\text{avec } f(1, q_1 \in F_1, q_2) = 2, f(2, q_1, q_2 \in F_2) = 1 \text{ et sinon } f(tour,q_1,q_2) = tour$$

Reste à montrer qu'il reconnait le bon langage de mots infinis. La preuve formelle est laissée à la charge du lecteur.

- Complémentation : la construction est simple pour les automates de Büchi déterministes, et marche encore pour le sous-ensemble des automates de Büchi non-déterministe pour lesquels tout mot reconnu n'a pas de calcul infini qui échoue (qui sont "presque" déterministes !). Un exemple est montré à la figure 6.4. Nous allons traiter ici le cas particulier uniquement, avant de revenir ultérieurement sur le cas général au paragraphe 6.5.
Soit $\mathcal{A} = (V_t, Q, q_0, F, T)$. On construit tout d'abord un automate $\mathcal{A}'$ sans état initial obtenu à partir de $\mathcal{A}$ en éliminant les sommets de F et les transitions qui leur sont associées :

$$\mathcal{A}' = (V_t, Q \setminus F, , Q \setminus F, T' = T_{|Q\setminus F})$$

puis on renomme les états de l'automate obtenu, q_i' remplaçant q_i, dont l'ensemble est maintenant appellé Q', de sorte que $Q' \cap Q = \emptyset$. On construit ensuite l'automate

$$\mathcal{A}'' = (V_t, Q \cup Q', q_0, Q', T'')$$

où
$$\begin{cases} \forall a \in V_t & \forall q \in Q \setminus F & T''(q,a) & = & T(q,a) \\ \forall a \in V_t & \forall q' \in Q' & T''(q',a) & = & T'(q',a) \\ \forall a \in V_t & \forall q \in F & T''(q,a) & = & T(q,a) \cup \{q_1' \in Q' \mid q_1 \in T(q,a) \cap (Q \setminus F)\} \end{cases}$$

Cet automate est non-déterministe : à un moment donné d'un calcul, il doit décider qu'il ne visitera plus jamais un sommet de F.
La preuve que l'automate $\mathcal{A}''$ reconnait le langage complémentaire de $L^\omega(\mathcal{A})$ est laissée au lecteur.

□

Notre preuve de clôture pour la complémentation ne passe pas au cas général, pour lequel nous devrons faire appel aux automates alternants abordés plus loin. En fait, la généralisation de cette preuve est possible, mais le complémentaire dans le cas général est un automate alternant.

Soient $L = (0+1)^*1^\omega$ et son comlémentaire $L' = (1^*0)^\omega$ les langages reconnus par les automates représentés à la figure 6.4.

Automates avec transitions vides reconnaissant les langages respectifs (0+1)*1 $^\omega$ et (1*0) $^\omega$

Automates sans transitions vides reconnaissant les langages respectifs (0+1)*1 $^\omega$ et (1*0) $^\omega$

FIG. 6.4 – Automates de Büchi non-déterministes reconnaissant $(0+1)^*1^\omega$ et $(1^*0)^\omega$.

6.5 Automates alternants

Le non-déterminisme correspond à un choix existenciel d'un calcul, alors que sa négation correspond à un choix universel. Afin de disposer d'un cadre unifié permettant de faciliter les preuves des propriétés de clôture, on va construire des automates qui alternent ces deux types de choix. Ces automates calculeront sur des mots finis ou infinis, et dans ce dernier cas on parlera d'automates de Büchi alternants.

Étant donné un ensemble X de variables propositionnelles, on désigne par $B(X)$ l'ensemble des formules Booléennes en forme normale disjonctive construit sur X avec les connecteurs logiques $\wedge$ et $\vee$. On assimilera une disjonction vide à la constante logique $false$ et la conjonction vide à la constante logique $true$. On utilisera la notation $B^+(X)$ pour signifier que la disjonction vide est interdite, de même que la conjonction vide.

Le choix de prendre des formes normales vise à une meilleure compréhension, mais il complique les preuves. Le lecteur pourra utilement récrire ce paragraphe en éliminant cette restriction.

Étant donnée $b = (x_1^1 \wedge \ldots \wedge x_{k_1}^1) \vee \ldots \vee (x_1^n \wedge \ldots \wedge x_{k_n}^n)$ une formule de $B^+(X)$ particulière, on dira que le *facteur* $(x_1^i \wedge \ldots \wedge x_{k_i}^i)$ appartient à b. On aura parfois besoin de substituer dans une conjonction $b = x_1 \wedge \ldots \wedge x_k$ une variable x_i par une conjonction $a = x'_1 \wedge \ldots \wedge x'_{k'}$, résultant en la nouvelle conjonction $x_1 \wedge \ldots \wedge x_{i-1} \wedge x'_1 \wedge \ldots \wedge x'_{k'} \wedge x_{i+1} \ldots \wedge x_k$ dont la forme réduite (par application de l'idempotence de la conjonction) est notée $b\{x_i \mapsto a\}$.

Pour les automates déterministes complets, on a $T(a,q) = q' \in B^+(Q)$;

Pour les automates non-déterministes complets, on a $T(a,q) = q_1 \vee \ldots \vee q_n \in B^+(Q)$, chaque q_i étant un facteur de $T(a,q)$;

Pour les automates alternants complets, on généralise en posant $T(a,q) \in B^+(Q)$, les facteurs pouvant maintenant être des conjonctions non triviales. Pour les automates incomplets, on posera $T(a,q) \in B(Q)$.

Définition 6.10 *Un* automate alternant $\mathcal{A}$ *est un quintuplet* (V_t, Q, q_0, F, T) *où*

1. V_t *est le* vocabulaire *de l'automate ;*
2. Q *est l'ensemble des* états *de l'automate ;*
3. $T : Q \times V_t \to B(Q)$*, est la* fonction de transition *de l'automate.*

L'automate sera dit complet si $T : Q \times V_t \to B^+(Q)$*. On notera comme précédemment* $q \xrightarrow{a} b(q)$ *pour* $b(q) \in T(q, a \in V_t)$*. Plus généralement, on définit la relation d'accessibilité entre conjonctions d'états comme suit :*

$$q_1 \wedge \ldots \wedge q_n \xrightarrow{\varepsilon} q_1 \wedge \ldots \wedge q_n$$
$$q_1 \wedge \ldots \wedge q_n \xrightarrow{a \cdot u} b' \quad \textit{si} \quad b_1 \wedge \ldots \wedge b_n \xrightarrow{u} b' \textit{ avec } \forall i\ b_i \in T(q_i, a)$$

L'idée sous-jacente aux automates alternants est que les disjonctions ont un comportement non-déterministe, il suffit de trouver une branche qui conduise au succès, alors que les conjonctions imposent (par négation) que toutes les branches correspondantes conduisent au succès. Bien entendu, dans le cas où $T(q,a) = false$, le calcul se bloque.

Un calcul d'un automate alternant est donc un arbre de chemins de calculs de la forme représentée à la figure 6.5, dans lequel, pour chaque noeud q_i et pour chaque lettre $a \in V_t$, il y a des successeurs $q_i^1, \ldots, q_i^n$ ssi $q_i^1 \wedge \ldots \wedge q_i^n$ est l'un des facteurs de la forme normale disjonctive $T(q_i, a)$. Un calcul est donc l'arbre issu de l'état initial associé à la fonction de transition définie plus haut.

Un calcul sera *réussi* si pour tout chemin de calcul c de l'arbre d'exécution, $lim(c) \cap F \neq \emptyset$.

Notons que toutes les transitions à profondeur k sont étiquettées par la même lettre a. De plus, si le mot reconnu lors du calcul est de longueur (finie ou infinie) $\underline{n}$, alors tous les chemins du calcul auront la même longueur $\underline{n}$.

Il a bien sûr une similarité entre la notion de calcul pour les automates alternants et celle d'arbre de calcul des automates non déterministes. Cette analogie est toutefois trompeuse, puisque dans le cas des autromates alternants, l'arbre est un arbre *conjonctif* (tous les chemins doivent conduire au succès), alors qu'il s'agit d'un arbre *disjonctif* dans le cas des automates non-déterministes (un

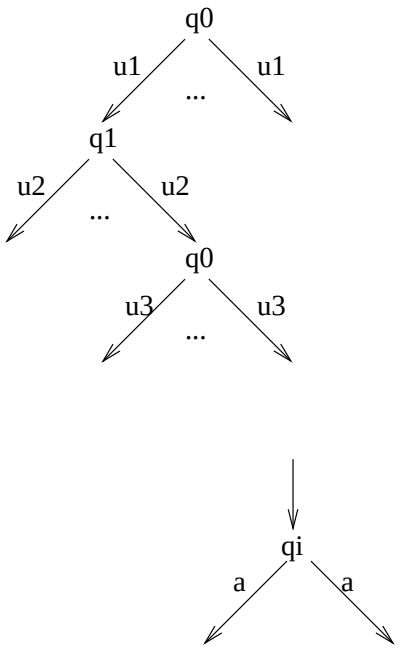


FIG. 6.5 – Calcul d'un automate alternant.

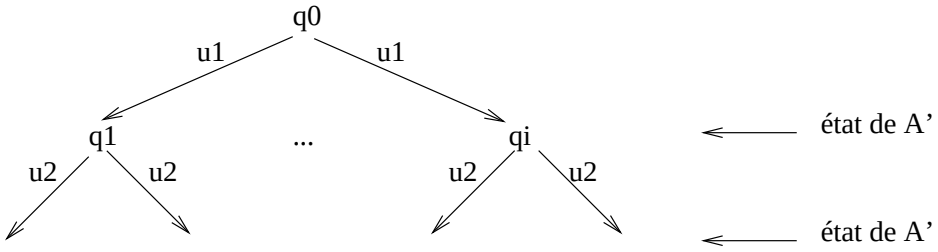


FIG. 6.6 – Calcul d'un automate de Büchi équivalent à un automate alternant.

chemin doit conduire au sucès). L'arbre des calculs d'un automate alternant serait donc en fait un arbre *disjonctif* d'arbres *conjonctifs* de chemins de calculs.

En fait, les automates alternants ont la même expressivité que les automates non-déterministes, que ce soit pour des mots finis ou infinis, il suffira de mener en même temps tous les chemins de calculs de l'arbre de calculs.

Théorème 6.11 *Soit $\mathcal{A}$ un automate alternant. Il existe des automates non-déterministes $\mathcal{A}'$ et $\mathcal{A}''$ tels que $L(\mathcal{A}') = L(\mathcal{A}'')$ et $L^\omega(\mathcal{A}') = L^\omega(\mathcal{A}'')$.*

Preuve:

Cas des mots finis. L'automate non-déterministe $\mathcal{A}'$ va deviner l'arbre de calcul du mot u et le construire grâce à une explosion exponentielle du nombre des états, conformément à la figure 6.6.

On définit donc

$$\mathcal{A}' = (V_t, \mathcal{P}(Q), \{q_0\}, \mathcal{P}(F), T')$$

où

$$T'(A \subseteq Q, a) = \bigcup_{q \in A} b_i \in T(q, a)$$

Cas des mots infinis. Il faut cette fois mémoriser quelles sont les branches qui n'ont pas rencontré un état de F récemment, un état final étant atteint lorsque toutes en auront rencontré un (au moins). On va pour cela partitionner les états de l'automate précédent en deux sous-ensembles, celui de gauche servant à calculer les transitions, et celui de droite à mémoriser ceux qui ont rencontré un état de F récemment.

$$\mathcal{A}'' = (V_t, \mathcal{P}(Q)^2, (\{q_0\}, \emptyset), \{\emptyset\} \times \mathcal{P}(F), T'')$$

où

$$\begin{aligned} T''((A \subseteq Q, B), a) &= \bigcup_{q \in A} C_i \in T(q, a) \\ T''((\emptyset, A), \varepsilon) &= (A, \emptyset) \end{aligned}$$

Cette ε-transition s'exécute chaque fois que l'on rencontre un état acceptant de $\mathcal{A}''$: elle a pour rôle de remettre le compteur à zéro.

La preuve que ces constructions remplissent leur objectif est laissée au lecteur. □

On peut montrer que cette explosion combinatoire est nécessaire. Pour cela, on construit des langages des mots finis ou infinis reconnaissables par des automates alternants et difficilement reconnaissables par des automates non-déterministes :

L_n : ensemble des mots pour lesquels il existe une paire de lettres distinctes à distance n l'une de l'autre.

$\overline{L_n}$ sera alors reconnu par un automate alternant ayant un nombre d'états linéaire en n, et un automate non-déterministe ayant un nombre d'états exponentiel en n. Pour montrer que l'on ne peut faire mieux, il suffit de montrer que l'automate déterministe minimal a un nombre d'états doublement exponentiel en n. Les détails sont à faire en exercice.

Les automates alternants sont clos par les opérations Booléennes. Cela est simple à prouver pour la conjonction et la disjonction, mais est plus délicat pour le complémentation.

Théorème 6.12 *Les automates alternants sont clos par union et intersection.*

La preuve est à faire en exercice. Il est un peu plus compliqué de montrer que le complémentaire du langage reconnu par un automate de Büchi est reconnu par un automate alternant. La construction est similaire à celle que nous avons faite pour les automates de Büchi. Mais le non-déterminisme va résulter en un automate alternant. En effet, dans le cas d'une transition non-déterministe, il faut vérifier qu'aucun des calculs n'est acceptant, et pour cela on va transformer l'automate en changeant le $\vee$ de la fonction de transition non-déterministe en un $\wedge$. De manière non-déterministe cette fois, on enverra un calcul se poursuivre dans une copie de l'automate de départ d'où la transition $\wedge$ choisie aura été divisée en deux sous-ensembles, et l'on recommence. On arrivera ainsi un jour à un automate déterministe, et on fera comme dans le cas déja traité.

Théorème 6.13 *Étant donné un automate alternant $\mathcal{A} = (V_t, Q, q_0, F, T)$, il existe un automate alternant $\overline{\mathcal{A}}$ qui reconnait le langage $\overline{L(\mathcal{A})}$.*

Preuve: La preuve généralise aux automates alternants celle que nous avons esquissée pour les automates de Büchi. □

Corollaire 6.14 *Les automates de Büchi sont clos par complémentaire.*

Il nous reste un dernier problème à examiner, celui du vide :

Théorème 6.15 *Le vide et le plein d'un automate alternant sont décidables en temps exponentiel et sont $PSPACE$-complets.*

Preuve: Pour la décision, on passe par le calcul de l'automate non-déterministe équivalent. Pour la $PSPACE$-complétude, on réduit successivement le plein des automates non-déterministes au vide des automates alternants, puis ce dernier au vide des automates de Büchi alternants. Pour cette dernière réduction, on remplace toutes les transitions $T(q,a) = q_1 \vee \ldots \vee q_p$ par $T(q,a) = (q_1 \wedge q) \vee \ldots \vee (q_p \wedge q)$. Les détails sont à faire en exercice. □

6.6 Exercices

Exercice 6.1 *1. Quel est le langage des mots finis et infinis reconnus par l'automate de la figure 6.7.*

2. Même question pour l'automate de la figure 6.8.

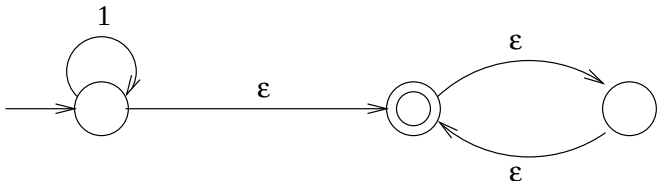


FIG. 6.7 – Automate avec transitions vides.

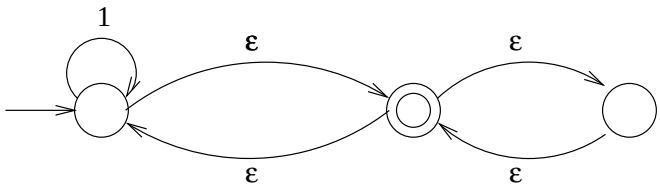


FIG. 6.8 – Automate avec transitions vides.

Exercice 6.2 *Soit $\mathcal{A}$ un automate non-déterministe pour lequel tout mot $u \in L^\omega(\mathcal{A})$ a la propriété que tout calcul lisant u le reconnait.*

1. *Montrer qu'il existe un automate fini déterministe qui accepte le langage $L^\omega(\mathcal{A})$.*
2. *Modifier la construction du cours donnée au paragraphe 6.4 de telle sorte que l'automate obtenu accepte pour mots finis ceux qui appartiennent au complémentaire de $L(\mathcal{A})$, et pour mots infinis ceux qui appartiennent au complémentaire de $L^\omega(\mathcal{A})$.*

Exercice 6.3 *On considère l'automate $\mathcal{A}$ de la figure 6.2. On demande de*

1. *Donner un automate complet équivalent $\mathcal{A}'$, c'est à dire qui reconnaisse les même mots finis et infinis.*
2. *Donner un automate complet $\mathcal{A}_1$ tel que $L(\mathcal{A}_1)$ soit le langage complémentaire de $L(\mathcal{A})$.*
3. *Donner un automate complet $\mathcal{A}_2$ tel que $L^\omega(\mathcal{A}_2)$ soit le langage complémentaire de $L^\omega(\mathcal{A})$.*
4. *Donner un automate $\mathcal{A}_3$ tel que $L(\mathcal{A}_3)$ soit le langage complémentaire de $L(\mathcal{A})$ et $L^\omega(\mathcal{A}_3)$ soit le langage complémentaire de $L^\omega(\mathcal{A})$.*
5. *Caractériser la propriété de l'automate $\mathcal{A}$ qui a permis d'exhiber l'automate $\mathcal{A}_3$ ayant la propriété demandée.*

Exercice 6.4 *Soit L_n le langage des mots finis sur l'alphabet $\{a, b\}$ pour lesquels il existe une paire de lettres distinctes à distance n l'une de l'autre, et $\overline{L_n}$ son complémentaire. De même, on notera par L_n^ω le langage des mots infinis sur l'alphabet $\{a, b\}$ pour lesquels il existe une paire de lettres distinctes à distance n l'une de l'autre, et $\overline{L_n^\omega}$ son complémentaire.*

On demande de montrer que

1. *L_n est reconnaissable par un automate fini déterministe. Quelle est la taille de l'automate fini déterministe minimal le reconnaissant ?*
2. *L_n est reconnaissable par un automate fini non-déterministe. Quelle est la taille de l'automate fini non-déterministe minimal le reconnaissant ?*
3. *L_n est reconnaissable par un automate fini alternant. Quelle est la taille de l'automate fini alternant minimal le reconnaissant ?*
4. *L_n^ω est reconnaissable par un automate fini déterministe. Quelle est la taille de l'automate fini déterministe minimal le reconnaissant ?*
5. *L_n^ω est reconnaissable par un automate fini non-déterministe. Quelle est la taille de l'automate fini non-déterministe minimal le reconnaissant ?*
6. *L_n^ω est reconnaissable par un automate fini alternant. Quelle est la taille de l'automate fini alternant minimal le reconnaissant ?*

Exercice 6.5 *Montrer que le vide des automates alternants est $PSPACE$-complet.*

Exercice 6.6 *Donner un algorithme décidant la finitude du langage des mots infinis d'un automate de Büchi. Quel est sa complexité ?*

Exercice 6.7 *Montrer que les automates alternants sont clos par union et intersection.*

Chapitre 7

Vérification de formules temporelles

Nous considérons ici des automates dont les états sont assimilés aux propriétés qu'ils vérifient.

7.1 Vérification des formules de CTL

Définition 7.1 *ϕ est une formule d'état si*

$$\forall u, v\ \forall i, j\ u(i) = v(j) \Longrightarrow u, i \models \phi \text{ ssi } v, j \models \phi$$

Pour de telles formules, on peut bien sûr définir la sémantique directement en fonction des états, et on peut alors la noter sous la forme $q \models \phi$. C'est cette propriété de CTL qui rend la vérification de formules de CTL simple algorithmiquement.

Théorème 7.2 *Les formules de CTL sont des formules d'état.*

Preuve: Par récurrence sur la structure des formules de CTL et par cas. On note ϕ une formule quelconque, (u, i) et (v, j) deux chemins accompagés de leurs états courants.

1. $\phi = p : u(i) = v(j) \Longrightarrow p \in u(i)$ ssi $p \in v(j)$ et donc $u, i \models \phi$ ssi $v, j \models \phi$.
2. $\phi = \neg\psi$ où ψ est une formule de CTL : Soit $u(i) = v(j)$. Par définition, $u, i \models \neg\psi$ ssi $u, i \not\models \psi$. Par hypothèse de récurrence, ψ est une formule d'état, et donc $u, i \models \psi$ ssi $v, j \models \psi$, ce qui équivaut à $u, i \not\models \psi$ ssi $v, j \not\models \psi$, et par définition, $v, j \not\models \psi$ ssi $v, j \models \neg\psi$.
3. $\phi = \phi_1 \wedge \phi_2$ où ϕ_1 et ϕ_2 sont des formules de CTL : la preuve est analogue à la précédente.
4. $\phi = \phi_1 \vee \phi_2$ où ϕ_1 et ϕ_2 sont des formules de CTL : la preuve est analogue aux deux précédentes.
5. $\phi = AX\psi$ où ψ est une formule de CTL : Soit $u(i) = v(j)$. Par définition, $u, i \models AX\psi$ si et seulement si, pour tout calcul u' prolongeant u à partir de i, on a $u', i+1 \models \psi$. Notons que pour tout calcul u' prolongeant u à partir de l'état i, il existe un calcul v' prolongeant v à partir de l'état j puisque $u(i) = v(j)$, et vice-versa. En notant $(u', i), (v', j)$ ces paire de calculs, on en déduit par hypothèse de récurrence appliquée à ψ que $u', i+1 \models \psi$ ssi $v', j+1 \models \psi$, ce qui, par définition, équivaut à $v, j \models AX\psi$.
6. $\phi = A\phi_1 U \phi_2$ ou $\phi = E\phi_1 U \phi_2$, où ϕ_1 et ϕ_2 sont des formules de CTL : la preuve est analogue à la précédente.
7. $\phi = A\psi$ ou $\phi = EX\psi$, où ψ est une formule de CTL : la preuve est analogue aux précédentes.

□

Ce résultat permet de travailler directement sur l'automate $\mathcal{A}$ par une procédure de marquage des états : pour chaque état $q \in \mathcal{A}$ et chaque sous-formule ψ de ϕ, on va calculer et stocker le résultat de $q \models \phi$. La procédure est donnée dans un style impératif pour en optimiser la complexité par l'utilisation d'effets de bords . Nous ne considérons pas les combinateurs superflus F et G.

```
Procedure marquage(phi)
%utilise un tableau indexé par les états q de l'automate A et les
%sous-formules psi de phi, dont les éléments sont notes q.psi ; on
%notera app(p,q) si la propriété p de Prop appartient à l'etat q.
Cas phi=p Alors Faire Pour tout q dans Q
                      Faire Si app(p,q) Alors Faire q.phi:=true
                            Sinon Faire q.phi:=false
    phi=not psi Alors Faire marquage(psi); Pour tout q dans Q
                                           Faire q.phi:= non q.psi
    phi=phi1 and phi2 Alors Faire marquage(phi1); marquage(phi2);
                                 Pour tout q dans Q
                                 Faire q.phi:= q.phi1 et q.phi2
    phi=phi1 or phi2 Alors Faire marquage(phi1); marquage(phi2);
                                Pour tout q dans Q
                                Faire q.phi:= q.phi1 ou q.phi2
    phi=E X psi Alors Faire marquage(psi);
                      Pour tout q dans Q
                      Faire Si il existe une transition (q,q') t.q.
                               q'.psi=true
                            Alors Faire q.phi:=true
                            Sinon Faire q.phi:=false
    phi=A X psi Alors Faire marquage(psi);
                      Pour tout q dans Q
                      Faire Si il existe une transition (q,q') t.q.
                               q'.psi=false
                            Alors Faire q.phi:=false
                            Sinon Faire q.phi:=true
    phi=E phi1 U phi2 Alors Faire
        %on utilisera une liste L des etats pour lesquels phi
        %est vraie, construite en parcourant à l'envers sans
        %boucler les transitions de l'automate.
        marquage(phi1); marquage(phi2); L:=vide;
        Pour tout q dans Q Faire q.marque:= false; q.phi:=false.
        Pour tout q dans Q Faire Si q.phi2 Alors Faire L:=L::{q}.
        Tantque L=/=vide Faire q:=choix(L); L:=L\{q}; q.phi:=true;
                                Pour toute transition (q',q)
                                Faire Si non q'.marque Alors
                                      Faire q'.marque:= true;
                                            Si q'.phi1 alors Faire L:=L::{q'}
                          %on notera par degré(q) le nombre de successeurs de
                          %l'état q dans l'automate. On utilisera un tableau
                          %q.nb pour parcourir tous les successeurs d'un état.
    phi=A phi1 U phi2 Alors Faire
        marquage(phi1); marquage(phi2); L:=vide;
        Pour tout q dans Q Faire q.nb=degre(q); q.phi:=false
        Pour tout q dans Q Faire Si q.phi2 Alors Faire L:=L::{q}
        Tantque L=/=vide Faire q:=choix(L); L:=L\{q}; q.phi:=true;
                                Pour toute transition (q',q)
                                Faire q'.nb:=q'.nb - 1
                                      Si q'.nb = 0 Et Non q'.phi Et q'.phi1
                                      Alors Faire L:=L::{q'}
Fin Procedure
```

Théorème 7.3 *La procédure Marquage(ϕ) est correcte et retourne son résultat en temps $\mathcal{O}(\mid \mathcal{A} \mid \times \mid \phi \mid)$ et en espace $\mathcal{O}(\mid Q \mid \times \mid \phi \mid)$.*

Preuve: La complexité en espace est bien celle annoncée puisque l'on remplit un tableau dont la taille est le nombre d'états de $\mathcal{A}$ par le nombre de sous-formules de ϕ, et que l'on utilise des tableaux ou listes auxiliaires dont la taille est toujours bornée par celle de l'automate $\mathcal{A}$. En ce qui concerne la correction et la complexité en temps, la preuve a lieu par récurrence sur la structure des formules de CTL.

1. $\phi = p$. La correction est claire, et le résultat de complexité est évident puisque l'on parcourt tous les états de l'automate $\mathcal{A}$ une fois et une seule.
2. $\phi = \neg\psi$ où ψ est une formule de CTL. Par hypothèse de récurrence, la propriété est vraie pour ψ. La correction est donc claire. Comme on effectue une boucle sur les états de l'automate, la complexité en temps est $\mathcal{O}(\mid \mathcal{A} \mid \times \mid \psi \mid) + \mid \mathcal{A} \mid$, ce qui prouve le résultat.
3. $\phi = \phi_1 \wedge \phi_2$ où ϕ_1 et ϕ_2 sont des formules de CTL : la preuve est analogue à la précédente.
4. $\phi = \phi_1 \vee \phi_2$ où ϕ_1 et ϕ_2 sont des formules de CTL : la preuve est analogue aux deux précédentes.
5. $\phi = EX\psi$ où ψ est une formule de CTL. Par hypothèse de récurrence, la propriété est vraie pour ψ. Pour la correction, il suffit de remarquer que `A X psi` est vraie dans un état q à condition que `psi` soit vraie pour au moins un successeur de q. Comme on visite pour cela une fois et une seule chaque noeud et chaque transition de l'automate $\mathcal{A}$, la complexité en temps est $\mathcal{O}(\mid \mathcal{A} \mid \times \mid \psi \mid) + \mid \mathcal{A} \mid$, ce qui prouve le résultat.
6. $\phi = AX\psi$ où ψ est une formule de CTL. Par hypothèse de récurrence, la propriété est vraie pour ψ. Pour la correction, il suffit de remarquer que `A X psi` est vraie dans un état q à condition que `psi` soit vraie de tous les successeurs de q. Comme on visite pour cela une fois et une seule chaque noeud et chaque transition de l'automate $\mathcal{A}$, la complexité en temps est identique au cas précédent.
7. $\phi = E\phi_1 U \phi_2$, où ϕ_1 et ϕ_2 sont des formules de CTL. Par hypothèse de récurrence, la propriété est vraie pour ϕ_1 et ϕ_2.

   Pour la correction, il suffit de remarquer que `E phi1 U phi2` est vraie dans un état q à condition que `phi2` soit vraie dans l'état q ou encore que `phi1` soit vraie dans l'état q et que `phi` soit vraie dans un état successeur de q. La mise en oeuvre de cette remarque consiste à calculer une liste `L` des états qui vérifient `E phi1 U phi2` en l'initialisant avec les états qui vérifient `phi2`, puis à ajouter un état `q'` à `L` à condition qu'il vérifie `phi1` et qu'il existe une transition de `q'` vers un état de `L`. C'est ce que fait l'algorithme, en cherchant parmi les antécédents non encore visités des sommets de `L` (c'est le rôle du tableau `.marque`) ceux qui vérifient `phi2`.

   Pour ce faire, les deux boucles `Pour` qui suivent les appels récursifs ont une taille bornée par $\mathcal{A}$. La boucle `Tantque` qui suit visite chaque état et chaque transition de l'automate au plus une fois. Elle prend donc elle-aussi un temps borné par la taille de l'automate $\mathcal{A}$. Au total le temps utilisé est de l'ordre de $\mathcal{O}(\mid \mathcal{A} \mid \times \mid \psi_1 \mid) + \mathcal{O}(\mid \mathcal{A} \mid \times \mid \psi_2 \mid) + 3\mid \mathcal{A} \mid$, ce qui prouve le résultat annoncé.
8. $\phi = A\phi_1 U \phi_2$, où ϕ_1 et ϕ_2 sont des formules de CTL. Par hypothèse de récurrence, la propriété est vraie pour ϕ_1 et ϕ_2.

   Pour la correction, il suffit de remarquer que `A phi1 U phi2` est vraie dans un état q à condition que `phi2` soit vraie dans l'état q ou que `phi1` soit vraie dans l'état q et que `phi` soit vraie dans tout état successeur de q. La mise en oeuvre de cette remarque consiste à calculer une liste `L` des états qui vérifient `E phi1 U phi2` en l'initialisant avec les états qui vérifient `phi2`, puis à ajouter un état `q'` non encore visité (c'est le rôle du test `Non q'.phi` et de l'initialisation préalable de `q.phi` à `false`) à `L` à condition que toutes les transitions

issues de `q'` mènent à un état de `L` (c'est le rôle du test `q'.nb = 0`) et qu'il vérifie `phi1` (c'est le rôle du test `q'.phi1`).
Pour ce faire, les deux boucles `Pour` qui suivent les appels récursifs ont une taille bornée par $\mathcal{A}$. La boucle `Tantque` qui suit visite chaque état et chaque transitgion de l'automate au plus une fois. Elle prend donc elle-aussi un temps borné par la taille de l'automate $\mathcal{A}$. Au total le temps utilisé est de l'ordre de $\mathcal{O}(\mid \mathcal{A} \mid \times \mid \psi_1 \mid) + \mathcal{O}(\mid \mathcal{A} \mid \times \mid \psi_2 \mid) + 3 \mid \mathcal{A} \mid$, ce qui prouve le résultat annoncé.

□

Cette complexité linéaire en temps et en espace est bien sûr le point fort de la logique CTL.

7.2 Vérification des formules de LTL

Le model checking de LTL n'est plus linéaire, mais $PSPACE$-complet. Il repose sur des techniques de théorie des langages, et les automates de Büchi alternants y jouent un rôle essentiel puisque l'on va s'intéresser à des exécutions finies ou infinies.

La formulation de nos automates étant un peu différente de celle adoptée précemment, nous allons reformuler la notion de calcul qui seront vus comme des mots finis ou infinis sur $\mathcal{P}(\mathcal{P}rop)$, où $\mathcal{P}rop$ désigne les proposition atomiques de l'automate $\mathcal{A}$ considéré. Cela est aussi vrai dans l'autre formulation via la fonction d'étiquetage. Cette formulation peut paraître plus générale, dans la mesure où elle n'impose pas qu'un calcul s'exprime comme la composition d'une fonction de $\underline{n}$ dans un ensemble fini Q muni d'une fontion d'étiquetage de Q dans $\mathcal{P}(\mathcal{P}rop)$. Elles sont en fait équivalentes, car les formules temporelles ne vérifient que des calculs qui se décomposent de cette manière.

Théorème 7.4 *Étant donnée une formule ϕ de LTL, il existe un automate alternant $\mathcal{A}_\phi$ tel que $L^\omega(\mathcal{A}_\phi)$ soit le langage des calculs qui satisfont la formule ϕ, où*

$$
\begin{array}{rcl}
\mathcal{A}_\phi & = & (V_t, Q, q_0, F, T) \\
V_t & = & \mathcal{P}(\mathcal{P}rop) \\
Q & = & \left\{ \begin{array}{c} \{\psi, \neg\psi \mid \psi \textit{ est une sous-formule positive de } \phi\} \cup \\ \{\psi, \neg\psi \mid \neg\psi \textit{ est une sous-formule négative de } \phi\} \cup \\ \{true, false\} \end{array} \right. \\
q_0 & = & \phi \\
F & = & \{\neg(\xi U \psi) \mid \neg(\xi U \psi) \in Q\} \cup \{true\} \\
T(p, A) & = & true \quad \textit{si } p \in A \\
T(p, A) & = & false \quad \textit{si } p \notin A \\
T(true, A) & = & true \\
T(false, A) & = & false \\
T(\xi \wedge \psi, A) & = & (T(\xi, A) \wedge T(\psi, A)) \downarrow \\
T(\neg\psi, A) & = & (\neg T(\psi, A)) \downarrow \\
T(X\psi, A) & = & \psi \\
T(\xi U \psi, A) & = & T(\psi, A) \vee (T(\xi, A) \wedge (\xi U \psi)) \downarrow
\end{array}
$$

où $\psi \downarrow$ désigne la forme normale disjonctive de la formule ψ, dont on suppose, sans perte de généralité, qu'elle ne contient pas de double négation.

Avant de prouver le résultat, il est important de remarquer que la définition

$$T(\xi U \psi, A) = T(\psi, A) \vee (T(\xi, A) \wedge (\xi U \psi)) \downarrow$$

résulte en deux types de transitions conjonctives comme représenté à la figure 7.1, pour lesquelles les états atteints sont soit des états de taille au plus égale à celle de ψ, soit des états de taille au plus égale à celle de ξ.

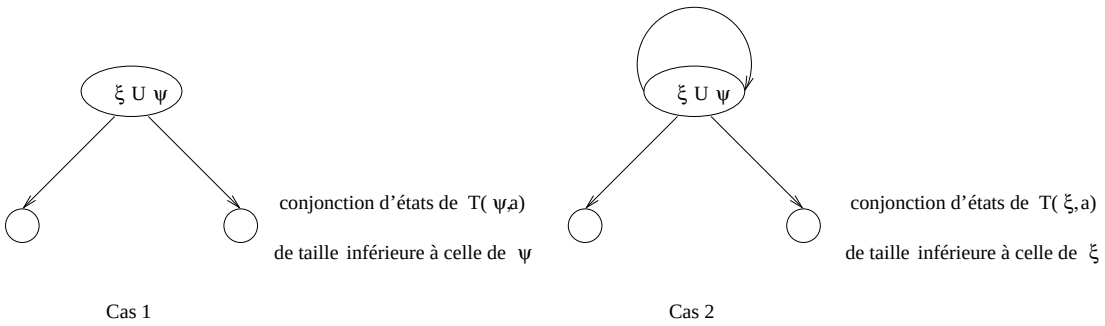


FIG. 7.1 – Transitions issues des formules Until.

De même, la définition induite de

$$\begin{aligned} T(\neg(\xi U \psi), A) &= (\neg T(\psi, A) \wedge (\neg T(\xi, A) \vee \neg(\xi U \psi))) \downarrow \\ &= ((\neg T(\psi, A)) \downarrow \wedge (\neg T(\xi, a)) \downarrow) \downarrow \vee ((\neg T(\psi, a)) \downarrow \wedge \neg(\xi U \psi)) \downarrow \end{aligned}$$

résulte de manière analogue en deux types de transitions conjonctives comme représenté à la figure 7.2.

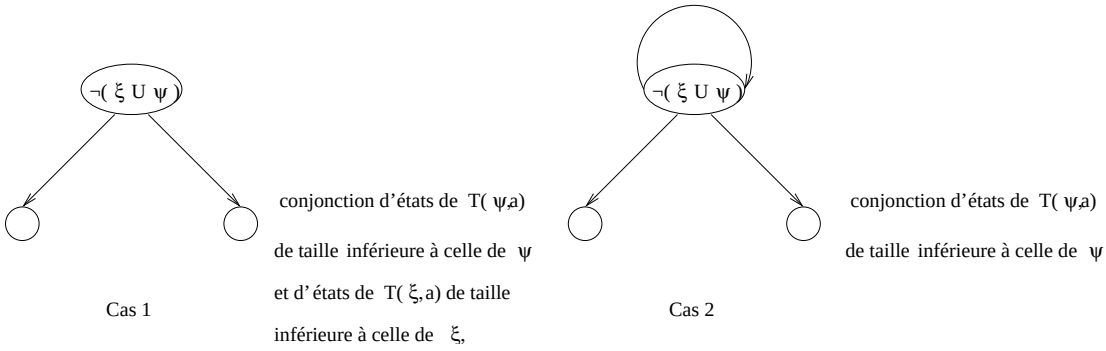


FIG. 7.2 – Transitions issues des négations de formules Until.

Dans tous les autres cas, les transitions conjonctives issues d'un état ϕ vont vers des états qui sont tous des sous-formules de ϕ. Les récurrences seront donc faciles, sauf dans les deux cas précités.

Preuve: On montre par récurrence sur la structure de la formule ψ que pour toute exécution u et tout ordinal $i < \omega$, on a $u, i \models \psi$ ssi $u_i \in L^\omega(\mathcal{A}_\psi)$ où

$\forall j < \omega \; u_i(j) = u(i + j)$, et

$\mathcal{A}_\psi$ est l'automate obtenu à partir de $\mathcal{A}_\phi$ en prenant ψ comme état initial.

On obtient l'énoncé du théorème en prenant $\psi = \phi$ et $i = 0$.

Soit donc u un mot infini sur $\mathcal{P}(\mathcal{P}rop)$ vu comme une exécution de l'automate $\mathcal{A}$, c'est-à-dire comme une application de $\overline{n}$ dans $\mathcal{P}(\mathcal{P}rop)$ pour $n = \omega$. On raisonne par cas suivant la forme de la formule ψ :

1. $\psi = p$:

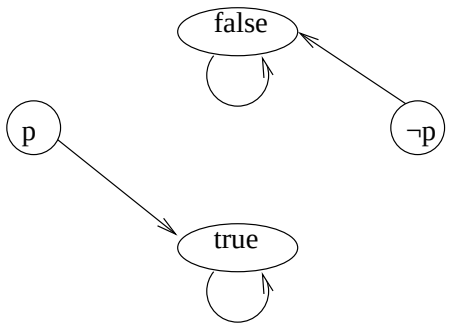


FIG. 7.3 – Transitions issues des formules atomiques.

$u, i \models p$
iff $p \in u(i)$ par définition de $\models$
iff $p \in u_i(0)$ par définition de u_i
iff $T(p, u_i(0)) = true$ par définition de T
iff $u_i \in L^\omega(\mathcal{A}_p)$ par définition de l'automate $\mathcal{A}_p$ représenté à la figure 7.3,
puisqu'une fois en l'état $true \in F$, on boucle indéfiniment sur cet état qui accepte donc le calcul u_i.

2. $\psi = \psi_1 \wedge \psi_2$:
   $u, i \models \psi$
   iff $u, i \models \psi_1$ et $u, i \models \psi_2$ par définition de $\models$
   iff $u_i \in L^\omega(\mathcal{A}_{\psi_1})$ et $u_i \in L^\omega(\mathcal{A}_{\psi_2})$ par hypothèse de récurrence
   iff $u_i \in L^\omega(\mathcal{A}_\psi)$ par définition de l'automate (lemme simple à prouver) .
3. $\psi = X\xi$:
   $u, i \models X\xi$
   ssi $u, i+1 \models \xi$ par définition de $\models$
   ssi $u_{i+1} \in L^\omega(\mathcal{A}_\xi)$ par hypothèse de récurence
   ssi $u_i \in L^\omega(\mathcal{A}_{X\xi})$ par définition de l'automate.
4. $\psi = \xi U \tau$: on distingue deux cas, représentés à la figure 7.1.
   Cas 1 : Il n'y a pas de boucle sur l'état $\xi U \tau$. Par hypothèse de récurrence, on reconnait u_{i+1} par les successeurs de τ en lisant $u_i(0)$, donc on reconnait u_i par $\mathcal{A}_\psi$.
   Cas 2 : Il y a une boucle sur l'état $\xi U \tau$. Par hypothèse de récurrence, et comme précédemment, $u_i \in L^\omega(\mathcal{A}_\xi)$. De manière plus générale, si l'on emprunte la boucle p fois, alors tant que l'on emprunte la boucle, c'est-à-dire $\forall k$ tel que $p > k \geq i$ alors $u_k \in L^\omega(\mathcal{A}_\xi)$; dès que l'on sort de la boucle, alors $u_p \in L^\omega(\mathcal{A}_\tau)$.
   On aura donc bien reconnu $\xi U \tau$, à condition que ls calculs qui empruntent la boucle une infinité de fois ne soient pas reconnus par l'automate, ce qui est le cas.
5. $\psi = \neg\xi$: on distingue deux cas suivant la forme de la formule ξ.
   - Si la formule ξ n'est pas une formule de la forme $\phi_1 U \phi_2$, alors les transitions conjonctives issues de l'état ξ mènent toutes à des états de taille plus petite que ξ, et donc
     $u, i \models \neg\xi$
     iff $u, i \not\models \xi$ par définition de $\models$
     iff $u_i \notin L^\omega(\mathcal{A}_\xi)$ par hypothèse de récurrence
     iff $u_i \in L^\omega(\mathcal{A}_{\neg\xi})$ par définition de T.
   - La formule ξ est de la forme $\phi_1 U \phi_2$: dans ce cas, la situation est rendue complexe par la boucle située sur l'état $\neg\xi$, comme représenté au cas 2 de la figure 7.2, et la récurrence ne suffit pas pour conclure puisque l'on a une boucle sur l'état ψ. Dans ce cas, cette boucle est soit empruntée un nombre fini de fois jusqu'à sortir par le cas 1 décrit à la figure 7.2,

ce qui assure que la formule ξ est fausse dans ce cas là par application du raisonnement par récurrence précédent. Où bien la boucle est empruntée un nombre infini de fois, et dans ce cas la formule ϕ_2 ne sera jamais vraie, ce qui assure que la formule ξ n'est pas vraie le long de ce calcul.

□

Décider si un automate $\mathcal{A}$ valide une formule ϕ revient donc à décider le vide du langage $L_\omega(\mathcal{A}) \cap L^\omega(\mathcal{A}_{\neg\phi})$. Comme le vide du langage reconnu par un automate alternant est dans la classe $PSPACE$, il en sera de même de la validation d'une formule par un automate de Büchi, pourvu que l'automate A_ϕ précédent soit de taille polynomiale en la formule ϕ. Cela ne sera pas le cas si nos automates alternants utilisent pour transitions des formules en forme normale conjonctive sur les noms d'états (la mise en CNF provoquant une explosion combinatoire), mais ce sera par contre le cas pour une définition des automates alternants où les transitions sont des formules queslconques sur le langage propositionnel des noms d'états.

7.3 Vérification des formules de CTL*

Décider la validité d'une formule de CTL* vis à vis d'un automate $\mathcal{A}$ requiert des techniques plus sophistiquées, basées sur des automates alternants particuliers, dit hésitants, sans lesquels la complexité optimale ne peut être obtenue [3]. Notons que l'on peut par cette technique décider les formules de CTL en temps linéaire [6].

7.4 Exercices

Exercice 7.1 *On demande de vérifier les propriétés exprimées en CTL de la description d'un ascenseur à trois étages faite au chapitre 3.*

Exercice 7.2 *On demande de vérifier les propriétés exprimées en LTL de la description du climat faite en chapitre 3.*

Chapitre 8

Automates et logiques temporisés

On modélise le temps par des variables entières -on parle de temps discret-, ou rationnelles (ou réelles) -on parle alors de temps dense. Une technique rudimentaire consiste à avoir une horloge globale, modélisée par un compteur qui éffectue ses transitions de manière synchrone avec les autres automates. La variable compteur peut alors être consultée -mais pas modifiée- par les autres automates. Cette solution est très rigide, il est souvent beaucoup plus aisé d'avoir des horloges locales aux états. Cela facilite en particulier la remise à zéro des horloges, qui peuvent alors être vues comme des alarmes. Dans notre exemple représenté à la figure 8.1, nous avons utilisé une unique variable d'horloge, qui est incrémentée aussi bien par le digicode que par la porte. Nous aurions pu utiliser deux horloges au lieu d'une, mais il est possible ici de réutiliser la même horloge ce qui a l'avantage de créer moins d'états dans l'automate fini équivalent.

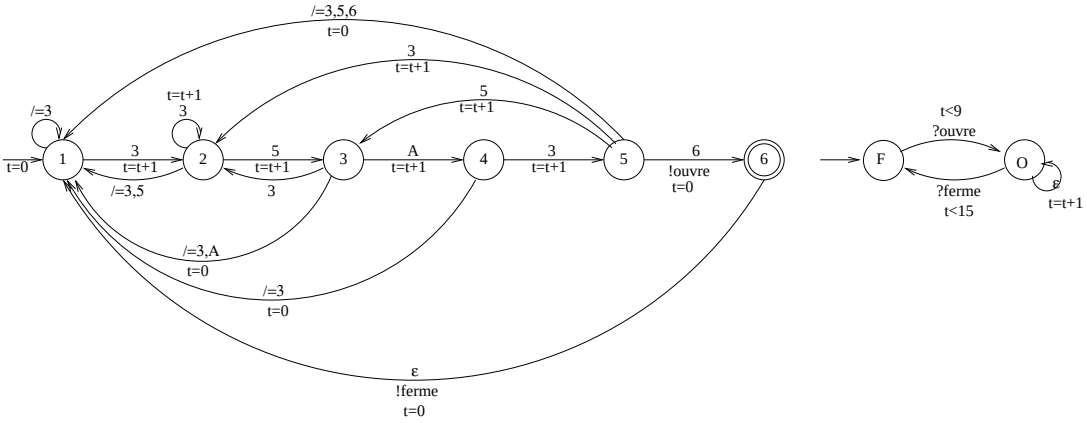


FIG. 8.1 – Digicode à utilisation multiples et temporisations.

Ce codage du temps est un peu lourd : il faut en gérer explicitement l'écoulement le long des transitions des automates. De plus, on ne peut changer la vitesse de l'écoulement du temps sans changer l'automate lui-même (ou du moins l'ensemble de synchronisation). Enfin, il faut parfois changer la structure d'un automate pour simuler l'écoulement du temps dans un état donné. Cette description n'est donc pas assez abstraite, et s'avère en conséquence sujette à erreurs. Le remède consiste à introduire une nouvelle classe d'automates.

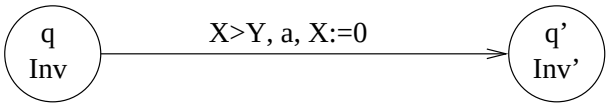


FIG. 8.2 – Représentation des transitions d'un automate temporisé.

8.1 Automates temporisés

Définition 8.1 *Un* automate temporisé $\mathcal{A}$ *est un sextuplet* $(V_t, Q, H, \mathcal{I}nv, F, T)$ *où*

1. V_t *est le* vocabulaire *de l'automate, dont les éléments sont appellés* noms d'actions *;*
2. Q *est l'ensemble fini des* états de contrôle *de l'automate ;*
3. H *est un ensemble fini de variables réelles* $h_1, \ldots, h_n$ *appellées* horloges *de l'automate. On lui associe l'ensemble* $\mathcal{T}cons$ *des conjonctions finies appellées* contraintes temporelles *d'expressions de la forme* $h \sim c$*, où* $h \in H,\ c \in \mathcal{R}at$ *et* $\sim \in \{\geq, >, \leq, <, =, \neq\}$ *;*
4. $\mathcal{I}nv(q) \in \mathcal{T}cons$ *est appellé* invariant *de l'état* q *;*
5. $F \subseteq Q$ *est l'ensemble des états acceptants de l'automate ;*
6. $T \subseteq Q \times V_t \times \mathcal{T}cons \times \mathcal{P}(H) \times Q$ *est la* fonction de transition *de l'automate.*

On notera par $h(q,t)$ *la valeur (ou* valuation*) de l'horloge* $h \in H$ *à l'instant* t*, l'automate étant dans l'état* q*, et par* $H(q,t) \in \mathcal{R}^{|H|}$ *(ou simplement* V*) le vecteur des valeurs d'horloges. On écrira une transition* $(q, a, \alpha, \zeta, q') \in T$ *sous la forme* $(q, V) \xrightarrow{(\alpha, a, \zeta)} (q', V')$*, et on la représentera comme indiqué à la figure 8.2 ;* $\alpha \in \mathcal{T}cons$ *est appellée* garde *ou* condition de franchissement*,* $\zeta \subseteq H$ *est l'ensemble des horloges remises à zéro par la transition, et* V *et* V' *sont les vecteurs des valeurs d'horloge respectives avant et après la transition.*

Un automate temporisé est dit déterministe *ssi* $\forall q \in Q,\ \forall (q, a, \alpha, \zeta, q') \in T,\ \forall (q, a, \beta, \zeta', q'') \in T,\ \forall V \in \mathcal{R}^{|H|}$ *tel que* $V \models \mathcal{I}nv(q), V \models \alpha$ *implies* $V \not\models \beta$*.*

La syntaxe des invariants et gardes est choisie délibérémment simple, dans le but de faciliter les développements techniques. On peut en fait autoriser des conjonctions de contraintes temporelles atomiques de la forme $h \sim c$ ou $h \sim h' + c$, où h et h' sont des horloges distinctes. Une syntaxe plus riche autorisant l'opération d'addition conduit immédiatement à l'indécidabilité du problème du vide pour cette classe d'automates.

Dans ce modèle, la progression du temps est universelle, et les transitions entre états sont instantanées. Le temps s'écoule dans les états, et les différentes horloges sont indépendantes les unes des autres, même si elles progressent à la même vitesse : elles servent à mesurer des délais entre actions, grâce aux remises à zéro à des instants arbitraires. Dans l'exemple du digicode temporisé représenté à la figure 8.3, il y a une horloge pour contrôler l'ouverture de la porte, et une seconde pour en contrôler la fermeture. Pour des raisons de place, nous n'avons pas utilisé la notation en triplets.

Définition 8.2 *On appelle* configuration *(ou* état global*) à l'instant* t *d'un automate temporisé* $\mathcal{A} = (V_t, Q, i, F, H, T)$*, toute paire* $(q, V = H(q,t))$ *composée d'un état de contrôle* q *de l'automate et d'une valuation* V *des différentes horloges telle que* $V \models \mathcal{I}nv(q)$*.*

Un automate temporisé change de configuration de deux façons différentes :

- en effectuant une *transition d'état* de l'automate, notée $(q, H(q,t)) \xrightarrow{t,a} (q', H(q',t))$, qui satisfasse les gardes et remette certaines horloges à zéro, les autres restant inchangées ;

- en effectuant une *transition temporelle* de l'automate, notée $(q, H(q,t)) \xrightarrow{\tau} (q, H(q, t+\tau))$, qui laisse s'écouler un délai $\tau \in \mathcal{R}$ satisfaisant l'invariant de l'état et incrémente de τ les valeurs de toutes les horloges.

Il peut y avoir plusieurs transitions d'état successives, le temps étant figé. Un calcul est donc une fonction constante par morceaux, les sauts éventuels successifs à un même instant étant ordonnés,

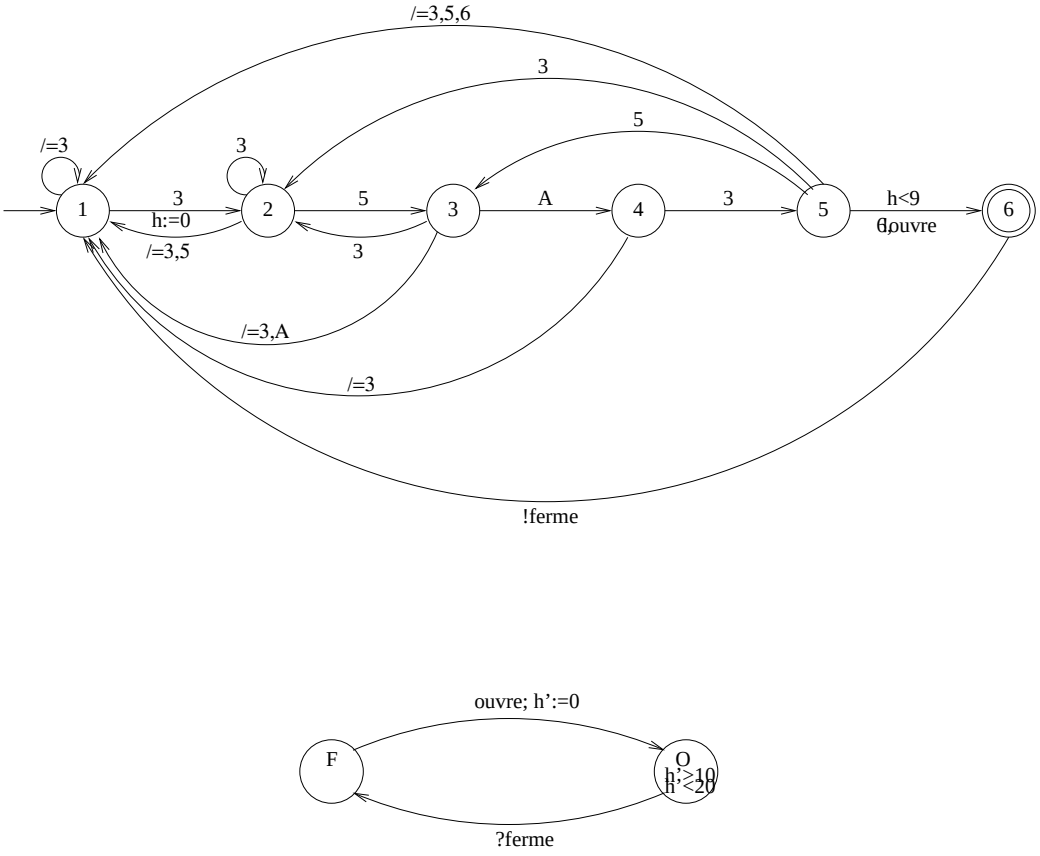


FIG. 8.3 – Digicode temporisé à utilisations multiples.

comme représenté à la figure 8.4 : c'est donc une application de $\mathcal{R}^+$ dans les suites de configurations que l'on peut reconstruire à partir de la suite de ses discontinuités.

Définition 8.3 *Étant donné un automate temporisé* $\mathcal{A} = (V_t, Q, H, \mathcal{I}nv, F, T)$*, on appelle* calcul temporisé *toute suite infinie de* transitions entre configurations *de la forme*

$$(q_0, H(q_0,0)) \xrightarrow{\tau_0} (q_0, H(q_0,t_1)) \xrightarrow{t_1,a_1} (q_1, H(q_1,t_1)) \xrightarrow{\tau_1} (q_1, H(q_1,t_2)) \xrightarrow{t_2,a_2} (q_2, H(q_2,t_2))$$

$$\ldots (q_i, H(q_i,t_i)) \xrightarrow{\tau_i} (q_i, H(q_i,t_{i+1})) \xrightarrow{t_{i+1},a_{i+1}} (q_{i+1}, H(q_{i+1},t_{i+1})) \ldots$$

généralement abbrégé en sa suite infinie de transitions d'états *:*

$$q_0 \xrightarrow{t_1,a_1} q_1 \xrightarrow{t_2,a_2} q_2 \ldots q_i \xrightarrow{t_{i+1},a_{i+1}} q_{i+1} \ldots$$

lisant le mot $(t_1,a_1)(t_2,a_2)\ldots(t_i,a_i)(t_{i+1},a_{i+1})\ldots$ *et telle que*

1. $t_0 = 0$, $H(q_0,0)$ *est la valuation nulle,* $\tau_0 = t_1$ *et* $\forall i \geq 0\ \tau_i = t_{i+1} - t_i$ *;*
2. *la suite infinie* $t_1, t_2, \ldots, t_i, t_{i+1}, \ldots$ *est non décroissante ;*
3. $\forall i \geq 0,\ \exists (q_i, a_i, \alpha_i, \zeta_i, q_{i+1}) \in T$ *telle que*
   (a) $\forall t \in [t_i..t_{i+1}]\ H(q_i,t) \models \mathcal{I}nv(q_i)$ *;*
   (b) $H(q_i,t_{i+1}) \models \alpha_i$ *;*
   (c) $\forall h \in \zeta,\ h(q_{i+1},t_{i+1}) = 0$ *et* $\forall h \in H \setminus \zeta,\ h(q_{i+1},t_{i+1}) = h(q_i,t_{i+1})$.

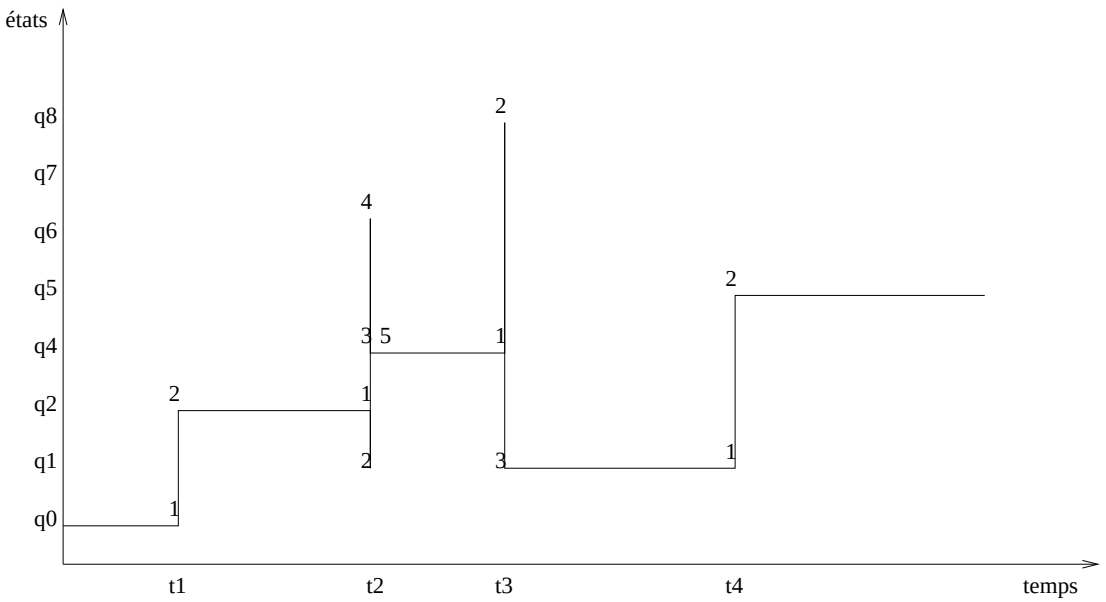


FIG. 8.4 – Graphe d'un calcul temporisé.

Le langage temporisé de l'automate, noté $\mathcal{L}^{\omega}(\mathcal{A})$ *est le langage des calculs de* $\mathcal{A}$ *qui sont réussis au sens de Büchi et satisfont l'axiome de* progrès *du temps, encore appellé* absence de comportement Zénon *:* $\forall t \in \mathcal{R}\ \exists i \in \mathbb{N}$ *such that* $t_i \geq t$.

Il n'y a malheureusement pas un unique modèle d'automate temporisé. Pour celui que nous avons donné ici, lors d'une transition d'état de q vers q' à l'instant t, les invariants des états q et q' doivent être satisfaits par les valuations d'horloges respectives $H(q,t)$ et $H(q',t)$. Dans certains modèles, l'une seulement de ces deux valuations doit être satisfaite, ce qui fait qu'il n'y a pas recouvrement du temps, et qu'il ne peut donc pas y avoir deux transitions d'état au même instant.

Le modèle des automates temporisés peut bien sûr être vu comme un produit synchronisé, avec une différence de taille toutefois : l'ensemble des états n'est plus fini en général dans ce cas, puisque les variables de temps peuvent prendre une infinité de valeurs distinctes. Il faudra donc une algorithmique spécifique pour traiter ce type d'automates.

Par ailleurs, il est bien sûr possible de faire un produit synchronisé d'automates temporisés. C'est ce que nous ferons systématiquement en pratique.

8.2 Analyse des calculs d'un automate temporisé

Les automates temporisés introduisent une nouveauté : certains calculs satisferont la condition d'acceptance (de Büchi puisque nous nous intéressons aux calculs infinis) sans être pour autant acceptés. La raison est qu'un calcul qui satisferait le paradoxe de Zénon, pour lequel Achille ne rattrappe jamais la tortue partie avant lui, ne correspondrait pas au modèle du temps du monde physique qui est le nôtre. Cette hypothèse fondamentale aura bien sûr des conséquences pour le problème de décision du vide de ces automates. Notons que tout automate, vu comme un cas particulier d'automate temporisé, a des calculs qui satisfont le paradoxe de Zénon.

Lemme 8.4 *Soit* $\mathcal{A}$ *un automate temporisé et* h *une horloge de l'automate. Pour tout calcul réussi, l'horloge* h *passe une infinité de fois par la valeur nulle, ou bien prend des valeurs arbitrairement*

grandes.

Preuve: Le temps ne pouvant être borné le long d'un calcul réussi, toute horloge qui reste bornée doit être remise à zéro infiniment souvent. □

Notons que le choix des réels comme modèle du temps a des conséquences importantes. L'automate de la figure 8.5 montre un exemple d'automate dont la suite des valeurs de l'horloge y lors de la transition de l'état 3 vers l'état 2 converge vers zéro, alors que la suite des valeurs de l'horloge x lors de la transition de l'état 2 vers l'état 3 converge vers un. Un exemple de calcul est en effet :

$$\begin{array}{llllllll}
(0,\{0,0\}) & \xrightarrow{1} & (0,\{1,1\}) & \xrightarrow{1,a} & (1,\{0,1\}) & \xrightarrow{0.5} & (1,\{0.5,1\}) & \xrightarrow{1.5,a} \\
(2,\{0.5,0\}) & \xrightarrow{0.5} & (2,\{1,0.5\}) & \xrightarrow{2,a} & (3,\{0,0.5\}) & \xrightarrow{0.25} & (3,\{0.25,0.75\}) & \xrightarrow{2.25,a} \\
(2,\{0.25,0\}) & \xrightarrow{0.75} & (2,\{1,0.75\}) & \xrightarrow{3,a} & (3,\{0,0.75\}) & \xrightarrow{0.1} & (3,\{0.1,0.85\}) & \xrightarrow{3.1,a} \\
(2,\{0.1,0\}) & \xrightarrow{0.9} & (2,\{1,0.9\}) & \xrightarrow{4,a} & (3,\{0,0.9\}) & \xrightarrow{0.05} & (3,\{0.05,0.95\}) & \xrightarrow{4.05,a} \\
(2,\{0.O5,0\}) & \ldots & & & & & &
\end{array}$$

Un tel comportement serait de fait impossible avec un temps entier, mais possible avec un temps rationnel, et en fait seule la densité du temps importe, on peut avoir les même calculs (à une approximation près) que le temps soit rationnel ou réel (pourvu que les constantes utilisées dans les invariants et dans les gardes soient rationnelles). Notons que cela est de toute façon nécessaire d'un point de vue algorithmique, puisque la comparaison de deux réels quelconques n'est en général pas décidable (mais on peut bien sûr prendre une sous-classe des réels plus riche que les rationnels, comme les algébriques).

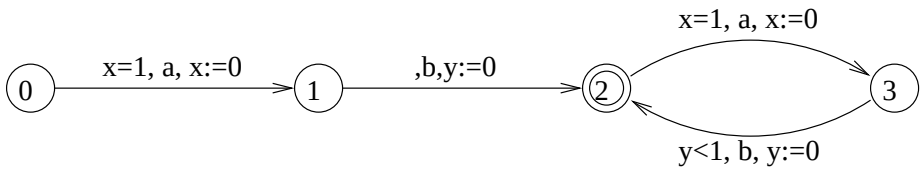


FIG. 8.5 – Calcul convergeant.

Définition 8.5 *On appelle* mot non temporisé *reconnu par le calcul* $\ldots (q_i, H(q_i,t_i)) \xrightarrow{\tau_i} (q_i, H(q_i,t_{i+1})) \xrightarrow{t_{i+1},a_{i+1}} (q_{i+1}, H(q_{i+1},t_{i+1})) \ldots$ *le mot* $a_0 \ldots a_i \ldots$ *projection du mot temporisé reconnu sur le vocabulaire* V_t.

8.3 Automate des régions

L'appartenance des horloges à des domaines infinis implique que l'automate produit équivalent à un automate temporisé donné possède une infinité d'états. Toutefois, les transitions purement temporelles, puisqu'elles se passent à l'intérieur d'un état normal, ne jouent la pluspart du temps pas un rôle très éffectif : seules celles qui déclenchent (ou peuvent déclencher) une transition d'état sont vraiment importantes. On va donc pouvoir définir une équivalence entre états de l'automate produit qui sera d'index fini, et l'automate quotient obtenu sera alors un automate fini sur lequel les techniques habituelles de vérification pourront s'appliquer.

Définition 8.6 *Étant donné un automate temporisé* $\mathcal{A} = (V_t, Q, i, F, H, T)$*, on dit que deux configurations* $(q, H(q,t))$ *et* $(q, H(q,t+\tau))$ *sont équivalentes dans l'*équivalence de comportement *si et seulement si pour toute transition sortante de q dont la garde est* ϕ*,* $H(q,t) \models \phi$ *ssi* $H(q,t+\tau) \models \phi$*.*

La relation entre configurations ainsi définie est bien une équivalence, et c'est même une bisimulation :

Lemme 8.7 *Supposons que $(q, H(q,t))$ et $(q, H(q,t+\tau))$ soient des configurations équivalentes et que $q \xrightarrow{a,t} q'$. Alors $(q', H(q',t))$ et $(q', H(q',t+\tau))$ sont des configurations équivalentes et $q \xrightarrow{a,t+\tau} q'$.*

Preuve: Conséquence immédiate de la définition. □

Nous allons maintenant voir qu'elle est d'index fini. L'automate quotient sera donc un automate fini qui décrira le comportement global du système en ce sens que toute transition de l'automate quotient sera une transition d'état du système initial et vice-versa. Calculer cet automate n'est pas chose simple. Une autre façon de voir le problème consiste à caractériser les états de manière symbolique par des formules faisant intervenir les variables d'horloge et les opérateurs de comparaison arithmétiques. Cela est possible grâce à une propriété de convexité des configurations, qui assure que si $(q, H(q,t))$ et $(q, H(q,t+\tau))$ sont équivalentes, alors $\forall \tau' \geq \tau$, $(q, H(q,t))$ et $(q, H(q,t+\tau'))$ sont équivalentes.

Définition 8.8 *On appelle* automate des régions *associé à un automate temporisé $\mathcal{A} = (V_t, Q, i, F, H, T)$, l'automate (non temporisé) $\mathcal{A}_R = (V_t, Q_R, I_R, F_R, T_R)$ dont*

1. *les états sont des paires (q, ϕ), où $q \in Q$ et la* région *est une conjonction arbitraire de formules de la forme $h_1 \in I_1 \wedge \ldots \wedge h_n \in I_n \wedge h_{\xi(1)} \lhd h_{\xi(2)} \lhd \ldots \lhd h_{\xi(n)}$, où I_j, pour chaque horloge h, est un intervalle* singulier $[k..k]$ *pour $k \leq max_h$ ou* borné $]k..k+1[$ *pour $k < max_h$ ou* non borné $]max_h..\infty[$ *où max_h est la plus grande constante apparaissant dans les formules $h \sim c$ présentes dans les gardes et invariants de l'automate $\mathcal{A}$, ξ est une permutation des indices des variables d'horloges appartenant à des intervalles singuliers ou bornés et $\lhd \in \{=, <\}$;*
2. *l'ensemble I_R des états initiaux est formé des paires $(i, \phi(h_1, h_2, \ldots, h_n))$, où ϕ est une* région diagonale, *c'est-à-dire contenant la formule $x_1 = \ldots = x_n$;*
3. *$F_R = \{(f, \phi) \mid f \in F, \phi$ ne contient pas de formule singulière$\}$ et (f, ϕ) est situé sur un cycle faisant progresser le temps ;*
4. *les transitions non vides sont les transitions d'états de l'automate de départ :*

$$T_R((q, \phi), a) = (q', \phi')$$

ssi il existe une transition $(q, a, \psi, \zeta, q') \in T$, un temps t, un intervalle de temps $\tau \geq 0$ et des valuations d'horloges $H(q,t)$, $H(q, t+\tau)$ et $H(q', t+\tau)$ telles que $(q, H(q,t)) \xrightarrow{\tau} (q, H(q, t+\tau)) \xrightarrow{t+\tau, a} (q', H(q', t+\tau))$.

Note : on peut aussi progresser par pas plus grand que 1, mais pas plus grand que le pgcd des valeurs apparaissant dans les gardes. Cela s'étend lorsque les constantes sont rationnelles. Avoir des constantes rationnelles va donc multiplier le nombre de régions.

La figure 8.6 représente l'ensemble des régions pour deux horloges dans le cas où l'on ne s'intéresse qu'à des contraintes de la forme $x \sim k$ avec $x = x_1, x_2$, $\sim \in \{=, \neq, >, \geq, <, \leq\}$ et $k = 0, 1, 2$ pour x_1 et $k = 0, 1$ pour x_2. Il y a alors 28 régions, sans compter la région vide. Certaines sont réduites à une unique configuration (comme la région r_0 décrite par $x_1 = x_2 = 0$), d'autres sont des parties ouvertes du plan (comme la région r_9 décrite par la contrainte $0 < x_2 < x_1 < 1$), les autres sont des segments (comme la région r_7 décrite par la contrainte $0 < x_2 = x_1 < 1$).

Le système démarre dans la région initiale r_0. Lorsque le temps passe, on accède à la région r_7, puis à la région r_4, etc., jusqu'à la région r_{28}. Si, au lieu de laisser passer le temps, ou éffectue une transition discrète, la remise à zéro de certaines horloges nous conduit aux régions situées sur les axes. Par exemple, la remise à zéro de x_2 dans la région r_7 nous conduit à la région r_8 d'où l'on ira en r_9, etc.

Lemme 8.9 *Les régions sont des convexes de $\mathcal{R}^n$ dont ils forment une partition.*

Preuve: Immédiat d'après leur définition. □

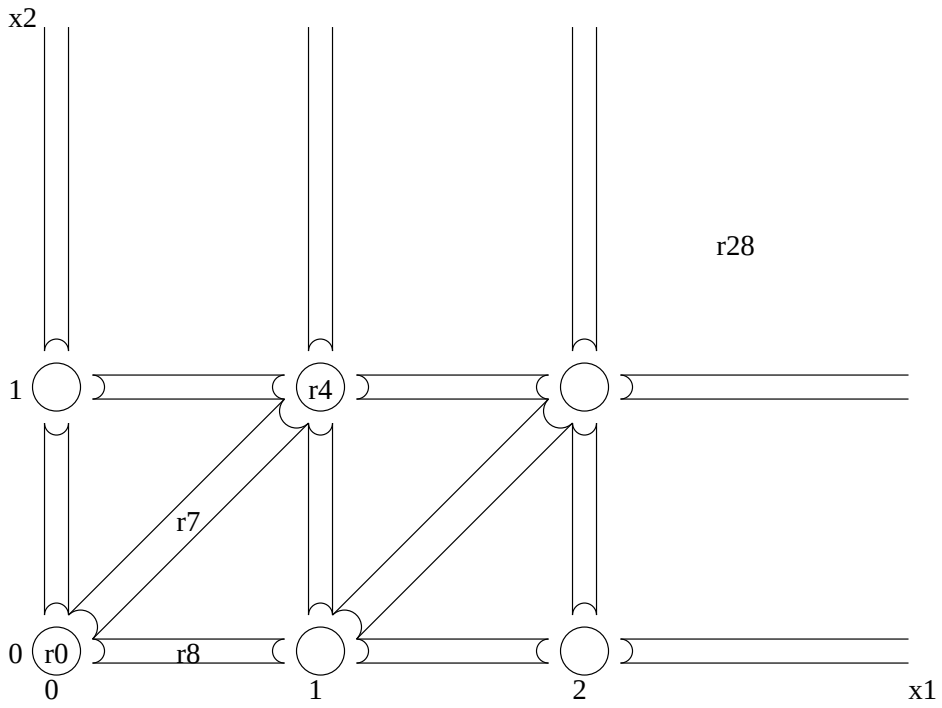


FIG. 8.6 – Régions pour les contraintes $x_1, x_2 \sim k$ avec $k = 0, 1, 2$.

Lemme 8.10 *Deux configurations d'une même région sont équivalentes.*

Preuve: Les régions sont des convexes bornés de $\mathcal{R}^n$ pour lesquels les valeurs d'horloges sont ordonnées de la même manière en chaque point du convexe, ce qui interdit une remise à zéro d'horloge à l'intérieur d'une telle région, ou des convexes non bornés. Il suffit donc de montrer qu'à l'intérieur d'un même convexe de la forme annoncée, deux valuations distinctes valident les mêmes gardes sortantes de l'état de contrôle, et les même invariants d'états. Cela résulte du fait que les gardes et les invariants sont eux-même des polygones convexes dont les faces ne peuvent intersecter une région sauf à la contenir toute entière. □

La notion de région montre bien que l'on peut accepter des gardes de la forme $x \sim y + c$.

Corollaire 8.11 *L'équivalence de comportement est d'index fini.*

Preuve: Il suffit de remarquer que le nombre d'états de l'automate des régions est fini. □

Notons que certaines formules définissent la région vide. Le nombre de régions non vides croît de manière exponentielle avec le nombre d'horloges : pour n horloges, et pour une plus grande constante M apparaissant dans les gardes et invariants, le nombre de régions est majoré par $n!(2M)^n$, donc exponentiel en le nombre d'horloges. De fait, la décision du vide d'un automate temporisé est $PSPACE$-complète. De même, pour chaque transition de l'automate temporisé étiquettée par la lettre a de l'alphabet des actions, il y a au plus $2M$ transitions étiquettées a au plus. Au total, la taille de l'automate des régions est donc exponentielle en le nombre d'horloges. C'est le problème majeur de la vérification dans le cas temporisé.

En fait, l'automate des régions n'est pas le quotient de l'automate de départ par l'équivalence de comportement, car la réciproque du lemme 8.10 n'est pas vraie. Il est possible d'agglomérer

certaines régions en *zônes* possédant la propriété de convexité grâce à un algorithme de minimisation qui teste l'équivalence de configurations appartenant à des régions différentes. La preuve du lemme 8.12 montre d'ailleurs que certaines régions sont inutiles.

La propriété de convexité des régions ou zônes a une importante conséquence : les calculs réussis de l'automate temporisés et de l'automate des régions se correspondent, de telle sorte que l'on peut projeter un calcul temporisé réussi en un calcul de régions et réciproquement reconstruire un calcul temporisé réussi à partir d'un calcul de régions. La réciproque n'est pas une question triviale, puisqu'il faudra construire des calculs qui ne satisfont pas le paradoxe de Zénon. Considérons l'automate temporisé représenté dans la partie gauche de la figure 8.7. L'automate des régions associé a 4 régions, définies par les formules $x = 0; x \in]0..1[; x = 1; x > 1$, et donc 4 états, il est représenté dans la partie droite de la meme figure.

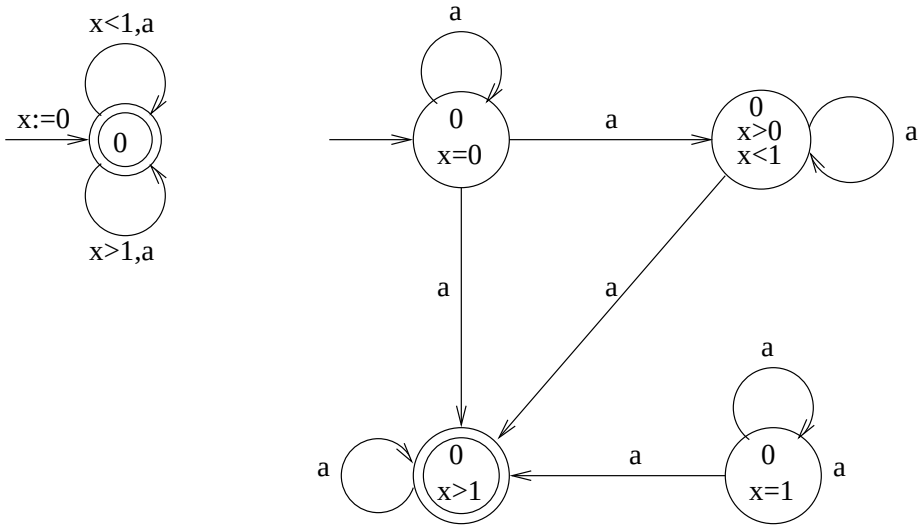


FIG. 8.7 – Automate temporisé et son automate des régions.

Considérons maintenant le mot a^ω qui est reconnu dans l'état $(0, x = 0)$. Le calcul correspondant de l'automate temporisé initial n'est bien sûr pas un calcul réussi puisque le temps n'a pas progressé. En fait, le seul état acceptant devrait être l'état $(0, x > 1)$, ce qui ne résoudrait bien sûr pas tous les problèmes.

Lemme 8.12 *Soit $\mathcal{A} = (V_t, Q, H, \mathcal{I}nv, F, T)$ un automate temporisé dont l'automate des régions est $\mathcal{A}_R = (V_t, Q_R, i_R, F_R, T_R)$.*

Si $(t_1, a_1)(t_2, a_2) \ldots (t_i, a_i)(t_{i+1}, a_{i+1}) \ldots \in \mathcal{L}^\omega(\mathcal{A})$ alors $a_1 a_2 \ldots a_i a_{i+1} \ldots \in \mathcal{L}^\omega(\mathcal{A}_R)$.

Réciproquement, si $a_1 a_2 \ldots a_i a_{i+1} \ldots \in \mathcal{L}^\omega(\mathcal{A}_R)$, alors il existe une suite de réels $t_1, t_2, \ldots, t_i, t_{i+1} \ldots$ tels que $(t_1, a_1)(t_2, a_2) \ldots (t_i, a_i)(t_{i+1}, a_{i+1}) \ldots \in \mathcal{L}^\omega(\mathcal{A})$.

Preuve: Soit $(q_0, H(q_0, 0)) \xrightarrow{\tau_0} (q_0, H(q_0, t_1)) \xrightarrow{t_1, a_1} (q_1, H(q_1, t_1)) \xrightarrow{\tau_1} (q_1, H(q_1, t_2)) \xrightarrow{t_2, a_2} (q_2, H(q_2, t_2)) \ldots (q_i, H(q_i, t_i)) \xrightarrow{\tau_i} (q_i, H(q_i, t_{i+1})) \xrightarrow{t_{i+1}, a_{i+1}} (q_{i+1}, H(q_{i+1}, t_{i+1}))$ un calcul réussi reconnaissant le mot $(t_1, a_1)(t_2, a_2) \ldots (t_i, a_i)(t_{i+1}, a_{i+1}) \ldots$. On montre par récurrence que pour tout i il existe une unique région α_i telle que $H(q_i, t_i) \models \alpha_i$ et que $(q_0, \alpha_0) \xrightarrow{a_1} (q_1, \alpha_1) \xrightarrow{a_2} (q_2, \alpha_2) \ldots (q_i, \alpha_i) \xrightarrow{a_i} (q_{i+1}, a_{i+1})$ est un calcul de $\mathcal{A}_R$. Il suffira pour conclure de noter qu'il n'y a qu'un nombre fini de régions de la forme (q, α), et donc tout calcul temporisé réussi de $\mathcal{A}$ passant infiniment souvent par un état global acceptant de la forme (q, t_k), le calcul ainsi construit de l'automate des régions passera une infinité de fois par un état de la forme (q, α).

À l'instant 0, l'automate des régions est dans la région initiale qui vérifie la propriété.

Supposons maintenant la propriété vraie à l'instant t_i. La valuation $H(q_i, t_i)$ appartient à l'unique région α_i qui contient l'invariant de l'état q_i et ordonne totalement les variables d'horloges pour $\leq$. Par définition de l'automate des régions, il existe une transition sortante de cet état vers l'état (q_{i+1}, α_{i+1}), où α_{i+1} est l'unique région que satisfait $H(q_{i+1}, t_{i+1})$ par le Lemme 8.9.

Réciproquement, étant donné un calcul acceptant quelconque $(q_0, \alpha_0) \xrightarrow{a_1} (q_1, \alpha_1) \xrightarrow{a_2} (q_2, \alpha_2) \ldots (q_i, \alpha_i) \xrightarrow{a_i} (q_{i+1}, a_{i+1})$ de l'automate des régions, on construit tout d'abord un (non unique) calcul temporisé acceptant $(q_0, H(q_0, 0)) \xrightarrow{\tau_0} (q_0, H(q_0, t_1)) \xrightarrow{t_1, a_1} (q_1, H(q_1, t_1)) \xrightarrow{\tau_1} (q_1, H(q_1, t_2)) \xrightarrow{t_2, a_2} (q_2, H(q_2, t_2))$ $\ldots (q_i, H(q_i, t_i)) \xrightarrow{\tau_i} (q_i, H(q_i, t_{i+1})) \xrightarrow{t_{i+1}, a_{i+1}} (q_{i+1}, H(q_{i+1}, t_{i+1}))$ par récurrence sur i tel que $H(q_i, t_i)$ appartient à la région α_i. L'idée est de faire progresser le temps de quantités constantes dans deux situations : lorsque l'on est sur un cycle qui fait progresser le temps ; lorsque l'on est dans une région ouverte. Dans tous les autres cas, on gardera si possible le temps constant.

À l'instant 0, la valuation initiale $H(q_0, 0)$ vérifie la propriété.

Supposons la propriété vérifiée pour i. t_i étant donné, si $\alpha_i \neq \alpha_{i+1}$, on choisira t_{i+1} minimal dans α_{i+1}. Si au contraire $\alpha_i = \alpha_{i+1}$, alors on choisira $t_{i+1} = t_i$, à moins que α_i soit une région ouverte, auquel on fera progresser le temps d'une valeur constante.

Une fois cette première étape éffectuée, on recalcule un nouveau mot temporisé en faisant progresser le temps dans les cycles de progression du temps si on ne termine pas en région ouverte.

Les états acceptants de l'automate des régions étant définis de façon à vérifier l'axiome de progression du temps, le calcul obtenu est accepté par l'automate temporisé. □

L'automate temporisé et son automate des régions associé montrés à la figure 8.8 illustrent le raisonnement.

8.4 Décision du vide des automates temporisés

La forme des contraintes temporelles et cruciale pour cette question : le vide devient indécidable si l'on permet des comparaisons de la forme $h_i < h_j + c$ où c est une constante rationnelle.

Décider le vide d'un automate temporisé va se faire grâce au lemme 8.12 en testant le vide de l'automate des régions. On obtient donc :

Théorème 8.13 *Le problème du vide des automates de Büchi temporisés est $PSPACE$-complet.*

Preuve: Il reste à prouver la propriété de complexité. □

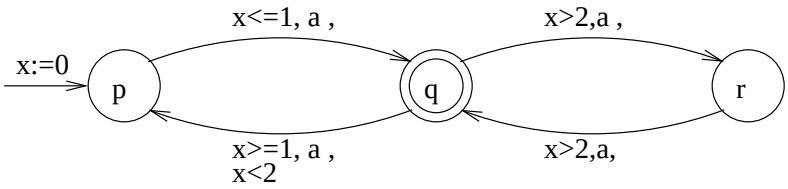
8.5 Intersection et complémentation des automates temporisés

Théorème 8.14 *Les automates de Büchi temporisés sont clos par intersection.*

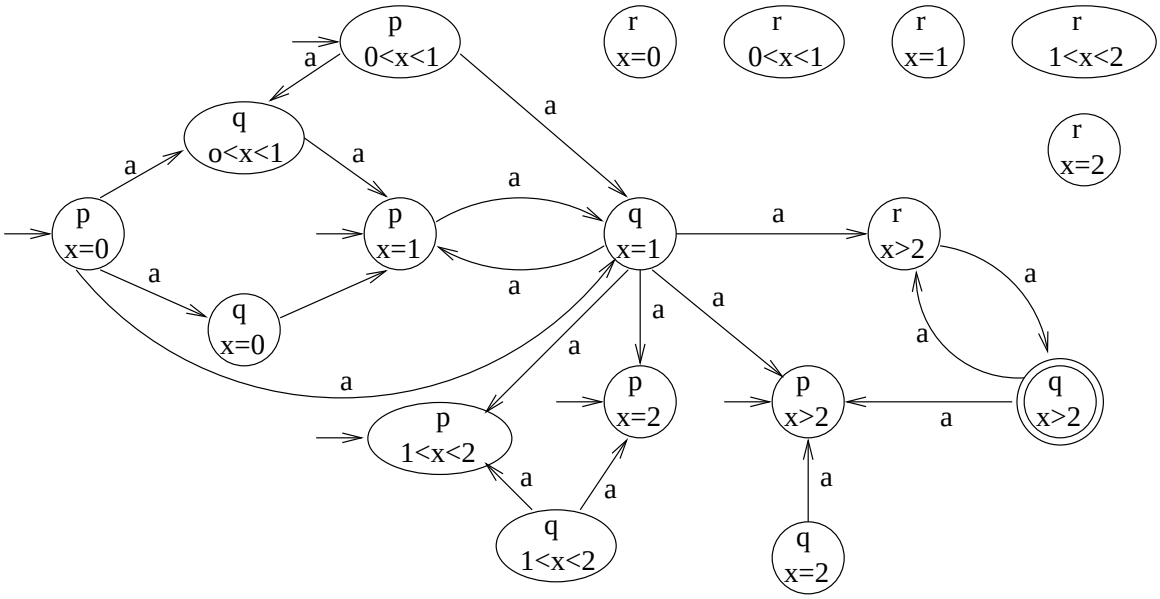
Preuve: La preuve se déduit sans difficulté de la preuve habituelle pour les automates de Büchi. □

Les automates de Büchi temporisés ne sont malheureusement pas clos par complémentation. Un exemple d'automate de Büchi temporisé qui reconnait un langage dont le complémentaire n'est pas reconnaissable par un automate de Büchi temporisé est donné à la figure 8.9.

En effet, cet automate de Büchi temporisé teste si dans une suite infinie de a, la lecture de deux d'entre eux est séparée par un intervalle de temps égal à un. Pour le complémentaire, il faudrait s'assurer qu'il n'y a pas deux transitions séparées par un intervalle de temps égal à un, ce qui nécessiterait de mémoriser tous les intervalles de temps, et donc demanderait un nombre infini d'horloges. Pour montrer que ce n'est pas possible, la solution la plus simple consiste à appliquer une technique de pompage en utilisant pour ce faire l'automate $\mathcal{A}_R$ des régions de l'automate $\mathcal{A}$ supposé reconnaitre le langage complémentaire.



Automate temporisé



Automate des régions

FIG. 8.8 – Automate temporisé et son automate des régions.

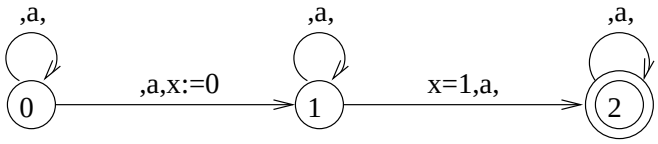


FIG. 8.9 – Automate de Bûchi temporisé non complémentable.

Soit $(q_0, V_0) \xrightarrow{t_1,a} (q_1, V_1) \xrightarrow{t_2,a} q_2 \ldots (q_i, V_i) \xrightarrow{t_{i+1},a} (q_{i+1}, V_{i+1}) \ldots (q_N, V_N)$ un calcul réussi qui lit un mot temporisé dont le nombre N de a est supérieur au nombre d'états de $\mathcal{A}_R$ pendant la première unité de temps. Alors, il existe $i < j$ tels que $q_i = q_j$, $tj < 1$, et V_i et V_j satisfont la

conjonction de contraintes atomiques de la région. On construit un nouveau début de mot temporisé en procédant comme suit : la suite est la même jusqu'à l'instant t_i, puis on continue avec la partie de suite commençant à l'instant t_j, les temps étant recalés à partir de t_i, dans laquelle on insère si nécessaire un a à l'instant t_j. On vérifie aisément que cette suite appartient au langage complémentaire, et elle est donc reconnue. De plus la construction assure que cette reconnaissance permet de passer par l'état q_i à l'instant t_i :

$$(q_0, V_0) \xrightarrow{t_1, a} (q_1, V_1) \xrightarrow{t_2, a} q_2 \ldots (q_i, V_i) \xrightarrow{t_{i+1}, a} \ldots (q_j, V_j) \xrightarrow{t_{j+1} - t_i, a} \ldots (q_k, t_j) \ldots$$

On peut maintenant réinsèrer le calcul allant de q_i à q_j en décalant à nouveau les instants des transitions pour construire un calcul reconnaissant un mot temporisé qui ne fait pas partie du langage.

Toutefois, il est possible d'obtenir un résultat de complémentation plus faible :

Théorème 8.15 *Le langage complémentaire d'un automate de Büchi temporisé déterministe est reconnu par un automate de Büchi temporisé (non déterministe).*

Preuve: Il est clair que la propriété usuelle d'unicité des calculs est encore vraie pour un automate de Büchi temporisé déterministe. Il suffit alors d'utiliser la preuve faite pour les automates de Büchi déterministes. □

Une conséquence importante de ce résultat est la suivante :

Théorème 8.16 *Soient $\mathcal{A}$ un automate de Büchi temporisé et $\mathcal{A}'$ un automate de Büchi temporisé déterministe. Alors, le problème $\mathcal{L}ang(\mathcal{A}) \subseteq \mathcal{L}ang(\mathcal{A}')$ est décidable et plus précisément $PSPACE$-complet.*

Il suffit de construire l'automate qui reconnait le complément de $\mathcal{L}ang(\mathcal{A}')$, d'en faire l'intersection avec $\mathcal{L}ang(\mathcal{A})$, puis de tester le vide de l'automate obtenu. Ces arguments nous suggèrent qu'il est possible d'étendre les techniques de vérification aux automates temporisés.

8.6 La logique temporelle temporisée TCTL*

À la différence de la logique temporelle, la logique temporelle temporisée construit ses énoncés en faisant référence au temps. Les résultats de clôture des automates temporisés nous montrent que le langage qui valide une formule temporisée donnée devra être reconnu par un automate de Büchi temporisé déterministe. Il y a essentiellement deux possibilités : prendre comme énoncés atomiques des comparaisons sur les valeurs des horloges, comme par exemple $AG(h < 5)$ pour énoncer que l'horloge h a une valeur inférieure à 5 le long de toute calcul ; indexer les connecteurs temporels par des contraintes de temps, comme par exemple $pU_{<5}q$ pour énoncer que p est vraie le long du calcul courant jusqu'à ce que q prenne le relai, ce qui doit être le cas en moins de 5 unités de temps. C'est ce second choix, plus simple, que nous allons faire.

8.6.1 Syntaxe

Dans la seconde des deux variantes précitées, la syntaxe des formules est la suivante :

1. les propriétés atomiques sont celles de $\mathcal{P}rop$, en supposant donné un automate avec propriétés dans les états au lieu de transitions étiquettées par un alphabet ;
2. les combinateurs logiques sont ceux de la logique propositionnelle : Si ϕ, ϕ_1 et ϕ_2 sont des formules, alors $\neg\phi$, $\phi_1 \wedge \phi_2$ et $\phi_1 \vee \phi_2$ sont des formules ;
3. $F_{\sim k}\phi$ et $G_{\sim k}\phi$ sont des formules temporisées si cela est vrai de ϕ ;
4. $\phi_1 U_{\sim k}\phi_2$ est une formule temporisée si cela est vrai de ϕ_1 et ϕ_2 ;

où $\sim$ est un opérateur de comparaison arithmétique appartenant à l'ensemble $\{<, \leq, =, \geq, >\}$, et k prend ses valeurs dans un ensemble de valeurs numériques (par exemple entières, rationnelles, ou réelles). On peut noter la disparition de l'opérateur X, ce qui se justifie par une hypothèse que nous faisons au paragraphe qui suit sur la sémantique des expressions.

8.6.2 Sémantique

La sémantique des formules de TCTL* se définit de façon analogue à précédemment, les modèles d'une formule temporelle étant maintenant des automates non-déterministes temporisés avec des propriétés dans les états, ce qui nous permettra d'écrire $p \in q$ si la propriété p appartient à l'état q. Nous devrons bien sûr faire un choix pour l'expression du temps, et nous choisirons un temps dense modélisé par l'ensemble $\mathcal{Q}$ des rationels. On écrira

$$\mathcal{A}, u, t \models \phi$$

pour exprimer la validité de la formule ϕ à l'instant $t \in \mathcal{Q}$ du calcul u de l'automate $A = (V, Q, q_0, T, \mathcal{P}rop, l)$. En pratique, $\mathcal{A}$ est souvent omis. La grande différence avec la situation antérieure est que le temps étant maintenant rationnel, les calculs deviennent des fonctions constantes par morceaux avec sauts multiples de $\mathcal{Q}$ dans les états de Q, appellées *trajectoires* de l'automate. Une trajectoire est donc une fonction de $\mathcal{Q} \times \mathcal{R}at \times \mathbb{N}$ dans Q comme représenté à la figure 8.4.

On ne considérera que des exécutions infinies pour simplifier la définition de la sémantique, qui est donnée à la figure 8.10. On fera deux hypothèses fondamentales :

- On supposera qu'il n'y a jamais deux transitions d"états successives, c'est-à-dire non séparées par une transition de temps ; cette hypothèse vise simplement à simplifier nos définitions sans poser de problèmes pour la pratique. Un calcul est alors simplement une fonction de $\mathcal{Q} \times \mathcal{R}at$ dans Q constante par morceaux, sans sauts. Le cas général est en effet rendu complexe par le fait qu'il combine les transitions temporelles avec les suites de transitions d'état comme dans les automates non temporisés ; il fait l'objet d'un exercice.
- On supposera qu'il n'y a qu'un nombre fini de transitions d'états dans un temps fini. Cette hypothèse est fondamentale : elle sert à éviter le paradoxe de Zénon, pour lequel Achille ne rattrappe jamais la tortue partie avant lui.

u,t	$\models$	p	ssi	$p \in u(t)$
u,t	$\models$	$\neg\phi$	ssi	$u,t \not\models \phi$
u,t	$\models$	$\phi \wedge \psi$	ssi	$u,t \models \phi$ et $u,t \models \psi$
u,t	$\models$	$\phi \vee \psi$	ssi	$u,t \models \phi$ ou $u,t \models \phi$
u,t	$\models$	$F_{\sim k}\phi$	ssi	$\exists t'\ t' \sim t+k \wedge t' \geq t$ tel que $u,t' \models \phi$
u,t	$\models$	$G_{\sim k}\phi$	ssi	$\forall t'\ t' \sim t+k \wedge t' \geq t$ alors $u,t' \models \phi$
u,t'	$\models$	$\phi U_{\sim k}\psi$	ssi	$\exists t'\ t' \sim t+k \wedge t' \geq t$ tel que $u,t \models \psi$ et $\forall t'' \in [t..t']\ u,t'' \models \phi$
u,t	$\models$	$E\ \phi$	ssi	$\exists v$ prolongeant u tel que $v,t \models \phi$
u,t	$\models$	$A\ \phi$	ssi	$\forall v$ prolongeant u alors $v,t \models \phi$
A	$\models$	ϕ	ssi	pour toute exécution u de A, $u,0 \models \phi$

FIG. 8.10 – Sémantique des formules temporelles temporisées.

On peut bien sûr se restreindre à TCTL et TLTL, les logiques CTL temporisée et LTL temporisée. En pratique, $TCTL$ s'avère un bon compromis.

Théorème 8.17 *Pour toute formule ϕ de $TLTL$, il existe un automate de Büchi temporisé déterministe qui reconnait les mots qui valident ϕ.*

8.7 Vérification

Nous allons maintenant montrer comment généraliser les méthodes précédentes aux logiques temporisées.

Théorème 8.18 *L'automate des régions et l'automate de départ satisfont les mêmes formules de TCTL construites sur le langage des propriétés de la forme $x_i \sim c_j$, où $\sim \in \{=, \neq, >, <, \geq, \leq\}$ et $c_i \leq max$.*

Par contre, ce n'est pas vrai pour la logique $TLTL$ qui n'est en fait pas décidable, à moins de la restreindre en interdisant les comparaisons d'horloges à des valeurs exactes, ce que l'on appelle la *ponctualité*.

8.8 Exercices

Exercice 8.1 *Donner une définition des automates temporisés complets et un algorithme qui complète un automate temporisé quelconque sans en détruire le déterminisme le cas échéant.*

Exercice 8.2 *On se propose de modéliser une montre pourvue d'un affichage digital*

- de la date, jour, mois, année ;

- de l'heure, des minutes et secondes accompagnées des symboles AM *ou* PM *;*

- d'un chonomètre qui affiche la durée en secondes depuis son déclenchement ;

-d'une fonction réveil, qui déclenche un signal sonore 2 fois successivement à 30 secondes d'intervalle lorsque l'heure de réveil est atteinte.

Il se peut que la description ci-dessus soit imprécise, auquel cas il faudra la préciser de manière adéquate.

Exercice 8.3 *On se propose de modéliser un train franchissant une route avec passage à niveau et feux de signalisation. La barrière est soit ouverte, auquel cas le feu est vert, soit fermée, auquel cas le feu est rouge. L'approche d'un train déclenche la fermeture de la barrière et le passage du feu au rouge. Le délai entre l'entrée en zône d'approche et le franchissement de la barrière doit être d'au moins 30 secondes. La fermeture de la barrière ne doit pas prendre plus de 10 secondes. La barrière doit rester fermée, et le feu au rouge, au moins 10 secondes après le passage du train.*

Modéliser ce système sous la forme de trois (en comptant le feu bicolore) automates temporisés communiquants. On fera tout particulièrement attention lors de la modélisation de la barrière à ce que l'horloge associée soit initialisée de manière adéquate.

Exercice 8.4 *On considère un automate temporisé pour lequel le temps varie dans un intervalle $[0..n]$ pour un certain n donné. En s'inspirant des constructions analogues faite dans le cours, montrer que cet automate est équivalent à un automate fini en ce sens qu'ils ont les même calculs.*

Cela est-il encore vrai si le temps varie dans l'ensemble des entiers naturels ? dans l'ensemble des rationnels ? dans l'ensemble des réels ? Peut-on donner une notion d'équivalence plus grossière qui permette de construire un automate fini équivalent dans tous les cas ? cela dépend-il du langage des gardes ?

Exercice 8.5 *On se propose de modéliser un distributeur de bonbons qui accepte des euro-pièces de $1, 2, 5, 10$ et 20 centimes. La machine distribue $(n + n^2)/4$ bonbons pour une somme de n centimes plafonnée à 20 centimes. Les bonbons sont délivrés (s'il y a lieu) dès que l'utilisateur a attendu plus de 5 secondes avant d'introduire une nouvelle pièce. Au cas où il introduirait plus de 20 centimes, la somme excédentaire est rendue automatiquement lors de la délivrance des bonbons. Un nouvel utilisateur ne peut introduire de pièce tant que le précédent n'est pas servi.*

Donner les automates temporisés communiquants qui modèlisent le comportement de cette machine.

Exercice 8.6 *Avec les défibitions du cours, montrer que TCTL* est un raffinement de CTL* .*

Exercice 8.7 *On se propose maintenant de définir la sémantique des formules de logique temporisée CTL* dans le cas général où un calcul de l'automate est une fonction partielle $u(t,i)$ de $\mathcal{Q} \times \mathbb{N}$ dans Q, où t figure le temps qui passe, et i le nombre de transitions d'états éffectuées à un instant donné. Un calcul u est tel que, si $u(t,m)$ et $u(t',n)$ pour $t' \geq t$ sont définis, alors $m \geq n$ et $\forall i \in [m..n]\ \exists t'' \in [t..t']\ \exists q \in \mathcal{Q}$ tel que $u(t'',i) = q$. On notera par $u,t,i \models \phi$ la validité de la formule ϕ pour la trajectoire u.*

1. *Donner la sémantique des formules de CTL* avec cette sémantique générale.*
2. *Nous supposons maintenant disposer d'un combinateur X dans la syntaxe qui nous sert comme précédemment à parcourir les changements successifs d'états. Donner la sémantique de ce combinateur.*
3. *A-t-on besoin du combinateur X ?*

Exercice 8.8 *On se propose de calculer l'automate des régions de l'automate temporisé représenté à la figure 8.11.*

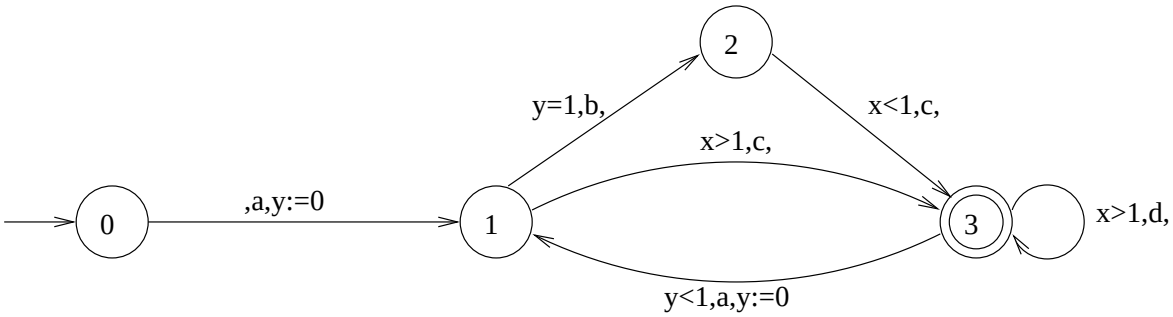


FIG. 8.11 – Automate temporisé.

Chapitre 9

Propriétés temporelles à tester

Bioen que cela soit possible, nous ne donnerons pas de définition précise des différents types de propriété que nous allons étudier, nous contentant de définitions très informelles.

9.1 Atteignabilité

Une propriété d'atteignabilité énonce qu'une certaine situation peut être atteinte. Ces propriétés sont faciles à exprimer en CTL. LTL, par contre, n'est pas adapté à l'expression de ces propriétés (il faut les exprimer négativement pour la pluspart), et ne peut pas les exprimer toutes. Donnons-en quelques exemples :

1. la variable n peut prendre la valeur zéro : $EF(n = 0)$;
2. la variable n peut ne pas prendre la valeur zéro : $EF(n \neq 0)$;
3. il existe un calcul pour lequel la variable n peut en permanence prendre la valeur zéro : $EGEF(n = 0)$;
4. la variable n peut en permanence prendre la valeur zéro : $AGEF(n = 0)$;
5. n peut prendre la valeur zéro sans que m le fasse : $E(m \neq 0)U(n = 0)$;

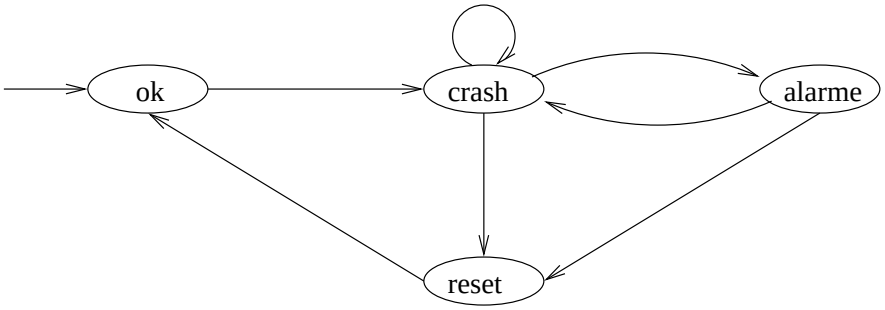
L'atteignabilté est donc associée au combinateur EF, et leur emboitement fait intervenir les combinateurs EG ou AG à l'extérieur. Les propriétés d'atteignabilté emboitée ne sont pas expressibles en LTL.

Ces propriétés sont les plus simples à vérifier, elles reposent sur le calcul du graphe d'atteignabilité, qui est l'outil de base des vérificateurs. Comme l'ensemble des états atteignables est défini comme le point fixe d'une fonctionnelle croissante sur le treillis des parties de l'ensemble des états, le calcul peut se faire de deux manières : vers l'avant, ou vers l'arrière. Le calcul vers l'arrière est en général plus compliqué si l'automate possède des variables, mais il est aussi plus dirigé, et les deux méthodes sont en général combinées. Dans le cas d'automates synchronisés, le graphe produit est généralement engendré à la volée, au fur et à mesure des besoins du calcul, de manière à éviter l'explosion combinatoire inhérente à la définition du produit synchronisé.

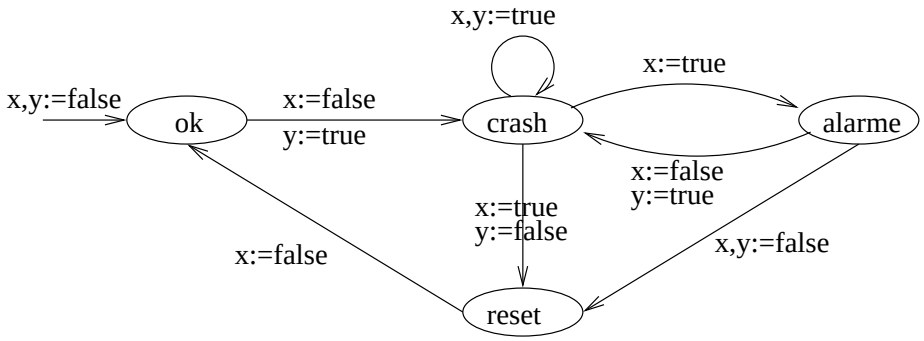
9.2 Sureté

Une propriété de sureté énonce qu'une certaine situation ne peut pas se produire. Ces propriétés s'expriment en CTL avec le combinateur AG et en LTL avec le combinateur G (le quantificateur de chemin externe A étant implicite). Donnons-en quelques exemples :

1. la variable n ne passe jamais par la valeur zéro : $AG(n \neq 0)$;



Modèle d'alarme



Modèle d'alarme enrichi de variables d'histoire

FIG. 9.1 – Utilisation de variables d'histoire.

2. tant que la variable x ne passe pas par la valeur zéro alors la variable n non plus : $A(n \neq 0)W(x = 0)$;

Notons l'importance ici du combinateur W. La formule $A(n \neq 0)U(x = 0)$ ne traduit pas la propriété voulue. En fait, elle n'est pas classée dans les propriétés de sureté.

On voit que la sureté est essentiellement une propriété de non atteignabilité (puisque $AG\phi$ équivaut à $\neg EF\neg\phi$. On peut donc utiliser le calcul des états atteignables, mais il faut bien sûr aller jusqu'au bout du calcul, ce qui n'est pas le cas pour les formules d'atteignabilité.

En fait, on veut aussi pouvoir vérifier des propriétés de sureté dont l'expression fait référence au passé ou au futur. Donnons en un exemple lié au fonctionnement d'une alarme modélisée à la figure 9.1 :

1. l'alarme ne sonne que s'il y a eu un crash précédement : $AG(alarme \Longrightarrow F^{-1}crash$;
2. chaque crash est suivi d'une activation de l'alarme sans passer par l'état reset $AG(alarme \Longrightarrow \neg(resetSincecrash)$;

On peut bien sûr exprimer la formule $AG(alarme \Longrightarrow F^{-1}crash$ en logique temporelle, sans utiliser le combinateur additionnel F^{-1}, le lecteur en fera l'exercice. La seconde peut bien sûr elle-aussi s'exprimer el CTL* , par exemple sous la forme $AG(crash \Longrightarrow \neg(resetUalarme))$.

Notre but maintenant est de montrer comment vérifier de telles formules en se ramenant à un problème d'atteignabilité implémentable sur un outil ordinaire. L'idée est de décorer le graphe avec

de nouvelles variables qui vont servir à mémoriser les propriétés du passé, da ns le cas présent les propriétés $F^{-1}crash$ et $\neg(resetUalarme))$.

La méthode des variables d'histoire est générale : on peut traiter les combinateurs du passé de cete manière, à condition d'introduire une variable d'histoire pour chaque occurrence d'une sous-formule possédant à sa racine un combinateur du passé. Cette technique d'introduction de nouvelles variables peut bien sûr aégalement être utilisée pour transformer une formule qui n'est pas une formule de CTL en une autre qui l'est.

9.3 Vivacité

Une propriété de vivacité énonce qu'une certaine situation finira par se produire.

1. la variable n passe forcément un jour par la valeur zéro : $AF(n = 0)$;

C'est le combinateur F qui caractérise le propriétés de vivacité.

1. Toute requêête sera satisfaite un jour : $G(requte \implies Fsatisfaite)$ en PLTL et $AG(requte \implies AFsatisfaite)$ en CTL ;
2. Le système peut toujours retourner à son état initial : $AG(EFinit)$ en CTL. En PLTL, la formulation de cette propriété demande de pouvoir caractériser explicitement l'ensemble des états vérifiant $EFinit$ à cause de l'emboitement de quantificateurs de chemins.

Notons que la propriété pUq est une combinaison de propriété de sureté (p tout le temps vraie jusqu'à ce que un certain évènement ait lieu) et de vivacité (q aura certainement lieu).

La vivacité est souvent présente dans le modèle lui-même. Par exemple, on suppose implicitement qu'un automate ne s'arrête pas de fonctionner sans raison : la prise en compte des actions (c'est-à-dire la lecture) est vivace. La vivacité implicite des modèles peut être source de confusions et d'erreurs. Par exemple, un produit de deux automates induira une hypothèse implicite de vivacité sur l'automate produit, mais pas sur ses composants : on pourra en général faire des calculs dans l'un des 2 automates à lk'exclusion de l'autre.

9.4 Équité

Une propriété d'équité énonce qu'une certaine situation se produira infiniment souvent. C'est une propriété de vivacité perpétuelle.

1. La barrière sera levée infiniment souvent : $\overset{\infty}{F}\, barr_leve$ en LTL ;
2. Si l'accès en section critique est demandé un nombre infini de fois, alors il sera accordé un nombre infini de fois : $\overset{\infty}{F}\, requte_section_critique \Longrightarrow \overset{\infty}{F}\, en_section_critique$ en LTL.

L'expression de la vivacité est possible en CTL dans certains cas, car $\overset{\infty}{F}\ \phi$ est équivalent à $AGAF\phi$, qui est une formule de CTL s'il n'y a pas de connecteurs temporels dans ϕ.

Les formules de la forme $E\ \overset{\infty}{F}\ \phi$ qui expriment que ϕ est vraie infiniment souvent au cours d'un certain calcul n'est pas exprimable en CTL, malgré son importance pratique (relative, toutefois).

9.5 Blocage

La propriété d'absence de blocage énonce que le système ne peut jamais se trouver dans une situation où il lui est impossible de progresser. Elle s'exprime sous la forme $AGEXtrue$. C'est donc une propriété CTL. En LTL, on essaiera de l'exprimer en fonction de l'automate à valider.

On range souvent l'absence de blocage parmi les propriétés de sureté, parce aque, pour un automate donné, il est généralement possible de décrire explicitement les états de blocage : il suffit alors d'exprimer leur non-atteignabilité, ce qui est une propriété de sureté.

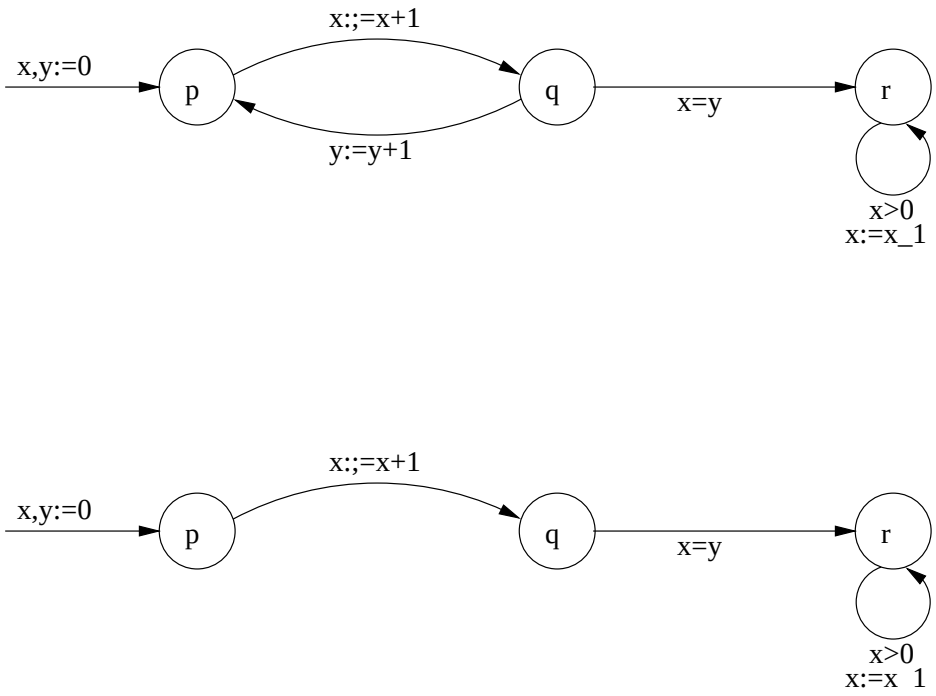


FIG. 9.2 – Deux automates avec états blocants.

9.6 Conclusion

La vérification obéit donc à une certaine méthodologie, qui se reflète dans les outils disponibles : la pluspart se concentre sur les propriétés d'atteignabilité et de non-atteignabilité, et l'utilisateur est prié de tout ramener à des propriétés de cette forme, au prix parfois de quelques acrobaties.

9.7 Exercices

Exercice 9.1 *1. Énoncer l'absence de blocage sous forme d'une propriété de sureté pour le premier automate de la figure 9.2.*

2. *La propriété est-elle vérifiée ?*
3. *Le second automate satisfait-il cette propriété ?*
4. *Énoncer l'absence de blocage sous forme d'une propriété de sureté pour le second automate de la figure 9.2.*
5. *La propriété est-elle vérifiée ?*
6. *Que concluez vous ?*

Chapitre 10

Abstractions

En pratique, l'automate à vérifier est souvent trop gros pour les techniques actuelles de vérification. On s'en tire alors en le simplifiant de manière à ce que l'automate de départ et l'automate simplifié vérifient les mêmes formules, ou, de manière plus réaliste, les mêmes formules à vérifier (on ne se souciera pas des propriétés qui ne nous intéressent pas).

Chapitre 11

Outils logiciels

Notre but dans ce cours est de présenter une même modélisation particulière dans chacun des quatre systèmes rapidement décrits ci-dessous, et de vérifier des propriétés de cette modélisation, de manière à avoir des points de comparaison de ces systèmes.

11.1 CHRONOS

CHRONOS, disponible à l'URL `http : //www-verimag.imag.fr/TEMPORISE/chronos` est développé par une équipe animée par Serge Yovine au laboratoire VERIMAG dirigé par Joseph Sifakis. Il permet de spécifier un réseau d'automates temporisés, puis de vérifier des propriétés de TCTL sur cet automate. C'est l'un des rares outils permettant de spécifier et vérifier des propriétés temporelles qui ne se ramènent pas à une propriété d'atteignabilité, comme la vivacité.

11.2 HYTECH

HyTech, disponible à l'URL `http : //www.eecs.berkeley.edu/~tah/Hytech` est développé par une équipe dirigée par Tom Henzinger à l'université de Berkeley. HyTech permet de spécifier des réseaux synchronisés de systèmes hybrides linéaires, pour lesquels la loi d'évolution de certaines variables réelles est régie par une équation différentielle linéaire. Il est le seul outil permettant aujourd'hui d'analyser des systèmes paramétrés aussi bien qu'hybrides. Les analyses sont limitées à des propriétés d'atteignabilité.

11.3 UPPAAL

UPPAAL, disponible à l'URL `http : //www.docs.uu.se/docs/rtmv/uppaal` est développé par P. Pettersson (université d'Uppsala en Suède) et Kim Larsen (université d'Alborg au Danemark), d'où son nom. Il permet de spécifier un réseau d'automates temporisés synchronisés par messages, et en vérifier des propriétés d'atteignabilité. Son interface est de bonne qualité, comportant des outils graphiques.

11.4 CMC

CMC, disponible à l'URL `http : //www.lsv.ens-cachan.fr/~{}fl` est développé par François Larroussinie (ENS-Cachan). Il a pour but d'utiliser la structuration des spécification en un réseau d'automates temporisés communiquants pour éviter l'explosion du logiciel.

Chapitre 12

Études de cas

Chapitre 13

Sujets de devoir

Exercice 13.1 *On se propose de modéliser l'intersection d'une route et d'une voie ferrée avec passage à niveau et feux de signalisation. La barrière est soit ouverte, auquel cas le feu est vert, soit fermée, auquel cas le feu est rouge. L'approche d'un train déclenche d'une part la fermeture de la barrière et le passage du feu au rouge, et d'autre part l'activation d'un sémaphore qui interdit (visuellement) à un second train circulant dans le même sens d'entrer en zone d'approche. Le délai entre l'entrée en zône d'approche et le franchissement de la barrière doit être d'au moins 30 secondes. La fermeture de la barrière ne doit pas prendre plus de 10 secondes. La barrière doit rester fermée, et le feu au rouge, au moins 10 secondes après le passage du train. On supposera qu'il y a deux trains circulant dans chaque sens et que deux trains circulant en sens inverse peuvent se croiser en zône d'approche.*

1. *Modéliser ce système sous la forme d'automates temporisés communiquants.*
2. *Vérifier la sureté de ce système dans le système UPPAAL.*
3. *Peut-on racourcir le délai d'approche sans compromettre la sureté du système ? Si oui, quel est le délai à partir duquel la sureté est compromise ?*
4. *Comment pourrait-on gérer trois trains circulant dans le même sens ?*

Exercice 13.2 *On se propose de modéliser le fonctionnement d'une machine à café, garantie conforme à la norme AFNOR JB007, qui indique la nécessité pour les machines à café électriques à usage privé de se mettre hors circuit en cas de réservoir vide. La machine peut délivrer du café ou du thé suivant les types de capsule et de tasse insérées dans les logements prévus à cet effet. Un bouton poussoir permet de sélectionner thé ou café et déclenche le fonctionnement de la machine en même temps. La délivrance d'un café prend 20 secondes et réclame 10 cl d'eau, et celle d'un thé prend 1 minute et réclame 20 cl d'eau, quantités qui tiennent compte de l'évaporation causée par le chauffage. Un réservoir de 2 litres permet d'alimenter manuellement la machine en eau. La machine ne peut être utilisée par deux personnes en même temps, le bouton poussoir étant inhibé pendant tout le temps de service d'un utilisateur. La machine se met d'elle-même en veille si elle reste inutilisée plus de 5 minutes, auquel cas elle a besoin d'un temps de chauffe de 90 secondes avant de pouvoir commencer à fonctionner à nouveau. Ce préchauffage est déclenché lorsque le nouvel utilisateur presse le bouton poussoir, et occasionne une perte d'eau par évaporation de 5 cl. Le fonctionnement de la machine est matérialisé par un voyant lumineux qui change de couleur suivant qu'elle est : en veille, en manque d'eau, prête à fonctionner, ou en service.*

1. *Modéliser ce système sous la forme d'automates temporisés communiquants.*
2. *Vérifier la conformité à la norme AFNOR JB007 dans le système UPPAAL.*

Exercice 13.3 *On se propose de modéliser le fonctionnement d'une montre digitale alimentée par une pile et pourvue de boutons poussoirs. La montre a plusieurs modes de fonctionnement, lecture,*

alarme et chronomètre. Le changement de mode se fait par pression sur un premier bouton poussoir. En cas d'absence de pression pendant au moins une minute sur l'ensemble des boutons poussoirs, la montre revient automatiquement en mode lecture. En mode lecture, la montre affiche la date et l'heure sur l'écran, sous la forme $x\ y\quad z : t$*, où* x *est le numéro du jour dans le mois,* y *est le numéro du mois,* z *est l'heure (variant de 0 à 24) et* t *le nombre de minutes dans l'heure. En mode alarme, elle indique dans le même format la date de la prochaine alarme. Enfin, en mode chronomètre, la montre affiche le temps écoulé depuis le déclenchement du chronomètre dans le format* $x : y : z$*, où* x *est le nombre d'heures,* y *le nombre de minutes et* z *le nombre de secondes. Le chronomètre se remet automatiquement à zéro au bout de 24 heures.*

En mode lecture, une pression longue (au moins 3 secondes) du second bouton poussoir permet de faire clignoter tout d'abord le jour, puis, après de nouvelles pressions longues, le mois, l'heure et enfin les minutes de l'affichage sur l'écran. Chacune de ces données peut alors etre modifiée par une succession de pressions courtes (moins d'une seconde) permettant de faire défiler les valeurs possibles. Les mises à jour sont effectives à la prochaine pression longue, ou en l'absence de pression sur le bouton poussoir pendant au moins 10 secondes.

En mode alarme, la même procédure permet cette fois de modifier la date de la prochaine alarme.

En mode chronomètre, l'écran affiche le temps écoulé depuis la dernière mise en route du chonomètre. La remise à zéro du chronomètre s'obtient par une pression sur le second bouton poussoir.

La pile est garantie pour 1000 heures de fonctionnement. Au bout de 900 heures, un voyant lumineux s'affiche sur l'écran afin de prévenir l'utilisateur de la nécesité de changer de pile.

On demande de modéliser cette montre et d'en vérifier le bon fonctionnement dans le système UPPAAL.

Bibliographie

[1] Rajeev Alur and David L. Dill. *A Theory of Timed Automata*, Theoretical Computer Science 126(2) :183-235. Elsevier, 1994.

[2] Béatrice Bérard, Michel Bidoit, François Laroussinie, Antoine Petit et Philippe Schnoebelen *Vérification de logiciels : Techniques et Outils du model-checking*, 1999. Vuibert Informatique. Philippe Schnoebelen, coordinateur.

[3] Hubert Comon *Automates et logiques temporelles : vériufication de systèmes réactifs*, mars 2000. Notes de cours du DEA *Prograammation, Sémantique et Preuves.*

[4] John E. Hopcroft et Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*, 1979. Addison Wesley.

[5] Moshe Vardi. *An Automata-Theoretic Approach to Linear Temporal Logic* , in *Logics for Concurrency : structure versus automata*, LNCS 1043 :238-266. Springer, 1994.

[6] Moshe Vardi, Orna Kupferman et Pierre Wolper. *An Automata-Theoretic Approach to Branching-Time Model Checking* , full version of CAV'94. , .