Introduction au Model Checking
ENSTA

Sébastien Bardin
CEA,LIST, Laboratoire de Sûreté logicielle
Boîte 65, Gif-sur-Yvette, F-91191 France
sebastien.bardin@cea.fr

24 octobre 2008

Table des matières

1 Introduction **3**
1.1 Systèmes réactifs . . . 3
1.2 Propriétés temporelles . . . 3
1.3 Model checking . . . 4
1.4 Historique . . . 4
1.5 En pratique . . . 5
1.6 Lectures conseillées . . . 7

I Bases du model checking **8**

2 Modélisation des systèmes réactifs **9**
2.1 Syntaxe : machines à états . . . 9
2.2 Sémantique : systèmes de transitions . . . 9
2.3 Structure de Kripke . . . 11
2.4 Discussion sur la terminologie . . . 13
2.5 Espace des états, propriétés d'accessibilité et d'invariance . . . 13
2.6 Systèmes concurrents . . . 15
2.7 Hypothèses d'équité . . . 16
2.8 Propriétés de sûreté. . . . 17
2.9 Quelques points de modélisation . . . 18

3 Logiques temporelles **20**
3.1 Panorama de propriétés temporelles . . . 20
3.2 Intuitions sur les logiques temporelles . . . 21
3.3 Logique linéaire LTL . . . 25
3.4 Logique branchante CTL* . . . 27
3.5 Logique branchante CTL . . . 28
3.6 Comparaison des trois logiques . . . 29

4 Model checking, algorithmes de base **32**
4.1 Prélude : composantes fortement connexes . . . 32
4.2 Model checking de CTL par labelling . . . 33
4.3 Model checking de fair CTL par labelling . . . 35
4.4 Model checking de LTL par automates . . . 37

Références **39**

A Rappels de logique 41
A.1 Définitions . . . 41
A.2 Problèmes classiques liés aux logiques . . . 41
A.3 Quelques logiques . . . 42
A.4 Exemple : logique classique propositionnelle . . . 42

B Notions de calculabilité et complexité 44
B.1 Calculabilité . . . 44
B.2 Complexité . . . 45

C Divers problèmes algorithmiques 46
C.1 Composantes fortement connexes . . . 46

D Sujets de partiel 48
D.1 ENSTA, année 2006-2007 . . . 48
D.2 ENSTA, rattrapages, année 2006-2007 . . . 49

Chapitre 1

Introduction

1.1 Systèmes réactifs

Habituellement un programme, que nous dirons *standard*, (1) termine ; (2) retourne un résultat et (3) manipule des données complexes mais sa structure de contrôle est assez simple. Pour ces programmes standards, les propriétés à prouver sont toujours du style *"quand la fonction est appellée et que la précondition est vérifiée, alors la fonction termine et la postcondition est vérifiée"*.

Exemple typique de programme standard : compilateur, algorithme de tri.

Ici nous nous intéressons à une classe très particulière de programmes : les *systèmes réactifs*. Quelques propriétés remarquables de ces systèmes : (1) ils ne terminent pas forcément ; (2) ils ne calculent pas un résultat mais plutôt maintiennent une interaction ; (3) les types de données manipulés sont souvent simples alors que le contrôle est complexe (exécution de plusieurs composants en parallèle). Enfin bien souvent ils interagissent avec un environnement par le biais de capteurs (prise d'information) et d'actionneurs (action).

Exemples typiques de systèmes réactifs : systèmes embarqués pour les transports/l'énergie, systèmes d'exploitation, protocoles de communication, etc.

1.2 Propriétés temporelles

Les propriétés que l'on veut prouver sur ces systèmes réactifs sont très différentes de celles que l'on veut prouver sur des programmes standards. On veut typiquement prouver des propriétés sur l'entrelacement des évènements tout au long de l'exécution (infinie) du programme, par exemple

- si un processus demande infiniment souvent à être exécuté, alors l'OS finira par l'exécuter ;
- il est toujours possible lors de l'exécution de revenir à l'état initial ;
- chaque fois qu'une panne est détectée, une alarme est émise ;
- chaque fois qu'une alarme a été émise, une panne avait été détectée.

Schématiquement, pour les programmes standards, les propriétés à vérifier impliquent des prédicats très riches sur les données manipulées[1] mais les aspects temporel sont très restreints, tandis que pour les systèmes réactifs l'aspect temporel est très élaboré mais les prédicats sur les données sont souvent basiques[2].

Pour exprimer ces propriétés temporelles on utilise des logiques temporelles. À ça deux avantages. (1) Ces logiques peuvent être utilisé avantageusement lors de la phase de spécification, puisqu'elles décrivent les comportements temporels de manière non ambigue. (2) Un algorithme qui vérifie toutes les propriétés

[1] Par exemple : le tableau doît être trié.
[2] Par exemple : $x \neq 0$

exprimables dans une certaine logique est bien plus souple et générique qu'un algorithme dédié à un problème particulier.

Il existe de nombreuses logiques temporelles. Nous verrons principalement CTL, LTL et CTL*.

1.3 Model checking

Définition 1.3.1. Le model checking est un ensemble de techniques de vérification automatique de propriétés temporelles sur des systèmes réactifs.

Schématiquement, un algorithme de model checking prend en entrée une abstraction du comportement du système réactif (*un système de transitions*) et une formule d'une certaine logique temporelle, et répond si l'abstraction satisfait ou non la formule. On dit alors que le système de transitions est un modèle de la formule, d'ou le terme anglais de model checking.

Le gros avantage du model checking est qu'il est (idéalement) complètement automatique, et que habituellement un contre-exemple est retourné quand la propriété n'est pas vérifiée. Ce dernier point a été déterminant pour une adoption industrielle.

La limitation théorique principale des techniques standards de model checking est que *le système de transition doit être fini* : grossièrement, le programme ne doit manipuler que des variables à domaine fini. C'est souvent le cas en pratique, mais pas toujours. Soit que le système est infini par nature (ex : systèmes dépendants du temps et donc variables dans $\mathbb{R}$), ou que les bornes sont impossibles à estimer (ex : bornes des canaux de communication de l'internet), ou bien encore que le système dépend de paramètres (mémoire disponible, nombre de clients) et qu'il doit fonctionner pour n'importequ'elles valeurs de ces paramètres.

En pratique la limitation majeure du model checking est la taille gigantesque des systèmes de transitions due au *phénomène d'explosion combinatoire du nombre d'états du système*. Par exemple on compte déjà 10^{255} états pour un programme manipulant 10 variables codées sur 8 bits[3]. Le phénomène d'explosion combinatoire a deux sources distinctes : la taille du système de transitions augmente exponentiellement d'une part avec le nombre de variables (et leur taille), d'autre part avec le nombre de composants du systèmes dans le cas où le système est concurrent (très courant).

Des techniques spécifiques ont été développées pour limitée chacune de ces sources potentielles d'explosion, par exemple le model checking symbolique pour le nombre de variables et les ordres partiels pour les entrelacements de composants. Ceci et l'augmentation des ressources de calcul permet aujourd'hui l'utilisation industrielle du model checking pour certains types d'application.

La recherche en model checking continue et se partage actuellement en trois grandes tendances : continuer d'améliorer l'efficacité pratique des techniques de model checking fini pour combattre l'explosion combinatoire, étendre le model checking à des systèmes plus complexes (infinis, temporisés, probabilistes) et enfin idéalement adapter le model checking à des logiciels quelconques.

1.4 Historique

1970. À la fin des années 70, les techniques de vérification développées durant la décennie (types, logique de Hoare, analyse statique) se révèlent inadaptées pour les systèmes réactifs. En 1977, Pnueli est le premier a proposé d'utiliser les logiques temporelles[4] pour spécifier le comportement de systèmes réactifs. Puis en 1981

[3] À titre de comparaison un petit programme a vite une centaine de variables codées sur 32 bits, le nombre de particules dans l'univers est de l'ordre de 10^{80} et le nombre de secondes depuis le Big-Bang est de l'ordre de 10^{17}.

[4] Développées jusque là par des philosophes et des logiciens.

Clarke et Emerson aux États-Unis et Quielle et Sifakis en France développent indépendamment les premiers algorithmes de model checking. La logique utilisée est CTL.

1980. Durant les années 80, les résulats sont principalement d'ordre théorique : résulats de décision et bornes de complexité, comparaisons de différentes logiques, découverte des techniques de model checking par automates (*automata-theoretic model checking*, par Vardi et Wolper). Du point de vue pratique, quelques preuves de concept sont réalisées sur des exemples jouets, mais le phénomène d'explosion combinatoire (et les capacités de calcul de l'époque) rendent les cas d'études industriels hors de portée. Un certain scepticisme règne dans la communauté de la vérification automatique.

1990 Les années 90 voient émergées deux techniques majeures pour combattre le phénomène d'explosion combinatoire et améliorer considérablement les performances des model-checkers. Les techniques d'*ordres partiels* (Peled, Godefroid) permettent d'alléger la vérification en ne construisant qu'une partie du système de transitions. Les techniques de *model checking symbolique par BDD* (McMillan) permettent elles de représenter de manière très compacte le système de transitions. Ces deux techniques rendent possibles les premières études de cas de taille industrielle, avec des systèmes de transition allant jusqu'à 10^{20} états.

1995-20 ? ? Les recherches sur le model checking fini continuent, pour améliorer encore l'efficacité. Citons par exemple la *model checking modulaire*, le *raffinement automatique d'abstractions* et le *bounded model checking*. En parallèle de nombreux travaux visent à aller plus loin que le model checking fini : systèmes infinis, systèmes temporisés, systèmes probabilistes, etc. Enfin, au début des années 2000, les techniques de model checking commencent à être adaptées dans le but de vérifier des programmes classiques. On assiste à l'émergence du *software model checking* (Ball, Godefroid, Henzinger).

Influence académique. Avec le software model checking, on commence à assister au mariage maintes fois annoncé du model checking, de l'analyse statique, de la génération de tests et de la démonstration automatique. De ce fait, les techniques développées pour le model checking se diffusent dans d'autres domaines, notamment les logiques temporelles *(design by contract, run-time verification, model-based testing)*, les ordres partiels *(génération de tests de systèmes concurrents)*, et le raffinement automatique d'abstractions par contre-exemple *(analyse statique)*.

1.5 En pratique

Champs d'application. Le model checking est réservé à des systèmes finis, ou pour lesquels on peut facilement trouver une abstraction finie. Les deux champs d'applications classiques sont la validation des composants électroniques et la validation des protocoles de communication. Les composants électroniques sont typiquement de nature finie (ils manipulent des booléens), mais l'explosion des états est due au nombre gigantesque de portes logiques dans un composant. Le model checking symbolique est utile dans ce cas. Pour les protocoles, il faut souvent se ramener à une abstraction finie. Le problème principal est l'explosion due à l'entrelacement des comportements des différents agents du protocole. Les ordres partiels sont utiles dans ce cas.

Dans un futur proche, les champs d'applications devraient s'étendre considérablement. On peut déjà citer les *Web services*, cas particuliers de protocoles de communication dont l'impact économique s'annonce très important du fait de leur place centrale dans la programmation répartie. De plus le model checking devrait profiter à plein du nouveau paradigme *Model Driven Development (MDD)*, qui insiste sur les phases de spécification et l'utilisation systématique de modèles.

Outils et succès académiques. Plusieurs outils académiques de bon niveau existent, et plusieurs études de cas industrielles ont déjà été menées. Les deux outils les plus connus sont sans doute SMV développé à CMU et implantant le model checking symbolique par BDD, et SPIN développé au Bell Labs et implantant

les ordres partiels. Parmi les nombreuses études de cas réalisées, on peut noter la vérification du protocole de bus FutureBus+ IEEE. C'était la première fois qu'un protocole IEEE était débuggé par des techniques complètement automatiques. D'autres exemples sont mentionnés dans [3] et [1, 2].

Process du model checking. En pratique, le model checking est un process en trois phases :

1. Modéliser le système (système de transitions $\mathcal{M}$) et les spécifications (logique temporelle φ).
2. Vérifier si $\mathcal{M}$ satisfait φ ou non. Si non, retourner un contre-exemple.
3. Analyser les résultats obtenus :
   (a) Si oui, le *modèle* $\mathcal{M}$ est sûr. Fin. Attention : le *système réel* est-il sûr pour autant ?[5]
   (b) Si non, rejouer le contre-exemple sur le système réel.
      i. Si c'est un vrai bug, avertir le concepteur et attendre qu'il corrige. Fin.
      ii. Si le bug vient de notre modélisation, repartir à (1) en raffinant le modèle grâce au faux bug.

Évidemment, c'est un peu plus compliqué que le discours "vérification totalement automatique" :-) Je ne vous cacherai pas que les parties 1 et 3 sont très délicates. Pourtant elles ne seront pas abordées du tout ici, ni d'ailleurs dans la plupart des ouvrages à part [1]. Nous nous concentrerons sur la partie 2, qui est effectivement la partie complètement automatisée du model checking. Remarquons que des travaux de recherche en cours permettent d'automatiser en partie l'aspect raffinement.

Place dans le cycle de développement. Le model checking prend place au niveau des phases de conception du système, avant l'implantation réelle. Cela permet de découvrir les bugs au plus tôt, et plus un bug est découvert tôt, moins il coûte cher. Cependant, pour appliquer le model checking on a besoin d'un modèle formel (et fini) du système et de ses spécifications. Aussi, si le process de développement choisi n'intègre pas cela, il faudra que l'équipe de vérification refasse un modèle formel, ce qui entraîne un surcoût. À l'inverse, par exemple chez les fondeurs de processeurs, les process intègrent déjà tout une batterie de modèles formels à différents niveaux d'abstraction et des spécifications rigoureuses. Dans ce cas, le model checking s'intègre très bien au process[6].

Par rapport aux autres techniques de vérification. Voici une comparaison, forcément très subjective, entre différentes méthodes de vérification. N'hésitez pas à demander à des spécialistes des autres domaines leur avis :-)

	phase du cycle	prise en main	assisté par ordi.	surcoût	debug	validation
preuve	conception	--	-	preuves	-	++
model checking	conception	+	+	modélisation concrétisation	++	+
tests autom. (modèle)	conception	+	+	modélisation concrétisation	+	-
analyse statique	code	++	++	faux négatifs stub	-	+
tests autom. (code)	code	++	++	stubs	+	-
test autom. (code+assert)	code	++	+	assertions stubs	++	-
tests standard	code	++	-	stubs jeu de tests	-	--

[5] Cela dépend de la modélisation.
[6] Là encore, on voit tout l'intérêt que le model checking a à tirer du développement du MDD.

Utilisation industrielle. Le model checking a fait une percée remarquée dans l'industrie des composants électroniques depuis quelques années [6], soit en interne soit dans des suites de CAO. On peut citer par exemple pour les produits commerciaux : Siemens, Bull, IBM, Lucent Technologies, Cadence ; et pour les outils internes : IBM, Intel, Motorola. En 2003, un langage industriel standardisé de spécifications temporelles nommé PSL a été mis au point par un consortium regroupant entre autre Intel et IBM.

Les industriels des domaines critiques (transport, énergie) commencent également à regarder ces outils (Airbus, Bosch) dans une approche MDD, mais je ne sais pas à quel point ils sont utilisés en interne. Enfin, remarquez que Microsoft investit énormément d'argent en ce moment dans le software model checking.

Les outils cités plus haut sont quasiment à la point de la technologie académique : ils intègrent des logiques très expressives et certains utilisent des algorithmes de model checking optimaux à base d'automates d'arbres alternants. Il est d'ailleurs assez amusant de noter qu'un standard industriel comme PSL est basé sur des logiques temporelles développées à la base par des philosophes [9].

De la théorie à la pratique. Il faut bien avoir à l'esprit que si l'ambition initiale du model checking était de prouver la correction d'un système, l'utilisation qui en est actuellement faite (au moins chez les fondeurs) est plutôt celle d'un *"super testeur automatique"*, capable de vérifier tous les comportements et entrelacements du système pour des chemins de taille fixée, par exemple ≤ 50 cycles d'horloge. Des techniques spécifiques, dites *bounded model checking*, sont adaptées à ce besoin. Cette "dérive" de la validation vers le test a au moins deux raisons : prouver la validité d'un processeur entier demanderait des puissances de calcul colossales, ensuite prouver la validité sur un modèle peut vite paraître suspect : qui dit que le modèle est valide ? que le model checker est valide ? ... Alors que si vous trouvez une exécution amenant à un bug, vous avez une preuve tangible du bug.

> It has been an exciting twenty years, which has seen the research focus evolve [. . .] from a dream of automatic program verification to a reality of computer-aided design debugging.
>
> Thomas A. Henzinger

1.6 Lectures conseillées

Je me suis inspiré des articles, cours et livres listés dans les références. Je décris rapidement ici ceux que j'estime être de bons auxiliaires à ce cours.

Livres. L'ouvrage [2] présente les fondements du model checking, et est rédigé par trois des chercheurs les plus actifs du domaine. Le contenu de ce document aborde à peu près les six premiers chapitres. N'hésitez pas à aller voir les onze autres. Le livre [1] (en français) est plus orienté pratique que le précédent. La deuxième partie est consacrée à l'utilisation des logiques temporelles du point de vue utilsateur, et la troisième partie décrit les *model checkers* (outil de model checking) académiques les plus connus.

Notes de cours. Les notes de cours [4, 5] sont très claires et très détaillées. Le contenu est beaucoup plus théorique qu'ici. [5] donne les preuves de correction et de complexité des algorithmes de model checking pour CTL, LTL et CTL*. Quand à [4], il traite du model checking par automate de LTL et surtout de CTL*, avec les constructions optimales par automates d'arbres alternants. Du très haut niveau.

Articles. Les trois articles [7, 8, 9] sont des états de l'art sur certains points particuliers : comparaison des logiques LTL et CTL [7], model checking par automates [8] et enfin histoire du model checking (orienté automates quand même), de la logique aux utilisations industrielles [9]. Enfin l'article [3] est un survey sur l'emploi des méthodes formelles dans l'industrie. Il est un peu daté (1996) mais très instructif.

Première partie

Bases du model checking

Chapitre 2

Modélisation des systèmes réactifs

Nous nous intéressons à une classe particulière de programmes : les *systèmes réactifs*. Le plus souvent ces systèmes interagissent avec leur environnement et sont distribués. Quelques propriétés remarquables de ces systèmes :
- ils ne terminent pas forcément ;
- ils ne calculent pas un résultat mais plutôt maintiennent une interaction ;
- ils sont souvent dirigés par le contrôle : les types de données manipulés sont assez simples.

Quelques exemples typiques : protocole de communication, système d'exploitation.
Contre-exemple typique : un compilateur.

Le model-checking étudie des systèmes réactifs abstraits sous forme syntaxique de *machines à états*. La sémantique d'une machine à états est donnée par un *système de transitions*. Ces machines à états peuvent être plus ou moins complexes, allant des machines à états finis (= automates finis) à de vrai programmes (= machines de Turing). Cependant, plus le formalisme d'entrée est puissant, moins on peut décider de propriétés automatiquement.

2.1 Syntaxe : machines à états

Nous définissons d'abord les machines à états, qui seront notre formalisme syntaxique pour les systèmes étudiés et le format d'entrée potentiel d'un model-checker.

Définition 2.1.1 (Machine à états)**.** Une machine à états est un quadruplet $P = \langle C, V, A, T\rangle$ où
- C est l'ensemble fini des états de contrôles,
- V est l'ensemble fini des variables,
- A est un ensemble d'actions, c-à-d des formules logiques sur les variables V,
- $T \subseteq C \times A \times C$ est un ensemble fini de transitions.

La figure 2.1 ci-dessous présente une machine à états modélisant le fonctionnement d'une machine à café. Formellement, la machine est définie par $P = \langle C, V, A, T\rangle$ avec les états de contrôle $C =$\{ `idle,serving,serve`\}, les variables $V =$ \{`x,paid`\}, les actions A sont définies par des opérations arithmétiques (tests, addition, remise à zéro) et il y a 5 transition nommées `money`, `cancel`, `choice`, `served`, `back`. Par exemple, `choice` est définie par le triplet (`idle`,`‘‘x ?=2,x :=0,paid :=true’’`,`serving`).

2.2 Sémantique : systèmes de transitions

Pour l'instant, notre machine à états P n'est que syntaxique. Pour simplifier les notations, nous considèrerons toujours associé une sémantique aux variables et aux actions de la machine, c'est à dire que :

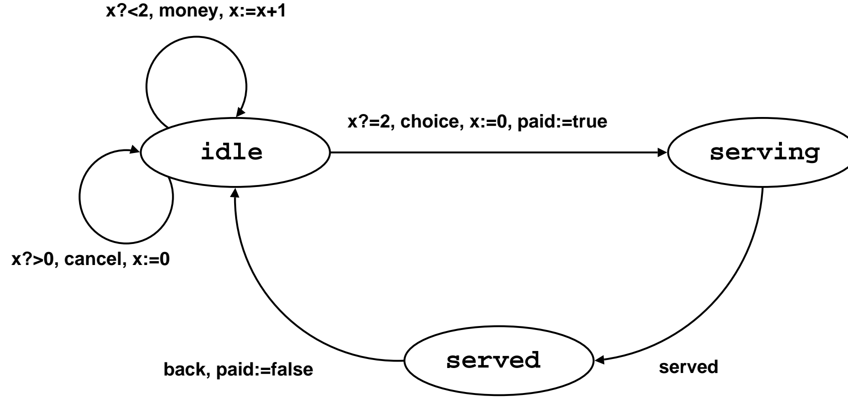


FIG. 2.1 – Machine à états représentant une machine à café.

- À chaque variable $v_i \in V$ est associé un domaine de définition D_i. On notera $D = D_1 \times \ldots \times D_{|V|}$. Une valuation de V est l'assignation à chaque $v_i \in V$ d'une valeur $d_i \in D_i$.
- Une configuration, ou état, de la machine à état est un couple $(q, d) \in C \times D$ formé d'un état de contrôle q et d'une valuation des variables d.
- À chaque action $a \in A$ est associée une relation binaire $[\![a]\!] \subseteq D \times D$. Informellement, $(d, d') \in [\![a]\!]$ signifie que si on effectue l'action a sur la valuation d, on obtient une nouvelle valuation d'.

Par exemple, pour la machine à café, $D_{\mathtt{x}} = \mathbb{N}$ et $D_{\mathtt{paid}} = \mathcal{B}$. Le domaine D vaut donc $\mathbb{N} \times \mathcal{B}$, soit des couples de valeurs pour les variables `(x,paid)`. La sémantique des actions est standard. Par exemple pour `money`, l'action ajoute 1 à `x` si `x` est inférieurer strict à 2, et laisse `paid` inchangé. On définit alors

$$[\![\mathtt{x}? < 2, \mathtt{x} := \mathtt{x} + 1]\!] = \{((x, p), (x', p')) \in (\mathbb{N} \times \mathcal{B}) \times (\mathbb{N} \times \mathcal{B}) | x < 2 \wedge x' = x + 1 \wedge p' = p\}$$

Intuitivement (x, p) représente la valeur des variables avant l'action, et (x', p') la valeur après l'action. Si $x < 2$, alors on l'incrémente de 1 et donc x après l'action vaut x avant l'action $+1$ $(x' = x + 1)$, tandis que p est inchangé $(p' = p)$. Si $x \geq 2$ l'action ne peut pas avoir lieu, cela se retrouve dans la définition donnée. Il n'y a pas de valeurs pour lesquelles $x \geq 2$.

Le comportement d'une machine à états est alors donné par un système de transitions.

Définition 2.2.1 (Système de transitions)**.** Un système de transitions S est un triplet $S = \langle Q, T, \rightarrow \rangle$ où
- Q est l'ensemble des états ou configurations,
- T est l'ensemble des transitions,
- $\rightarrow \subseteq S \times T \times S$ est la relation de transition. On note $q \xrightarrow{t} q'$ plutôt que $(q, t, q') \in \rightarrow$.

Intuitivement, $q \in Q$ représente une configuration possible du système réactif à un moment donné, et $q \xrightarrow{t} q'$ indique que si le système est dans l'état q, alors en prenant la transition t il arrivera dans l'état q'.

Passage machine à états - système de transitions. La sémantique d'une machine à états notée $P = \langle C, V, A, T \rangle$ est donnée par le système de transitions $S = \langle C \times D, T, \rightarrow \rangle$ où
- les états du système de transitions sont les configurations de la machine à états.
- la relation de transition $\rightarrow$ est définie par $(c, d) \xrightarrow{t} (c', d')$ si $t = (c, a, c')$ avec $a \in A$ et $(d, d') \in [\![a]\!]$.

Définition 2.2.2. On parlera de machine à états finis quand le système de transitions associé est fini. C'est le cas notamment quand les domaines de variables sont finis. Par exemple variables booléennes, compteurs modulo, compteurs bornés, etc.

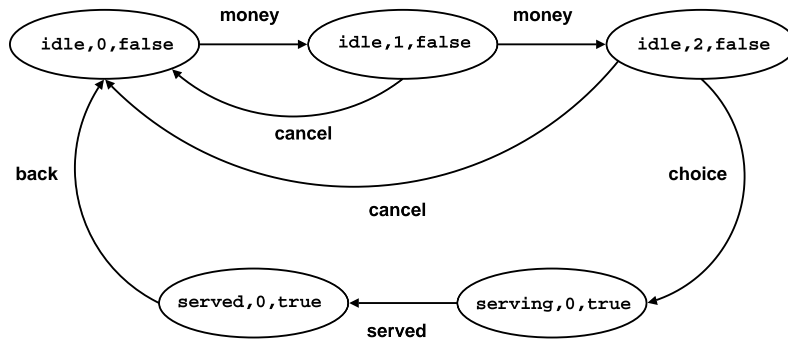


FIG. 2.2 – Une partie du système de transitions de la machine à café.

Exécution du système. Une exécution[1] σ d'un système S est une séquence *infinie* $(q_1, t_1) \dots (q_n, t_n) \dots$ d'éléments de $Q \times T$ telle que pour tout i, $q_i \xrightarrow{t_i} q_{i+1}$. Intuitivement, les q_i se suivent par la relation de transition. Le langage $\mathcal{L}(S)$ est l'ensemble des exécutions de S. Dans le cas des automates finis, et si on projette les exécutions sur les t_i, on retrouve la notion de mot[2] et de langage de l'automate.

Pour la machine à café, des débuts d'exécutions sont par exemple :
- (i,0,f) $\xrightarrow{\texttt{money}}$ (i,1,f) $\xrightarrow{\texttt{money}}$ (i,2,f) $\xrightarrow{\texttt{choice}}$ (sg,0,t) $\xrightarrow{\texttt{served}}$ (sd,0,t) $\xrightarrow{\texttt{back}}$ (i,0,f) $\xrightarrow{\texttt{money}}$ ... (utilisateur normal)
- (i,0,f) $\xrightarrow{\texttt{money}}$ (i,1,f) $\xrightarrow{\texttt{cancel}}$ (i,0,f) $\xrightarrow{\texttt{money}}$ (i,1,f) $\xrightarrow{\texttt{cancel}}$ (i,0,f) $\xrightarrow{\texttt{money}}$ (i,1,f) ... (utilisateur indécis)

Si on ne s'intéresse qu'aux transitions cela donne :
- `money`, `money`, `choice`,`served`,`back`,`money` ... (utilisateur normal)
- `money`, `cancel`,`money`, `cancel`,`money` ... (utilisateur indécis)

On pourrait aussi ne s'intéresser qu'aux suites d'états visitées. D'ailleurs c'est ce qu'on fera.

Exercice 1. *Pourriez-vous modéliser sous forme de machine à états : les automates finis, les automates à pile, les machines de Turing, un programme impératif écrit en C ?*

Exercice 2. *Le système de transitions de la machine à café présenté à la figure 2.2 n'est que partiel. Pourquoi ? Complétez le pour obtenir le système de transitions complet, et donnez sa définition formelle* $S = \langle Q, T, \rightarrow \rangle$.

2.3 Structure de Kripke

La structure de Kripke est dérivée du système de transitions, et modifiée avec les informations utiles au model checking.

Pour réellement prouver des propriétés sur nos systèmes, on va enrichir un petit peu nos systèmes de transitions, en ajoutant des propriétés atomiques sur les états. Intuitivement, un état s sera étiquetté par une propriété p si p est vraie dans s. On va aussi ajouter un état particulier $s_0 \in Q$, considéré comme l'état initial du système. Ainsi toutes les exécutions commenceront à partir de s_0.

En même temps, on va également enlever les étiquettes des arcs (les actions), car on fait le choix de ne s'intéresser qu'à des propriétés sur les suites d'états visités[3].

Définition 2.3.1 (Structure de Kripke)**.** Un structure de Kripke $\mathcal{M}$ est définie par $\mathcal{M} = \langle Q, \rightarrow, P, l, s_0 \rangle$ où

[1] On dit aussi un chemin.
[2] Bien qu'ici on soit plutôt intéressé par les mots infinis.
[3] On retrouve facilement des propriétés sur les actions en ajoutant à P des atomes comme : *"l'action a vient d'avoir lieu"*

– Q est l'ensemble des états ou configurations,
– $\rightarrow \subseteq Q \times Q$ est la relation de transition,
– P est un ensemble de propositions atomiques,
– $l : Q \rightarrow 2^P$ est la fonction d'étiquettage des états,
– $s_0 \in Q$ est l'état initial.

Si on note $\rightarrow_{\mathcal{M}}$ la relation de transition sur $\mathcal{M}$ et $\rightarrow_S$ la relation de transition sur S, on a le lien suivant : $q \rightarrow_{\mathcal{M}} q'$ ssi il existe $t \in T$ telle que $q \xrightarrow{t}_S q'$.

La notion d'exécution est modifiée en conséquence, en ne conservant que les états. On notera $\mathcal{L}(\mathcal{M}, s)$ l'ensemble des exécutions de $\mathcal{M}$ partant de l'état s, et le langage de $\mathcal{M}$ sera défini par $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{M}, s_0)$.

Exemple. On reprend l'exemple de la machine à café. On considère l'état initial correspondant à `(idle,c=0,paid=false)` et les propriétés atomiques `m` : paid = true, et `s` vrai si l'état de contrôle est dans `served`. On obtient alors la structure de Kripke complète suivante. Notez que cette fois on a représenté toute la structure, et non une partie. On peut vérifier à partir de la structure de Kripke que si un café est servi (`s`), c'est qu'il a bien été payé (`m`).

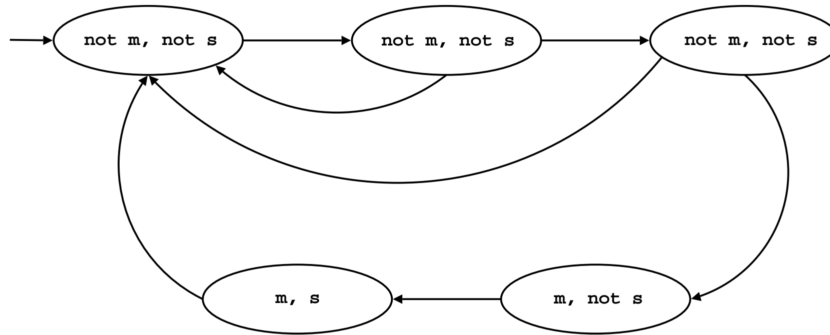


FIG. 2.3 – La structure de Kripke de la machine à café.

Dépliage. Le dépliage (*unfolding*) d'une structure de Kripke[4] est un arbre infini dont la racine est l'état initial de la structure, et chaque nœud de l'arbre a pour successeur ses successeurs par la relation de transition. La différence avec la structure de Kripke est qu'on n'identifie plus les nœuds correspondants aux mêmes états. Le dépliage permet de visualiser plus facilement les exécutions possibles de la structure. Par exemple, la figure 2.4 présente le dépliage (partiel) de la structure de Kripke de la machine à café. Les noms des transitions ont été ajoutés pour simplifier la lecture.

Exercice 3 (Exemple de l'ascenceur.)**.** *Le système de contrôle d'un ascenceur (pour 3 étages) est défini par :*
– *le contrôleur garde en mémoire l'étage courant et l'étage cible.*
– *en mode actif, quand l'étage cible est atteint, les portes s'ouvrent et le contrôleur passe en mode attente.*
– *en mode actif, quand l'étage cible est plus élevé que l'étage courant, le contrôleur fait s'élever l'ascenceur.*
– *en mode actif, quand l'étage cible est moins élevé que l'étage courant, le contrôleur fait descendre l'ascenceur.*
– *en mode attente, il se peut que quelqu'un entre dans l'ascenceur et choisisse un nouvel étage cible. L'ascenceur ferme alors les portes et redevient actif.*

[4] On peut faire pareil avec un système de transitions si on choisit un état initial.

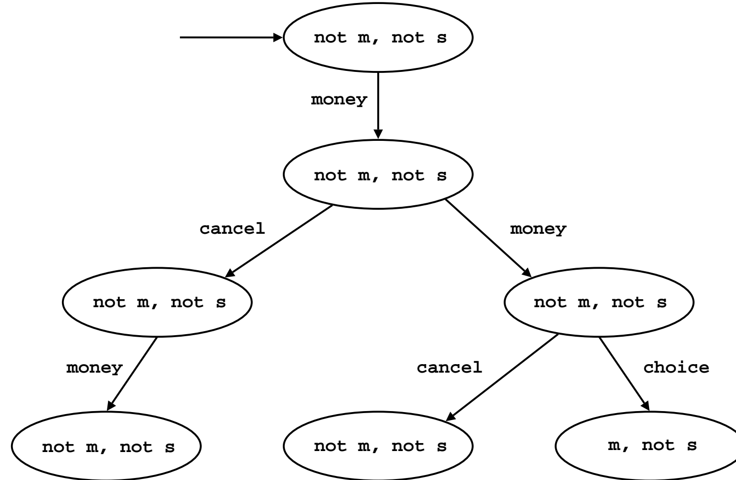


Fig. 2.4 – Dépliage (partiel) de la structure de Kripke de la machine à café.

Questions : 1. Proposez une machine à états modélisant le contrôle de l'ascenceur (définition formelle et dessin). 2. Définissez et dessinez le système de transitions correspondant. 3. Dessinez la structure de Kripke correspondante si on prend comme propositions atomiques `A` *: être actif et* `OPEN` *: portes ouvertes. 4. Est-ce que les portes peuvent s'ouvrir quand l'ascenceur est actif ?*

2.4 Discussion sur la terminologie

Les termes précédents ne sont pas fixés et varient d'un ouvrage à l'autre. La notion de système de transitions vient de l'informatique et est un peu lâche, tandis que celle de structure de Kripke vient des logiques modales. Cependant un terme est parfois utilisé à la place de l'autre. On peut aussi mettre ou non un état initial ou diverses étiquettes, sur les configurations ou sur les transitions. On pourrait ainsi appeler nos structures de Kripke des *"systèmes de transition étiquetés et initialisés"*. On peut également fusionner nos structures de Kripke et nos systèmes de transitions en une unique structure (avec le détail des transitions, les propriétés d'état et les états initiaux), ce qui est plus proche d'une implantation effective.

Pour conclure, dans les autres chapitres nous ne considèrerons plus que des structures de Kripke car c'est réellement ce qui sert dans le model checking. Cependant il faut garder à l'esprit que cette structure de Kripke provient d'un système de transitions défini syntaxiquement par une machine à états.

2.5 Espace des états, propriétés d'accessibilité et d'invariance

L'espace des états est l'ensemble des états d'un système de transitions accessibles à partir d'une configuration initiale en suivant les chemins du système de transitions. L'espace des états a une structure plus simple que le système de transitions explicite puisqu'on ne garde que les configurations en oubliant la structure de graphe, et en même temps il est suffisant pour vérifier deux types de propriétés simples mais importantes : l'accessibilité et l'invariance.

Soit un système de transitions $S = \langle Q, T, \rightarrow \rangle$, et $q \in Q$ une configuration arbitraire. La relation d'accessibilité en un coup, notée post, est définie par $\mathrm{post} = \{(q_1, q_2) \in Q \times Q | \exists t \in T. q_1 \xrightarrow{t} q_2\}$. On définit ensuite récursivement post^i par $\mathrm{post}^0 = I$, et $\mathrm{post}^{n+1} = \mathrm{post} \bullet \mathrm{post}^n$. Enfin on définit la relation d'accessibilité post^* par $\mathrm{post}^* = \bigcup_{i \in \mathbb{N}} \mathrm{post}^i$. Les états accessibles en un coup à partir de q sont donnés par $\mathrm{post}(q)$, c'est à

dire les q' tels que $(q, q') \in$ post. Les états accessibles à partir de q sont donnés par $\text{post}^*(q)$. On étend ces définitions naturellement à des ensembles $X \subseteq Q$, avec par exemple $\text{post}(X) = \bigcup_{x \in X} \text{post}(x)$.

Définition 2.5.1 (Invariant)**.** Soit un système de transitions $S = \langle Q, T, \rightarrow \rangle$. On appelle invariant de S tout $I \subseteq Q$ tel que $\text{post}(I) \subseteq I$.

Définition 2.5.2 (Point fixe)**.** Soit un ensemble K et $\tau : 2^K \rightarrow 2^K$. On dit que $X \subseteq K$ est un point fixe de τ si $\tau(X) = X$.

Exercice 4. *Soit un système de transitions $S = \langle Q, T, \rightarrow \rangle$ et $q_0 \in Q$ une configuration initiale. On note Post la fonction définie par $Post(X) = post(X) \cup X$, et $Post_{q_0}$ la fonction définie par $Post_{q_0}(X) = post(X) \cup \{q_0\}$. Montrez que :*

1. *l'ensemble d'accessibilité $post^*(q_0)$ est le plus petit invariant de S contenant q_0 ;*
2. *l'ensemble d'accessibilité $post^*(q_0)$ est le plus petit point fixe de Post contenant q_0 ;*
3. *l'ensemble d'accessibilité $post^*(q_0)$ est le plus petit point fixe de $Post_{q_0}$;*
4. *il existe $k \in \mathbb{N}$ tel que $post^*(q_0) = \bigcup_0^k post^i(q_0)$;*
5. *il existe $k \in \mathbb{N}$ tel que $post^*(q_0) = Post^i(q_0)$;*
6. *il existe $k \in \mathbb{N}$ tel que $post^*(q_0) = Post^i_{q_0}(\emptyset)$.*

L'exercice ci-dessus donne un moyen très simple de calculer $\text{post}^*(q_0)$ dans le cas fini sous l'hypothèse, très raisonnable, que les ensembles $\text{post}(X)$ sont simples à calculer. Il suffit d'itérer l'opération $X := \text{post}(X) \cup X$ en partant de $X := \{q_0\}$.

Propriétés d'accessibilité et d'invariance. Les propriétés les plus simples qu'on puisse vouloir vérifier sont celles d'*accessibilité*, du type *"il existe un état accessible où x vaut 0"*, et celles d'*invariance*, du type *"dans tous les états accessibles x est différent de 0"*. On dira que dans une propriété d'accessibilité on teste si un ensemble de mauvais états A peut être atteint, tandis que dans une propriété d'invariance on teste si les états accessibles restent dans un bon ensemble d'état I.

Ces propriétés sont simples à vérifier une fois que $\text{post}^*(q_0)$ a été calculé :
- accessibilité : A accessible ssi $\text{post}^*(q_0) \cap A \neq \emptyset$.
- invariance : I est vérifié ssi $\text{post}^*(q_0) \subseteq I$.

On peut améliorer le calcul en faisant les tests ensemblistes après chaque itération de post plutôt qu'à la fin. On y gagne respectivement si l'ensemble est accessible ou si l'invariant est violé. Sinon il faut de toute manière calculer $\text{post}^*(q_0)$ en entier pour conclure.

Exercice 5. *Quel est le lien entre accessibilité et invariance ?*

Co-accessibilité. On peut définir symmétriquement la relation de co-accessibilité en un coup pre par $pre = post^{-1}$, et on définit pre^i, pre^*, $\text{pre}(q)$, $\text{pre}(X)$, $\text{pre}^*(q)$ et $pre^*(X)$. Un algorithme de calcul itératif de $\text{pre}^*(q)$ est possible. La différence principale est que $\text{pre}(q)$ n'est pas toujours évident à calculer, même si dans le cas fini on peut toujours s'en sortir en énumérant.

Exercice 6. *Comment vérifier une propriété d'accessibilité en passant par le calcul de pre^* ? Et pour l'invariance ?*

Remarques diverses. Les propriétés d'accessibilité et d'invariance sont simples mais extrêmement importantes en pratique. On remarque que leur vérification a une complexité polynômiale dans le nombre d'états du système de transitions, donc souvent exponentielle dans la taille de la machine à états initiale si on a des variables pouvant prendre beaucoup de valeurs. C'est toute l'histoire du model checking fini : se battre contre cette explosion d'états.

Les avantages et inconvénients des calculs en avant (post) et en arrière (pre) ont fait couler beaucoup d'encre, mais il n'y a pas de solution définitive. Le calcul en avant ne prend en compte que les états du système,

mais cherche un peu au hasard sans s'occuper de l'objectif à atteindre. À l'inverse le calcul en arrière est contraint par l'objectif, mais rique de s'encombrer d'états qui n'appartiennent pas au comportement normal du système. Dans le cas où on dispose déjà d'un invariant I, il est souvent très efficace de calculer en arrière en coupant à chaque fois avec l'ensemble I.

Enfin remarquez que les algorithmes de calcul de $\text{post}^*(q_0)$ et $\text{pre}^*(q_0)$ présentés ici ne fonctionnent pas sur des systèmes infinis.

2.6 Systèmes concurrents

Comme les systèmes réactifs sont souvent distribués, il sera plus simple de les modéliser comme des ensembles finis de machines à états concurrentes (s'exécutant en parallèle). Il y a de nombreux travaux sur la sémantique de la concurrence. Voici quelques distinctions classiques sur les systèmes concurrents.

Modes d'exécution. Un *système synchrone* est rythmé par une horloge global, et tous ces composants avancent à la même vitesse, c-à-d une action par *tick* d'horloge. Donc une transition du système globale est le déclechement simultané d'une transition de chaque composant. Formellement, si on note T_i les transitions de la machine P_i et T les transitions de la machine composée $P_1 \times \ldots \times P_n$, alors $T = T_1 \times \ldots \times T_n$. *Les processeurs sont typiquement synchrones.*

Au contraire, un *système asynchrone* n'a pas d'horloge globale, et une transition du système globale est soit *(sémantique 1)* une transition d'un seul des composants et $T = \cup_i T_i$ (dans ce cas il n'y jamais deux actions simultanées) ; soit *(sémantique 2)* un ensemble d'au plus une transition par composant et $T = (T_1 + \varepsilon) \times \ldots \times (T_n + \varepsilon)$ où ε désigne l'action qui ne fait rien (ici les actions simultanées sont possibles). *Les protocoles de communication sont typiquement asynchrones.*

Modes de communication. Les composants peuvent interagir en modifiant des variables communes ou en s'envoyant des messages.

Dans le cas de *la mémoire partagée*, chaque machine a ses variables privées et il existe un pool de variables communes / partagées. L'avantage est l'efficacité, la difficulté principale est l'accès concurrent aux donnéees : un composant rique de modifier une valeur qu'un autre composant voulait stocker.

Dans le cas des *envois de messages*, un composants peuvent envoyer un message à un autre composant via un canal de communication FIFO. Il peut y avoir différents canaux de communication, ils peuvent être privés ou communs, en lecture ou en écriture, avoir des bornes sur le nombre de messages présents simultanément ou encore être plus ou moins fiables, par exemple si la perte de messages est possible. Normalement l'envoi d'un message et sa réception ne sont pas synchronisés : le composant qui envoie continue sont exécution, et le récepteur lira peut être le message plus tard.

Un mécanismes très classique d'envoi de message permet de synchroniser deux composants : le *rendez-vous* (ou handshaking). Dans le rendez-vous, un émetteur envoie sur un canal privé un message et attend que le récepteur lui retourne un acquittement de réception sur un autre canal privé. Vu de loin, tout se passe comme si l'envoi et la réception du message étaient simultanés. Ainsi quand l'envoyeur reprend son exécution, il sait à quel point de programme se trouve le récepteur, et réciproquement.

Enfin dans le *broadcast* un émetteur diffuse un message à tous les autres participants, et souvent n'attend pas d'acquittement.

Exemples de systèmes concurrents.

- threads JAVA : asynchrone, variables partagées.
- processus UNIX : asynchrone, envoi de messages.
- composants électroniques dans un processeur : synchrone, variables partagées.
- agents d'un protocole de communication internet : asynchrone, envoi de messages.
- calculateur massivement parallèle GALS[5] : synchrone + asynchrone, variables partagées et messages.

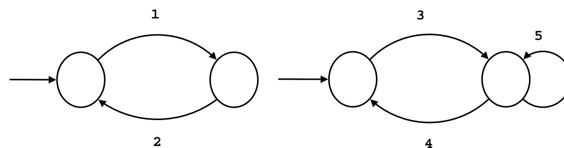
[5] *Globally Asynchronous Locally Synchronous.*

Machines concurrentes en pratique. Voici quelques éléments classiques de modélisation de systèmes concurrents. On se donne un ensemble de machine à états.

- Le produit synchronisé (surtout le choix des transitions possibles) se fait selon le mode d'exécution choisi.
- Les variables partagées s'intègrent facilement en identifiant les variables de même nom des différents composants, ou en rajoutant des contraintes d'égalité entre variables.
- On peut rajouter des contraintes spécifiques de synchronisation, dites vecteurs de synchronisation, c'est à dire que certains groupes de transitions ne peuvent être que simultanées. Cela permet de modéliser des rendez-vous et des situations plus générales, dues par exemples à des parties non modélisées du système (propriétés électroniques, mécaniques, etc.)
- Parfois seuls les rendez-vous sont disponibles : dans ce cas les synchronisations se font toujours entre deux transitions, *l'émetteur* étiquetté `!m` envoie le message `m` au *récepteur* étiquetté `?m`. Les deux transitions ne peuvent être franchies que simultanément, modélisant l'envoie du message avec attente (et réception) de l'accusé de réception. La réception et l'envoi sont bloquants.
- Des opérations de broadcast sont plus rarement disponibles.
- Enfin on peut préférer modéliser l'envoi de messages en utilisant un type spécifique de variables partagées : les files de communication. Dans ce cas, `?m` signie lire le message `m` en queue de file et `!m` signie mettre le message `m` en tête de file. Les files peuvent être parfaites ou à pertes, de taille bornée ou non. La modélisation est plus fidèle que les rendez-vous, mais le système obtenu est plus complexe.

Entrelacement et explosion combinatoire. En général, le comportement de la machine $P_1 \times P_2$ est beaucoup plus complexe que le comportement de P_1 ou P_2. C'est ce qu'on appelle le phénomène d'explosion combinatoire : le système de transitions de $P_1 \times P_2$ (dans le cas fini) a une taille beaucoup plus grande que $|S_1| + |S_2|$, dû aux entrelacements de transitions possibles. Typiquement pour deux systèmes asynchrones (1) sans variable partagée ni synchronisation, $|S| = |S_1| \times |S_2|$ en nombre d'états, et le nombre de transitions augmente plus, surtout avec la sémantique asynchrone (2).

Exercice 7. *Soit les machines concurrentes suivantes.*



Quelles sont les transitions du système concurrent dans les cas suivants :

1. *sémantique synchrone ;*
2. *sémantique asynchrone (1) ;*
3. *sémantique asynchrone (2) ;*
4. *sémantique synchrone + synchronisation entre 1 et 4 ;*
5. *sémantique asynchrone (1) + synchronisation entre 1 et 4 ;*
6. *sémantique asynchrone (2) + synchronisation entre 1 et 4 ;*

Quel est le lien entre vecteur de synchronisations et rendez-vous ?

2.7 Hypothèses d'équité

Le processus d'abstraction inhérent à toute modélisation peut introduire des comportements irréalistes, soit impossibles dans la machine réelle soit hautement improbables. De tels comportements peuvent être par exemple : rien ne se passe dans le système, un canal de communication perd systématiquement tous les messages, un composant n'a jamais la main, etc. Ces comportements sont très gênants en pratique car

ils risquent de fausser l'analyse du modèle, par exemple si les seules exécutions qui mènent à une erreur s'avèrent être des exécutions aberrantes.

Les hypothèses d'équité permettent justement d'écarter ces chemins aberrants en se restreignant aux chemins *équitables*, c'est à dire ceux le long desquels le système progresse régulièrement, selon des critères à définir. Une hypothèse d'équité peut être par exemple que tout composant obtient la main infiniment souvent le long de l'exécution. Il y a deux manières de rajouter des hypothèses d'équité : en modifiant la formule à vérifier ou en modifiant (un peu) la sémantique du modèle.

Équité dans les formules. Comme on veut vérifier une propriétés exprimée par une formule de logique φ, on va alors vérifier plutôt une formule $\psi \to \varphi$ où ψ décrit nos contraintes d'équité. Cette solution est très souple mais elle nécessite des logiques suffisament expressives, et donc plus difficiles à vérifier automatiquement.

Équité dans le modèle. On peut aussi préférer modifier directement la sémantique des systèmes de transitions en ne conservant que les chemins passant infiniment souvent par certains états spécifiés par l'utilisateur. On peut ainsi simuler des formules d'équité suffisantes en pratique.

Techniquement, on ajoute au système de transitions un ensemble fini F d'ensembles d'états $F_i \subseteq Q$ appelés contraintes d'équité. Une exécution *fair* du système est une exécution passant infiniment souvent par chaque F_i. Du coup $\mathcal{L}(\mathcal{M})$ et $\mathcal{L}_F(\mathcal{M})$ sont différents, et certaines configurations qui étaient accessibles peuvent ne plus l'être avec une sémantique $fair$.

Exercice 8 (**)**.** *Comment calculer* $post_F^*$ *et* pre_F^* *dans le cas d'un système de transitions avec hypothèses d'équité sur le modèle ?*

Équité et produit de machines à états. On pourrait penser que les hypothèses d'équité ne doivent être ajoutées qu'exceptionnellement. Il n'en est rien, c'est au contraire quasi-obligatoire si on passe par des machines à états finis concurrentes. Remarquons déjà que le modèle sous-jacent des systèmes de transitions assure implicitement que ne sont considérés que les chemins où le système fait quelquechose s'il le peut. Cela règle *"gratuitement"* un des cas dégénérés évoqués plus haut. Malheureusement si on considère des machines concurrentes, cette hypothèse implicite s'applique au système de transitions total et pas à chacune des machines. Ainsi, sans hypothèse supplémentaire, il se peut très bien qu'une des machines monopolise toute l'exécution tandis que les autres la regardent.

2.8 Propriétés de sûreté.

On peut aller avec de simples calculs d'ensembles d'états un peu plus loin que l'accessibilité et l'invariance.

Définition 2.8.1. On définit informellement les propriétés de sûreté comme celles stipulant que *"quelquechose de mauvais n'arrive jamais"*. Une définition semi-formelle pratique est que quand une propriété de sûrete est violée, il existe un contre-exemple fini. C'est à dire une exécution finie qui viole la propriété.

Ainsi par exemple l'invariance est un cas particulier de sûreté, mais pas l'accessibilité. Il y a d'autres propriétés de sûreté, par exemple *"je ne peux accéder à mon compte sans avoir donné le bon password."*

En principe on peut ramener n'importequ'elle propriété de sûreté à du calcul d'états accessibles, à condition de modifier le système de départ. L'avantage est d'utiliser des techniques simples et homogènes, l'inconvénient est que le nouveau système modifié peut être beaucoup plus large que l'ancien. Il y a deux manières de faire : l'ajout de *variables d'historique* ou la synchronisation avec un *automate observeur*. Nous allons illustrer les deux méthodes sur des exemples.

Variables d'historique. Admettons que l'on veuille vérifier la propriété φ : *"Lors d'une exécution, l'état de contrôle p est toujours visité avant l'état de contrôle q"*. On rajoute une nouvelle variable h qui vaut 0 dans q_0, et est mise à 1 par les transitions qui mènent à p. Dès lors, vérifier la propriété φ revient à vérifier *dans notre système modifié* la propriété d'invariance I : *"tous les états accessibles vérifient* $q \rightarrow (h = 1)$ *"*.

Les variables d'historique permettent de vérifier des propriétés de sûreté en se souvenant de certains faits passés lors de l'exécution. Il y a deux désavantages : le nouveau système de transitions est plus large, par exemple au pire deux fois plus large si on rajoute juste un booléen, et surtout il faut un nouveau système modifié pour chaque propriété ce qui peut s'avérer difficile à maintenir. On peut pallier à ce dernier problème par le biais d'automates observeurs.

Automates observeurs. L'idée est de mettre toutes les variables d'historique dans une machine à états à part (automate observeur) qui sera synchronisée avec le système initial. On ne gagne pas en expressivité (on pourrait coder l'automate observeur par des variables d'historique) mais on gagne en clarté : le système ne change pas et il y a un automate observeur par propriété à vérifier.

Pour l'exemple ci-dessus, l'automate observeur aurait une variable h initialement à 0 et des transitions mettant h à 1, synchronisées avec les transitions du système initial menant à l'état de contrôle p.

Exercice 9. *On se donne un système de transitions S dont certaines transitions sont distinguées et correspondent à des opérations d'aquisition de verrou (`lock`), de rendu de verrou (`unlock`), de lecture (`read`) et d'écriture (`write`). On se donne la propriété φ suivante :*

Si on ne regarde que les `lock` et `unlock` : `unlock` est toujours précédé directement de `lock` et `lock` est toujours suivi directement de `unlock`, ET une suite arbitraire de `read`,`write` est toujours précédée directement d'un `lock` et fermée directement par un `unlock`.

Questions :

1. *Est-ce que φ est une propriété de sûreté ? Sinon modifiez là en conséquence.*
2. *Écrivez un automate observeur pour vérifer φ (ou sa modification) et expliquez la nouvelle propriété à vérifier.*

Remarque. Pour des propriétés plus compliquées, par exemple *"un jour j'arriverai à Là-bas"*, on ne peut plus se ramener à de l'accessibilité. Intuitivement, la différence est que les contre-exemples sont maintenant infinis, puisque ce n'est pas parceque je ne suis pas encore arrivé que je n'arriverai jamais. On peut étendre la méthode en synchronisant cette fois avec des automates de mots infinis. Cette technique est à la base du model checking de LTL présenté plus loin.

2.9 Quelques points de modélisation

Cette section regroupe quelques considérations sur certains mécanismes des machines à états et leur utilité pour la modélisation. Cependant, gardez à l'esprit que ce cours n'est pas un cours de modélisation. Nous ne considérons que des modèles assez simples, et nous nous intéressons surtout aux structures de Kripke dérivées.

Non déterminisme. La relation de transition n'est pas forcément déterministe. Formellement, cela signifie que dans nos systèmes de transitions, il peut y avoir une transition t et trois états q, q', q'' tels que $q \xrightarrow{t} q'$ et $q \xrightarrow{t} q''$. Ainsi, partant de la configuration q et effectuant la transition t, on peut aller arbitrairement à q ou à q'. Il y a deux utilisations principales : d'une part modéliser des choix de l'environnement dans les systèmes ouverts, d'autre part abstraire certains éléments du système initial trop complexes ou remplacer un des composants par sa spécification.

Actions observables. On peut considérer que seules certaines actions/états de la machine sont observables par un utilisateur extérieur, ou encore que le label observé peut être différent de l'action/état. Dans ce cas, on peut avoir deux labels : un label d'action/état comme précédemment, et un label observable. Les actions/états non observables sont alors étiquettées ε. On retrouve un peu la notion d'ε-transitions des automates finis.

Types classiques de variables. Les types de variables suivants sont très classiques pour les machines à état : les booléens, les piles (appels de procédure), les files FIFO (canaux de communication), les compteurs (ressources) et les horloges (aspects temps-réel).

Modularité, hiérarchie. Les machines à états sont un formalisme très simple et bien adapté à la vérification. Malheureusement il est très laborieux d'écrire des spécifications réalistes par ce biais, car les constructions offertes sont de trop bas niveau même avec la synchronisation de composants. Les concepteurs aiment utiliser des constructeurs plus élaborés comme par exemple des modules (un composant est défini une fois et réutilisé à plusieurs endroits) et de la hiérarchie (des composants simples sont associés en composants complexes via différents mécanismes). Certains model checkers proposent des langages d'entrée avec de tels constructions, par exemple AltaRica (Uni. Bordeaux) ou SMV.

Pour la vérification de ces formalismes étendus, on peut bien sûr calculer le système de transitions puis se ramener aux techniques usuelles. Une meilleur solution est d'essayer d'adapter les techniques de vérification au modèle de haut niveau, pour profiter du niveau d'abstration plus élevé. Cependant il n'y a malheureusement guère de résultats en ce sens actuellement.

Systèmes ouverts. Les systèmes ouverts interagissent avec un environnement extérieur. Le système reçoit des informations par des capteurs (ex : mesurer la profondeur d'un réservoir), et peut parfois agir sur l'environnement via des actionneurs (ex : ouvrir ou fermer les vannes du barrage). On doit vérifier la correction du système pour toute suite d'actions de l'environnement. Selon les cas, l'environnement peut soit être modélisé comme une autre machine à états (souvent indéterministe) concurrente de notre système, ou alors décrit par des formules de logiques restreignant son comportement. Ce dernier cas est plus complexe à gérer.

Chapitre 3

Logiques temporelles

Ce chapitre introduit les *logiques temporelles*, qui seront le formalisme employé pour spécifier les comportements attendus de nos systèmes réactifs. D'un côté, ces logiques surpassent les autres moyens usuels de spécification du comportement, que ce soit le langage naturel (trop ambigu) ou des formalismes à base de diagrammes (peu expressifs et souvent ambigus). De l'autre, elles sont plus concises et faciles à lire que des langages de spécification plus généralistes (comme la logique du premier ordre). Enfin, et surtout, leur vérification peut être automatisée, contrairement aux exemples mentionnés ci-dessus.

Après avoir introduit intuitivement la notion de logique temporelle et les principaux concepts, on va présenter formellement les trois logiques temporelle LTL, CTL* et CTL. LTL est une logique linéaire, c-à-d qu'on s'intéresse aux exécutions du système sans prendre en compte les entrelacements des différents futurs possibles. CTL* est la logique la plus expressive que nous verrons. C'est une logique branchante. Enfin CTL est aussi une logique branchante, moins expressive que CTL*, mais le model-checking est beaucoup plus facile.

3.1 Panorama de propriétés temporelles

Voici une liste de catégories de propriétés temporelles utiles en vérification. Certaines ont déjà été mentionnées au chapitre précédent.

Accessibilité. *Une certaine situation peut être atteinte.*

- le compteur x peut prendre la valeur 0 ;
- le point final du programme peut être atteint.

Invariance. *Tous les états du système satisfont une bonne propriété.*

- pas de division par 0, respect des préconditions de fonction, pas de débordement de tableaux ;
- (*exclusion mutuelle*) deux processus ne sont jamais simultanément en section critique.

Sûreté. *Quelquechose de mauvais n'arrive jamais.*

- chaque fois que j'utilise `unlock`, j'ai utilisé `lock` avant ;
- chaque fois que j'accède à mon compte, j'ai bien rentré le bon mot de passe au préalable ;
- (*correction partielle*) quand la précondition du programme est respectée et que le programme termine alors la postcondition est respectée.

Vivacité/progrès . *Quelquechose de bon finira par arriver.*

- quand une impression est lancée, elle finira par s'achever ;
- quand un message est envoyé, il finira par être reçu ;

– (*correction totale*) quand la précondition du programme est respectée, alors le programme termine et la postcondition est respectée.

Équité. *Quelquechose se répètera infiniment souvent.*
– (*équité faible*) si un processus demande continuellement son exécution, il finira par l'avoir.
– (*équité forte*) si un processus demande infiniment souvent son exécution, il finira par l'avoir.

Absence de blocage. *Le système ne se bloque pas.*
– (*non blocage total*) il existe au moins une exécution infinie de la machine ;
– (*non blocage partiel*) il n'y a pas d'état bloquant.

Équivalence comportementale. *Est-ce que deux systèmes se comportent de la même manière ?*
Cela permet par exemple de réaliser rapidement un premier système, de le valider puis de remplacer un composant par un autre équivalent et plus optimisé sans avoir à tout revalider.

Exercice 10. *Quels sont les liens entre accessibilité, invariance et sûreté ? La vivacité est-elle de la sûreté ? Justifiez. Même question pour l'équité et l'absence de blocage.*

3.2 Intuitions sur les logiques temporelles

Logique temporelle versus logique temporisée. Les logiques que nous verrons dans ce chapitre sont dites *temporelles*. Elles décrivent le séquencement d'évènements observés et peuvent exprimer par exemple la *causalité* : *"chaque fois que j'observe q, j'ai observé p avant"*. Ces logiques ne manipulent jamais explicitement le temps comme durée absolue : entre deux évènements observés (observations), il peut a priori se passer une seconde ou une journée. On ne pourra pas exprimer par exemple que *"quand je vois p, je vois ensuite q exactement 3 secondes après"*. On pourra juste dire : *"quand je vois p, je vois ensuite q exactement 3 observations après"*.

Il existe des logiques dites *temporisées* manipulant explicitement le temps. Le model checking de telles logiques demande des techniques plus élaborées et plus coûteuses, bien que partageant les mêmes bases. Ces techniques ne seront pas abordées dans ce cours.

Dans la suite nous parlerons parfois d'instants, mais il faudra comprendre observations.

Pourquoi des logiques ? On pourrait imaginer décrire les propriétés temporelles soit directement dans les langues naturelles, soit au moyen d'un formalisme graphique. Malheureusement, les langues naturelles sont trop ambigues pour les besoins de l'informatisation et pas assez concises : des propriétés un peu complexes demanderaient des dizaines de mots, voir de phrases. Les formalismes graphiques (ex : Message Sequence Charts, utilisés dans les Télécoms) sont plus intéressants car faciles à comprendre et concis. Cependant souvent, soit ils ne peuvent pas exprimer des propriétés un peu élaborées, soit leur sémantique n'est pas suffisament claire.

Les logiques sont des langages de spécification formels, c'est à dire qu'ils ont une définition mathématique précise. Les avantages reconnus pour le génie logiciel sont :
– exprimer sans ambiguité les propriétés attendues du système (spécification, documentation),
– permettre la simulation, voir la vérification automatique du système (vérification).

Pourquoi des logiques temporelles ? On pourrait imaginer de modéliser le temps en logique classique du premier ordre : une variable $t \in \mathbb{N}$ indique l'instant où a lieu une observation et on temporise les prédicats en les faisant dépendre de t. Dans cette optique $P(t)$ signifie que P est vrai à l'instant t. Par exemple pour la propriété *"Chaque panne est suivie d'une alarme"*, on écrirait : $\forall t, \text{panne}(t) \rightarrow (\exists t' \geq t, \text{alarme}(t'))$. Ce formalisme est très expressif, malheureusement il est peu concis et pas très lisible, même pour des experts. De plus son expressivité implique qu'on ne peut espérer prouver automatiquement de telles spécifications.

On va donc se tourner vers des logiques moins expressives (pas de quantification arbitraire), mais plus adaptées à nos besoins grâce à l'ajout de deux types d'opérateurs liés au temps : les *connecteurs temporels* et les *quantificateurs de chemins*.

On parle de logique*s* temporelle*s* au pluriel, définies selon les opérateurs ajoutés.

Connecteurs temporels et propriétés de chemin. Les connecteurs temporels permettent de parler du séquencement des états/ évènements observés *le long d'une exécution*[1]. On introduit les connecteurs suivants : **X** (next), **F** (future), **G** (globally) et **U** (until). Dans la suite, les p_i sont des propriétés atomiques des états observés.

- **X** (next) : **X**p signifie que p est vrai dans l'état suivant le long de l'exécution.

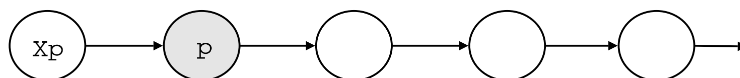


Fig. 3.1 – Opérateur **X**

- **F** (future) : **F**p signifie que p est vrai plus tard au moins dans un état de l'exécution.

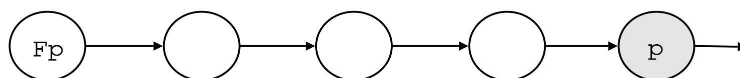


Fig. 3.2 – Opérateur **F**

- **G** (globally) : **G**p signifie que p est vrai dans toute l'exécution.

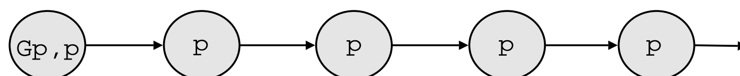


Fig. 3.3 – Opérateur **G**

- **U** (until) : p**U**q signifie que p est toujours vrai jusqu'à un état où q est vrai.

On exprime des propriétés complexes le long d'un chemin en imbriquant les connecteurs. Voici quelques exemples de *propriétés de chemin*.

- accessibilité (le long du chemin) : $\mathbf{F}(x = 0)$
- invariance (le long du chemin) : $\mathbf{G}\neg(x = 0)$
- vivacité (le long du chemin) : $\mathbf{G}(p \rightarrow \mathbf{F}q)$
- correction totale (le long du chemin) : $(\text{init} \wedge \text{precondition}) \rightarrow \mathbf{F}(\text{end} \wedge \text{postcondition})$

Exercice 11. *Quelques petits exercices sur les connecteurs temporels :*

- **F**p *est-il vrai si* p *vrai tout de suite dans l'état courant ?*
- **G**p *est-il vrai si* p *faux dans l'état courant et vrai partout ailleurs ?*
- p**U**q *est-il vrai si* p *faux et* q *vrai dans l'état courant ?*
- p**U**q *est-il vrai si* q *est toujours faux, et* p *toujours vrai ?*
- *Faites un schéma d'une exécution où* p *est vrai infiniment souvent. Comparez avec vos camarades. Qu'en déduisez-vous sur les langages naturels et graphiques ?*

Quantificateurs de chemins et propriétés des états (d'un système). Jusqu'à maintenant on a considéré une seule exécution du système, et les connecteurs permettent de parler de propriétés le long de cette exécution. C'est une vision linéaire du temps : le futur est fixé. Cependant, on peut aussi voir le temps

[1]Ce n'est donc pas suffisant pour exprimer des propriétés sur tout un système, mais c'est un début.

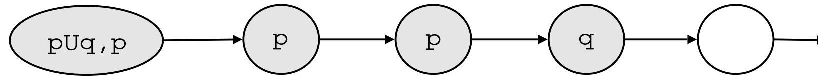


FIG. 3.4 – Opérateur **U**

comme une structure branchante : à chaque état du système, plusieurs futurs sont possibles, selon l'action qui sera effectuée[2]. Les quantificateurs de chemins permettent de quantifier des propriétés sur les exécutions futures possibles à partir d'un état. On parle de *propriétés d'états (d'un système)*.

- l'expression $\mathbf{A}\varphi$ signifie que toutes les exécutions partant de l'état courant satisfont φ.
- l'expression $\mathbf{E}\varphi$ signifie qu'il existe (au moins) une exécution partant de l'état courant qui satisfait φ.

On peut alors associer un quantificateur de chemin à une formule de chemin bâtie sur $\mathbf{X}, \mathbf{F}, \mathbf{G}, \mathbf{U}$. On verra quand on étudiera la logique CTL* qu'on peut même combiner de manière plus complexe. Voici quatre cas de base (s est l'état courant) :

- $\mathbf{EF}p$: il existe un chemin partant de s qui atteint p.

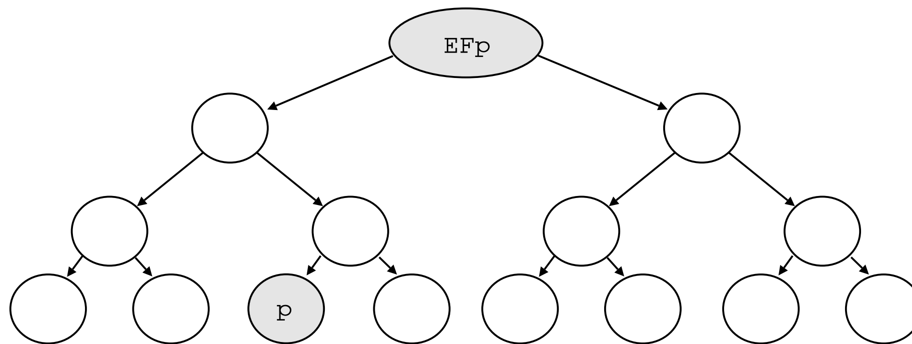


FIG. 3.5 – **EF**

- $\mathbf{AF}p$: tous les chemins partant de s finissent par atteindre p.

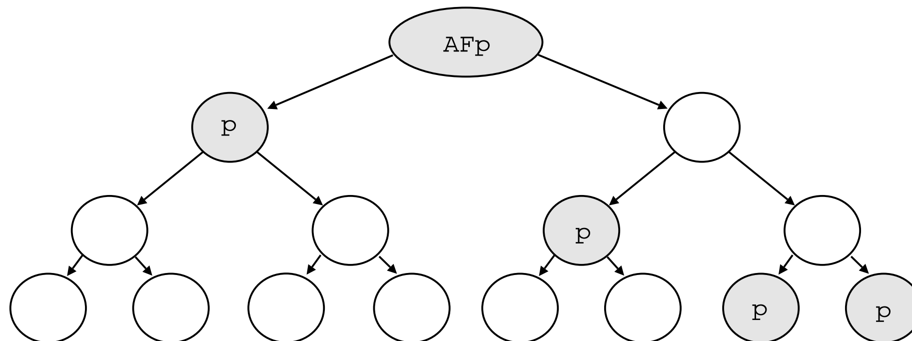


FIG. 3.6 – **AF**

- $\mathbf{EG}p$: il existe un chemin partant de s tel que p est vrai tout au long du chemin (et aussi dans s).
- $\mathbf{AG}p$: p est toujours vrai en partant de s (et aussi vrai dans s).

Les propriétés qui nous intéressent sur un système S sont en fait les propriétés d'états sur l'état initial du système.

[2]La structure de Kripke modélise ces entrelacements de futurs possibles.

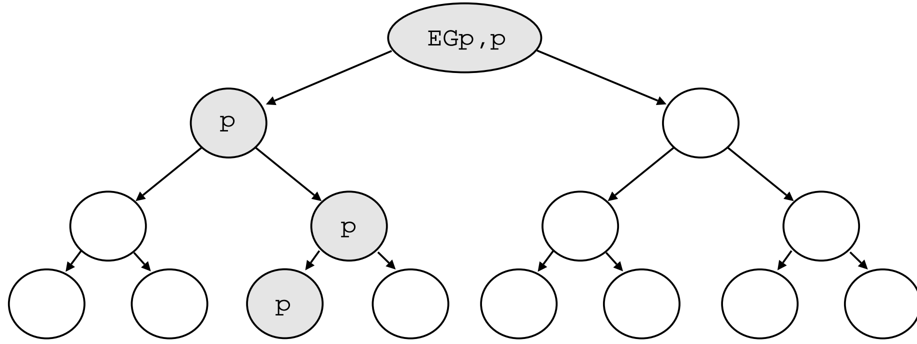


FIG. 3.7 – **EG**

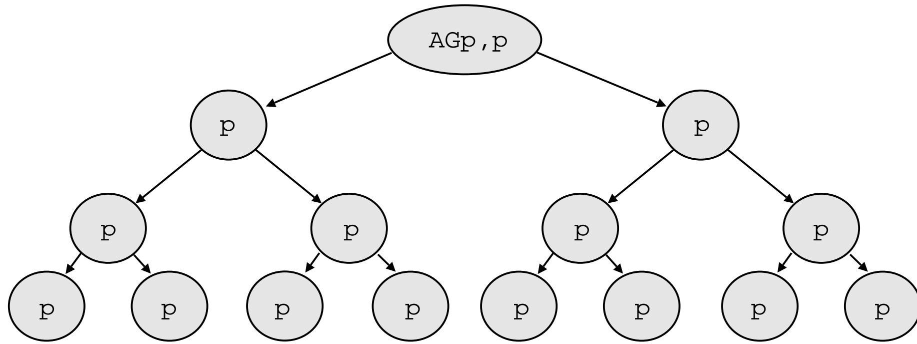


FIG. 3.8 – **AG**

Exercice 12. *Dessinez des dépliages sur lesquels vous illustrerez les propriétés* **EX***,* **AX***,* **EU***,* **AU**.

Exercice 13. *Exprimer les propriétés suivantes :*

1. *Tous les états satisfont p.*
2. *On peut atteindre p par un chemin où q est toujours vrai.*
3. *Quelquesoit l'état, on finit par revenir à l'état initial init.*
4. *Quelquesoit l'état, on peut revenir à l'état initial init.*
5. *Absence de deadlock (partiel).*

Différents types de logiques temporelles. Comme précédemment dit, il existe plusieurs logiques temporelles selon les opérateurs du temps employés. Distinguer selon les opérateurs permis est la manière la plus simple de distinguer deux logiques temporelles. Voici d'autres éléments de distinction.

Linéaire vs. branchant. Dans les logiques linéaires on s'intéresse aux exécutions du système indépendamment les unes des autres, sans prendre en compte les entrelacements des différents futurs possibles à un point donné de l'exécution. Autrement dit, on s'intéresse à un ensemble d'exécutions plutôt qu'à un arbre des exécutions possibles. La quantification sur les chemins est donc quasi-absente des logiques linéaires. Au contraire les logiques branchantes permettent de quantifier sur les futurs possibles.

Expressivité. Intuitivement, est-ce que deux logiques peuvent exprimer la même chose, ou est-ce que l'une sait dire plus de choses que l'autre, ou est-ce qu'elles sont incomparables ? L'expressivité est une notion *sémantique*, pas syntaxique, car des formules différentes peuvent parfois signifier la même chose. Typiquement, la logique classique du premier ordre est plus expressive que les logiques temporelles usuelles.

Concision. Quand deux logiques peuvent exprimer la même chose, il est intéressant de savoir laquelle le fait avec les phrases les plus courtes. C'est la concision. Ainsi par exemple, une logique temporelle est toujours plus concise que la logique classique du premier ordre.

Pouvoir de séparation. Capaciter d'une logique à distinguer deux structures de Kripke. Une logique sépare $\mathcal{M}_1$ et $\mathcal{M}_2$ si il existe une formule φ de la logique en question telle que $\mathcal{M}_1 \models \varphi$ et $\mathcal{M}_2 \not\models \varphi$.

Avec ou sans passé. Nous n'avons présenté ici que des connecteurs du futurs. On pourrait aussi définir des connecteurs du passé[3], par exemple pour dire que quand j'observe q, c'est que j'ai observé p avant. On ne le fera pas ici. Retenez que ca n'augmente pas l'expressivité (on peut se ramener aux opérateurs du futur), mais le pouvoir de concision. Retenez aussi qu'à l'heure actuelle on ne sait pas comment vérifier efficacement ces propriétés : on est obligé de revenir à la traduction (exponentielle) en opérateurs du futur.

Complexité du model checking. C'est une propriété fondamentale pour l'informatique, qui indique si le model checking peut être fait efficacement (par un algorithme polynômial) ou pas. Malheureusement, plus une logique est expressive, plus le problème du model checking est difficile. Attention : entre deux logiques L_1 et L_2 de même expressivité, si la complexité de L_1 est meilleure mais que sa concision est moins bonne, L_1 n'est pas forcément plus facile à vérifier en pratique.

Dans la suite on va présenter trois logiques temporelles différentes : LTL, CTL et CTL*. LTL est une logique linéaire, c-à-d qu'on s'intéresse aux exécutions du système sans prendre en compte les entrelacements des différents futurs possibles à un point de l'exécution. La quantification sur les chemins y est donc quasi-absente. CTL* est la logique la plus expressive que nous verrons, il n'y a pas de limitation à l'utilisation des connecteurs temporels et quantificateurs de chemins. C'est une logique branchante. Enfin CTL est aussi une logique branchante, mais où l'emploi des connecteurs temporels est restreint. La logique est moins expressive que CTL* mais le model checking est bien plus facile.

Exercice 14. *Quel est l'intérêt d'utiliser des logiques temporelles par rapport à (a) langage naturel, (b) formalisme graphique, (c) logique classique du premier ordre. Vous discuterez les points suivants : précision du langage, concision, expressivité, simplicité d'accès.*

Comparez maintenant entre logique classique du premier ordre et logique classique du premier ordre augmentée des opérateurs temporels.

3.3 Logique linéaire LTL

LTL est une logique linéaire. Le seul quantificateur autorisé est **A**, seulement en début de formule. Dans la suite on se donne un ensemble fini de propriétés atomiques AP.

On va tout d'abord définir LTL sur un chemin, puis on définira LTL sur une structure de Kripke.

Définition 3.3.1 (LTL sur un chemin)**.** La logique temporelle LTL (*Linear Temporal Logic*) sur un chemin est définie par la grammaire suivante :

$$\varphi ::= p \in \mathrm{AP} \;—\; \neg\varphi \;—\; \varphi \vee \varphi \;—\; \varphi \wedge \varphi \;—\; \mathbf{X}\varphi \;—\; \mathbf{F}\varphi \;—\; \mathbf{G}\varphi \;—\; \varphi\mathbf{U}\varphi$$

Le domaine d'interprétation est un ensemble de chemins σ. Un chemin est une séquence infinie $\sigma = s_0 s_1 s_2 \ldots s_n \ldots$ d'états $s \in Q$. On note σ^i le chemin extrait de σ à partir de la i-ème position. Par exemple $\sigma^2 = s_2 s_3 \ldots s_n \ldots$. On note $\sigma(k)$ le k-ième état de σ. On se donne une fonction $l : Q \to 2^{\mathrm{AP}}$ indiquant quelles propriétés $p \in \mathrm{AP}$ sont vraies dans un état $s \in Q$.

On définit ensuite récursivement la relation de satisfaction $\sigma \models \varphi$ par :

. $\sigma \models p$ iff $p \in l(\sigma(0))$ (p vrai dans l'état courant),
. $\sigma \models \neg\varphi$ iff $\sigma \not\models \varphi$,
. $\sigma \models \varphi_1 \vee \varphi_2$ iff $\sigma \models \varphi_1$ ou $\sigma \models \varphi_2$,
. $\sigma \models \varphi_1 \wedge \varphi_2$ iff $\sigma \models \varphi_1$ et $\sigma \models \varphi_2$,
. $\sigma \models \mathbf{X}\varphi$ iff $\sigma^1 \models \varphi$,
. $\sigma \models \mathbf{F}\varphi$ iff il existe $k \geq 0$ tel que $\sigma^k \models \varphi$,
. $\sigma \models \mathbf{G}\varphi$ iff pour tout $k \geq 0$ on a $\sigma^k \models \varphi$,

[3] Qui permettent une définition très élégante de la notion de sûreté.

. $\sigma \models \varphi_1 \mathbf{U} \varphi_2$ iff il existe $k \geq 0$ tel que $\sigma^k \models \varphi_2$ et pour tout $0 \leq j < k$, $\sigma^j \models \varphi_1$.

Exercice 15. *On va voir que certains connecteurs sont redondants.*
- *Exprimer* $\mathbf{G}p$ *avec les connecteurs* $\neg, \mathbf{F}$ *et* p.
- *Exprimer* $\mathbf{F}p$ *grâce au connecteur* $\mathbf{U}$.
- *Peut-on exprimer* $\mathbf{X}$ *en fonction des autres connecteurs ?*
- *Peut-on exprimer* $\mathbf{U}$ *en fonction des autres connecteurs ?*
- *Comparez alors LTL, LTL-F, LTL-U et LTL-X.*
- *Donner une grammaire minimale pour LTL.*

Exercice 16 (Autres connecteurs.)**.** *On va définir quelques connecteurs additionels utiles.*

1. *Définir la relation* $\models$ *pour les connecteurs additionnels suivants :*
- $p\mathbf{W}q$ *(weak until) : signifie que p est vrai jusqu'à ce que q soit vrai, mais q n'est pas forcément vrai à un moment. Dans ce cas, p reste vrai tout le long du chemin.*
- $\mathbf{F}^{\infty}p$ *(infiniment souvent) : p est infiniment vrai au long de l'exécution.*
- $\mathbf{G}^{\infty}p$ *(presque toujours) : à partir d'un moment donné, p est toujours vrai.*
- $p\ \mathbf{U}_{\leq \mathbf{k}}q$ *(bounded until) : p vrai jusqu'à ce que q soit vrai, et q vrai dans au plus k observations.*
- $p\mathbf{R}q$ *(release) : q est vraie jusqu'à (et inclus) le premier état où p est vraie, sachant que p n'est pas forcément vraie un jour.*

2. *On va maintenant faire le lien entre ces connecteurs et les anciens.*
- *Exprimer* $\mathbf{F}^{\infty}$ *et* $\mathbf{G}^{\infty}$ *par des connecteurs de LTL.*
- *Que pensez-vous de LTL-*$\mathbf{U}$*+*$\mathbf{W}$ *?*
- *Que pensez-vous de LTL et LTL+*$\mathbf{U}_{\leq \mathbf{k}}$ *(pour toute valeur de k) ? Quel lien entre* $\mathbf{U}_{\leq \mathbf{k}}$ *et* $\mathbf{U}$ *?*
- *Exprimer* $\mathbf{G}$ *et* $\mathbf{U}$ *en fonction de* $\mathbf{R}$*. Qu'en déduire sur LTL+*$\mathbf{R}$*, LTL et LTL+*$\mathbf{R}$*-*$\mathbf{U}$ *?*

Exercice 17. *Exprimer en langage naturel les propriétés de chemin suivantes.*
- $\mathbf{G}(emission \rightarrow \mathbf{F}reception)$
- $\mathbf{F}^{\infty} ok \rightarrow \mathbf{G}(emission \rightarrow \mathbf{F}reception)$

Exercice 18. *Parmi les opérateurs suivants, lesquels correspondent à des propriétés de sûreté ?*
$\mathbf{X}, \mathbf{F}, \mathbf{G}, \mathbf{U}, \mathbf{W}, \mathbf{U}_{\leq \mathbf{k}}, \mathbf{R}, \mathbf{F}^{\infty}, \mathbf{G}^{\infty}$.

Exercice 19 (*)**.** *Montrez que* $\mathbf{U}$ *n'est pas associatif.*

Exercice 20 (Opérateurs du passé (**))**.** *On va définir des opérateurs du passé.*

1. *Modifier la définition de la logique et de la sémantique pour prendre en compte les opérateurs du passé* $\mathbf{X}^{-1}$,$\mathbf{F}^{-1}$, $\mathbf{G}^{-1}$ *et* $\mathbf{U}^{-1}$.
2. *Montrer que LTL+(opérateurs du passé) est équivalent à LTL.*
3. *Comparez la concision des deux logiques.*

Exercice 21 (Théorème de Kamp (*))**.** *Ce résultat est le sens (facile) du théorème de Kamp, qui établit que LTL a même pouvoir d'expression que la logique monadique du premier ordre à un successeur.*

On se donne un ensemble AP de prédicats atomiques $P : \mathbb{N} \rightarrow \mathcal{B}$ *et un ensemble Var de variables. On considère la logique monadique du premier ordre à un successeur, définie par :*

. $t ::= 0 | v \in Var | t+1$

. $atome ::= t \geq t | t = t | P(t), P \in AP$

. $f ::= f \vee f | f \wedge f | \neg f | \exists v, f | atome$

La logique est interprétée sur $\mathbb{N}$*. En considérant que* $P(t)$ *signifie que la propriété* P *est vrai au temps* t *(=* P *vrai à la t-ième étape du chemin), donner une traduction (récursive) des formules LTL* φ *en formules* $\widetilde{\varphi}$ *de logique monadique ayant même signification.*

Jusqu'ici on a défini LTL sur un unique chemin. On étend les définitions pour une structure de Kripke $\mathcal{M} = \langle Q, \rightarrow, P, l, s_0 \rangle$ comme suit.

Définition 3.3.2 (LTL sur une structure de Kripke)**.** Les formules LTL sur une structure de Kripke sont de la forme $\mathbf{A}f$ où f est une formule LTL de chemin. La formule est satisfaite si tous les chemins de la structure de Kripke satisfont f. Cela donne la grammaire suivante :

$\varphi_s ::= \mathbf{A}\varphi_p$

$\varphi_p ::= p \in \mathrm{AP} — \neg\varphi_p — \varphi_p \vee \varphi_p — \varphi_p \wedge \varphi_p — \mathbf{X}\varphi_p — \mathbf{F}\varphi_p — \mathbf{G}\varphi_p — \varphi_p\mathbf{U}\varphi_p$

Satisfaction. Le domaine d'interprétation est maintenant un couple $(\mathcal{M}, s)$ associant une structure de Kripke et un état. On définit alors récursivement la relation de satisfaction $\mathcal{M}, s \models \varphi$ par :

. $\mathcal{M}, s \models \mathbf{A}\varphi_p$ ssi tous les chemins σ partant de s vérifient $\sigma \models \varphi_p$,

. $\sigma \models \varphi_p$ est défini comme précédemment.

On vient de définir la relation $\mathcal{M}, s \models \varphi$. Ce n'est pas exactement ce qu'on veut, puisqu'on s'intéresse uniquement à des structures de Kripke, pas à des couples. On définit alors $\mathcal{M} \models \varphi$ par $\mathcal{M}, s_0 \models \varphi$.

Exercice 22.

1. *Soit $\varphi = \mathbf{A}f$ une formule de LTL. $\neg\varphi$ est-il exprimable en LTL ? On dit que LTL n'est pas clos par négation. Était-ce le cas pour LTL sur les chemins ? Quelle est la différence ?*
2. *Peut-on avoir à la fois $\mathcal{M} \not\models \mathbf{A}f$ et $\mathcal{M} \not\models \mathbf{A}\neg f$? Si oui, donner un exemple, sinon prouvez le.*
3. *Quel connecteur faudrait-il rajouter à LTL pour obtenir la clôture par négation ? Écrivez la nouvelle grammaire et le relation de satisfaisabilité correspondante.*

Exercice 23. *Montrer que si on ajoute $\mathbf{R}$ aux connecteurs de bases, on peut restreindre l'emploi de la négation aux propriétés atomiques. Cette propriété est très importante, car la gestion de la négation non atomique est très coûteuse dans les algorithmes de model checking.*

Exercice 24 (Équivalence comportementale (*).)**.** *Que peut-on dire de deux structures de Kripke $\mathcal{M}_1$ et $\mathcal{M}_2$ telles que $\mathcal{L}(\mathcal{M}_1) = \mathcal{L}(\mathcal{M}_2)$? Que peut-on dire de deux structures de Kripke $\mathcal{M}_1$ et $\mathcal{M}_2$ telles que $\mathcal{L}(\mathcal{M}_1) \subseteq \mathcal{L}(\mathcal{M}_2)$?*

Exercice 25 (Stuttering equivalence.)**.** *à faire.*

Exercice 26 (**)**.** *Montrer que le problème du model checking de LTL se ramène au problème de la validité de LTL. Plus précisément, on se donne une formule LTL φ et une structure de Kripke $\mathcal{M} = \langle Q, \rightarrow, P, l, s_0 \rangle$. On va construire une formule φ'' telle que $\mathcal{M}, s_0 \models \varphi$ ssi φ'' est valide.*

1. *Commencez par construire une formule φ' telle que $\sigma \models \varphi'$ ssi $\sigma \in \mathcal{L}(\mathcal{M})$.*
2. *Concluez en construisant la formule φ'' cherchée à partir de φ' et φ.*

3.4 Logique branchante CTL*

La logique CTL* est la plus expressive que nous verrons. Par rapport à LTL, elle ne restreint pas l'emploi des quantificateurs de chemin.

Définition 3.4.1 (logique CTL*)**.** On distingue dans CTL* des formules d'état (φ_s), interprétées sur les états de la structure de Kripke, et des formules de chemin (φ_p), interprétées sur les chemins de la structure de Kripke. Les formules CTL* sont alors définies par la grammaire suivante :

$\varphi_s ::= p \in \mathrm{AP} — \neg\varphi_s — \varphi_s \vee \varphi_s — \varphi_s \wedge \varphi_s — \mathbf{A}\varphi_p — \mathbf{E}\varphi_p$

$\varphi_p ::= \varphi_s — \neg\varphi_p — \varphi_p \vee \varphi_p — \varphi_p \wedge \varphi_p — \mathbf{X}\varphi_p — \mathbf{F}\varphi_p — \mathbf{G}\varphi_p — \varphi_p\mathbf{U}\varphi_p$

Satisfaction. Le domaine d'interprétation est encore un couple $(\mathcal{M}, s)$ associant une structure de Kripke et un état. On définit alors $\mathcal{M}, s \models \varphi$ par :

formules d'état, relation $\models_s$

. $\mathcal{M}, s \models_s p \in \mathrm{AP}$ iff $p \in l(s)$ (p vrai dans l'état courant),
. $\mathcal{M}, s \models_s \neg\varphi$ iff $\mathcal{M}, s \not\models_s \varphi$,
. $\mathcal{M}, s \models_s \varphi_1 \vee \varphi_2$ iff $\mathcal{M}, s \models_s \varphi_1$ ou $\mathcal{M}, s \models_s \varphi_2$,
. $\mathcal{M}, s \models_s \varphi_1 \wedge \varphi_2$ iff $\mathcal{M}, s \models_s \varphi_1$ et $\mathcal{M}, s \models_s \varphi_2$,
. $\mathcal{M}, s \models_s \mathbf{A}f$ ssi tous les chemins σ partant de s vérifient $\mathcal{M}, \sigma \models_p f$,
. $\mathcal{M}, s \models_s \mathbf{E}f$ ssi il existe un chemins σ partant de s vérifiant $\mathcal{M}, \sigma \models_p f$,

formules de chemin, relation $\models_p$

. $\mathcal{M}, \sigma \models_p f$ iff $\mathcal{M}, \sigma(0) \models_s f$ (f formule d'état),
. $\mathcal{M}, \sigma \models_p \neg\varphi$ iff $\mathcal{M}, \sigma \not\models_p \varphi$,
. $\mathcal{M}, \sigma \models_p \varphi_1 \vee \varphi_2$ iff $\mathcal{M}, \sigma \models_p \varphi_1$ ou $\mathcal{M}, \sigma \models_p \varphi_2$,
. $\mathcal{M}, \sigma \models_p \varphi_1 \wedge \varphi_2$ iff $\mathcal{M}, \sigma \models_p \varphi_1$ et $\mathcal{M}, \sigma \models_p \varphi_2$,
. $\mathcal{M}, \sigma \models_p \mathbf{X}\varphi$ iff $\mathcal{M}, \sigma^1 \models_p \varphi$,
. $\mathcal{M}, \sigma \models_p \mathbf{F}\varphi$ iff il existe $k \geq 0$ tel que $\mathcal{M}, \sigma^k \models_p \varphi$,
. $\mathcal{M}, \sigma \models_p \mathbf{G}\varphi$ iff pour tout $k \geq 0$ on a $\mathcal{M}, \sigma^k \models_p \varphi$,
. $\mathcal{M}, \sigma \models_p \varphi_1 \mathbf{U} \varphi_2$ iff il existe $k \geq 0$ tel que $\mathcal{M}, \sigma^k \models_p \varphi_2$ et pour tout $0 \leq j < k$, $\mathcal{M}, \sigma^j \models_p \varphi_1$.

Là encore on définit $\mathcal{M} \models \varphi$ en partant de l'état initial, avec $\mathcal{M} \models \varphi$ si $\mathcal{M}, s_0 \models_s \varphi$.

Exercice 27. *Exprimer toutes les propriétés de la section 3.1 en tenant compte des quantificateurs de chemin.*

Exercice 28.

1. *Est-ce que CTL* est clos par négation ?*
2. *Montrer que* $\vee$, $\neg$, $\mathbf{X}$, $\mathbf{U}$ *et* $\mathbf{E}$ *suffisent à exprimer les autres connecteurs.*
3. *Restreignez la grammaire de CTL* (les* φ_p *ou les* φ_s*) pour retrouver LTL (avec la même interprétation).*
4. *Montrez que si on ajoute* $\mathbf{R}$*, on peut restreindre* $\neg$ *aux propositions atomiques.*

Exercice 29. *On définit la logique ACTL* comme la restriction de CTL* dans laquelle le seul quantificateur permis est* $\mathbf{A}$*, et la négation ne peut intervenir qu'au niveau des propositions atomiques. Ainsi,* $AG\neg p$ *est une formule de ACTL*, mais pas* EFp *ni* $A\neg Gp$*. On ajoute* $\mathbf{R}$ *aux connecteurs de base.*

1. *Comparez ACTL* et ACTL*-*$\mathbf{R}$.
2. *Montrer que* $LTL \subseteq ACTL^*$. (au sens sémantique)
3. *Est-ce que* $AFAGp \equiv AFGp$ *?*
4. *Que déduire sur LTL et ACTL* ?*

Remarque : ACTL est le fragment de CTL* bien adapté à la vérification modulaire.*

Exercice 30 (Bisimulation et simulation.)**.** *à faire.*

Exercice 31 (ACTL* et assume-guarantee)**.** *à faire.*

Exercice 32 (Stuttering bisimulation)**.** *à faire.*

3.5 Logique branchante CTL

On définit maintenant la logique CTL (Computation tree logic), qui est une restriction de CTL* dans laquelle les connecteurs temporels $\mathbf{X}, \mathbf{F}, \mathbf{G}$ et $\mathbf{U}$ doivent être directement précédés d'un quantificateur de chemin $\mathbf{A}$ ou $\mathbf{E}$. Par exemple, $AFAGp$ est une formule CTL, mais pas $AFGp$.

Une formule CTL s'obtient donc à partir de $\neg, \vee, \wedge$ et des huit opérateurs suivants :

- **AX** et **EX**,
- **AF** et **EF**,
- **AG** et **EG**,
- **AU** et **EU**.

CTL est un fragment très important de CTL*, car il est raisonnablement expressif et il existe des algorithmes de model-checking polynômiaux. Comme dans le cas de CTL*, on peut définir la restriction ACTL bien adaptée au calcul modulaire.

Exercice 33. *Montrer que p, $\vee$, $\neg$,* **EX**, **EG** *et* **EU** *suffisent à exprimer les autres connecteurs. Montrer ensuite que p, $\wedge$, $\neg$,* **EX**, **AU** *et* **EU** *suffisent aussi.*

Exercice 34. *Ce n'est pas parcequ'une formule n'est pas syntaxiquement dans CTL qu'il n'y a pas de formule CTL équivalente. Tranformez les formules suivantes en formules CTL :* $\mathbf{E}(p \wedge \mathbf{F}q)$, $\mathbf{AGF}p$.

Peut-on exprimer les notions suivantes en CTL : **AW**, **EW**, $\mathbf{AU}_{\leq \mathbf{k}}$, $\mathbf{EU}_{\leq \mathbf{k}}$ *?*

Peut-on exprimer les notions suivantes en CTL : $\mathbf{AG}^\infty$, $\mathbf{EG}^\infty$ *?*

Équité : peut-on exprimer $\mathbf{AF}^\infty\varphi$ *en CTL ? et* $\mathbf{EF}^\infty\varphi$ *?*

Exercice 35 (CTL et CTL+)**.** *a faire*

Exercice 36 (CTL et équité (*))**.** *On peut essayer d'étendre CTL pour lui ajouter l'équité. Il y a plusieurs manières de faire.*

- *fair CTL : on fait du CTL mais sur des structures fair, comme définies au chapitre précédent*[4].
- *ECTL : on ajoute* $\mathbf{AF}^\infty$ *et* $\mathbf{EF}^\infty$

Questions :

1. *Quel lien entre CTL et ECTL ?*
2. *Quel genre de formules permet de vérifier fair CTL ? Déduire le lien entre CTL et fair CTL, puis le lien entre ECTL et fair CTL.*

Exercice 37 (CTL et bisimulation)**.** *a faire*

Exercice 38 (ACTL et assume-guarantee)**.** *a faire*

3.6 Comparaison des trois logiques

La figure 3.9 donne quelques résultats de complexité sur les différentes logiques, et la figure 3.10 indique les relations d'inclusion entre les logiques LTL, CTL, CTL* et ACTL*.

	CTL	LTL	CTL*
MC	P-complet	PSPACE-complet	PSPACE-complet
MC concurrent	PSPACE-complet	PSPACE-complet	PSPACE-complet
MC open	EXPTIME-complet	PSPACE-complet	2-EXPTIME-complet
satisfaisabilité	EXPTIME-complet	PSPACE-complet	2-EXPTIME-complet

MC : est-ce que $\mathcal{M} \models \varphi$?
MC concurrent : est-ce que $\mathcal{M}_1, \ldots, \mathcal{M}_n \models \varphi$?
MC open : est-ce que $\mathcal{M}, E \models \varphi$ pour tout E tel que $E \models \psi$, où ψ est la contrainte d'environnement ?
Satisfaisabilité : est-ce qu'il existe $\mathcal{M}$ tel que $\mathcal{M} \models \varphi$?

Fig. 3.9 – Quelques résultats de complexité

[4]Ce que je note fair CTL est parfois aussi appelé CTL^F

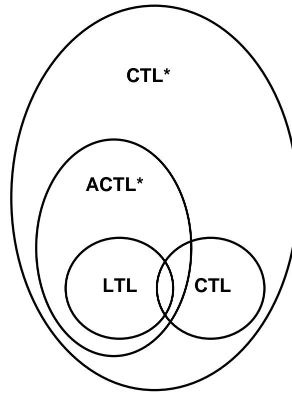


Fig. 3.10 – Hiérarchie des logiques temporelles vues en cours.

Exercice 39. *Quand une logique est incluse dans une autre, dîtes pourquoi. Quand deux logiques sont distinctes, trouvez une formule expressible dans l'une et pas dans l'autre.*

Exercice 40 (Questions pratiques)**.**
(1) Quelles logiques ci-dessus sont capables d'exprimer la notion d'équité ?
(2) Comment vérifieriez-vous les propriétés suivantes si vous aviez un model checker pour CTL, LTL, CTL* *?*
- *accessibilité,*
- *invariance,*
- *sûreté,*
- *vivacité,*
- *absence de deadlock,*
- *équité,*
- *équivalence de modèle.*

Exercice 41 (Pouvoir de séparation.)**.** *a faire.*

Comparaisons des logiques. Les logiques linéaires comme LTL s'avèrent plus naturelles en pratique pour spécifier les comportements attendus du système ou de l'environnement. De plus elles permettent d'exprimer des notions utiles comme l'équité, les contre-exemples retournés sont simples (trace d'exécution) et ces logiques sont adaptées à la vérification à l'exécution. Par contre LTL manque parfois d'expressivité (ex : *"on peut toujours revenir à l'état initial"*).

À l'inverse les logiques branchantes s'avèrent parfois contre-intuitives (environnement, équivalence de modèle), les contre-exemples retournés sont difficiles à interpréter (arbres d'exécution) et la vérification à l'exécution n'a pas de sens. CTL a cependant l'énorme avantage d'avoir des algorithmes de model checking polynômiaux, alors que ceux de LTL sont PSPACE donc exponentiels en pratique. CTL peut aussi exprimer certaines propriétés utiles absentes de LTL et l'équité peut s'obtenir avec des algorithmes ad hoc.

La logique CTL* est très expressive et le model checking a la même complexité que pour LTL. On pourrait donc se dire autant utiliser CTL*, quitte à se restreindre à des formules "humainement compréhensibles". Cependant, d'une part le gain d'expressivité n'est plus gratuit si on considère un environnement, d'autres part ce n'est pas certain que ce gain d'expressivité soit utile, puisqu'on peut ajouter l'équité à CTL au niveau des algorithmes, et qu'on peut toujours utiliser un model checker de LTL et un autre de CTL pour les quelques propriétés manquantes.

Une voie raisonnable d'utilisation des model-checkers semble être d'utiliser CTL pour vérifier les propriétés les plus simples (sûreté) sur tout le modèle et d'en éprouver la validité ; puis après que quelques bugs grossiers aient été trouvés et que le modèle soit validé, utilisé un model checker LTL sur une partie seulement du système (*bounded model checking*).

Autres logiques. D'autres logiques ont été développées. Nous en mentionnons seulement trois. Le μ-calcul est le formalisme de spécification le plus puissant, mais il est difficile d'accès et peu concis. Les propriétés w-régulières sont assez intuitives (basées sur des automates) et très expressives. Il existe des versions branchantes ou linéaires. Les algorithmes de model checking basés sur les automates travaillent en général non pas sur LTL ou CTL* mais sur des propriétés w-régulières linéaires ou branchantes. Enfin les *Hierarchical Message Sequence Charts (HMSC)* sont des spécifications provenant d'UML *(Unified Modeling Language)* et empruntées au monde des protocoles de télécommunications. Ces spécifications sont simples à comprendre à première vue (graphiques), mais leurs liens avec les autres logiques temporelles ne sont pas encore claires.

Chapitre 4

Model checking, algorithmes de base

Nous présentons dans ce chapitre des algorithmes de model checking pour CTL et LTL. On présentera deux algorithmes pour CTL, d'abord l'algorithme standard à base d'étiquetage *(labelling)* des états puis une extension pour l'équité. Ensuite on présentera un algorithme pour LTL à base d'automates.

4.1 Prélude : composantes fortement connexes

On commence par un problème algorithmique sous-jacent au model checking de CTL et de LTL : la recherche de composantes fortement connexes d'un graphe. Formellement un graphe orienté G est une paire $G = \langle Q, T \rangle$ où Q est un ensembles d'états (ou nœuds) et $T \subseteq Q \times Q$ un ensemble de transitions. Comme d'habitude, $(q, q') \in T$ signifie qu'on peut aller de q à q' en prenant une transition du graphe. On dit que q' est atteignable à partir de q, noté $q \xrightarrow{*} q'$ si il existe un chemin (une suite de transitions) de q à q'.

Définition 4.1.1. Soit $G = \langle Q, T \rangle$ un graphe orienté. On appelle composante connexe de G tout sous ensemble non vide $C \subseteq Q$ tel que pour tout $c_i, c_j \in C$, c_j est atteignable à partir de c_i *en restant dans* C.

Une composante fortement connexe est une composante connexe maximale : si on lui rajoute un nouveau nœud, elle n'est plus connexe. Un composante fortement connexe non triviale a soit au moins deux nœuds, soit un seul nœud avec une transition sur lui-même, c-à-d $(q, q) \in T$. Autrement dit une composante fortement connexe non triviale a au moins un arc. On s'intéresse à la décomposition en composantes fortement connexes non triviales d'un graphe orienté.Un exemple de décomposition est donné à la figure 4.1.

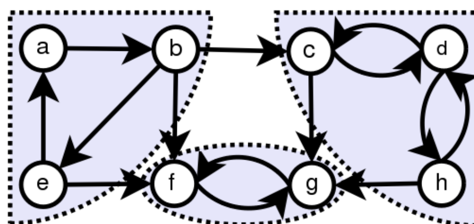


Fig. 4.1 – Exemple de décomposition en composantes fortement connexes

Le lien avec CTL et LTL est le suivant : intuitivement, pour qu'une propriété du type $\varphi = \mathbf{EG}p$ soit vraie, comme on s'intéresse à des chemins *infinis* et que la structre de Kripke $\mathcal{M}$ est finie, la seule manière qu'a un état s_0 de satisfaire φ est de pouvoir atteindre par un chemin *fini* d'états vérifiant p une composante fortement connexe C dont tous les états satisfont p. Le chemin vérifiant φ à partir de s_0 est donc la forme

$s_0 s_1 \dots s_n \sigma_C$ où $s_0 s_1 \dots s_n$ est un chemin fini d'états satisfaisant p et σ_C est un chemin infini restant dans C (où tous les états satisfont p).

La section C.1 en annexe présente deux algorithmes de découverte des composantes fortement connexes : l'algorithme de Kosaraju et l'algorithme de Tarjan. Ce dernier est le plus rapide en théorie, mais il est un peu délicat à bien implanter.

Dans la suite on écrira "composante fortement connexe" en lieu et place de "composante fortement connexe non triviale".

4.2 Model checking de CTL par labelling

C'est le premier algorithme de model checking a avoir été développé. Son avantage majeur est de tourner en temps linéaire en chacune des entrées (la structure de Kripke et la formule). L'algorithme repose sur le fait que toute formule CTL peut s'exprimer par un nombre restreint de formules sur les états. Cela nous permet de raisonner en termes d'états (satisfaisant la formule) plutôt que d'exécutions.

Principe. L'algorithme prend en entrée la structure de Kripke $\mathcal{M} = \langle Q, \rightarrow, P, l, s_0 \rangle$ et une formule CTL φ. Il est à base de marquage : pour chaque sous-formule φ' de φ, en commençant par la plus interne, on va marquer les états s de $\mathcal{M}$ qui vérifient φ'. On procède ensuite récursivement en réutilisant les marquages des sous-formules plus internes pour une sous-formule plus externe. Finalement, $\mathcal{M}$ satisfait φ ssi l'état initial s_0 est marqué par φ.

Par exemple pour la formule $\neg \mathbf{EX} p$, on procède ent trois phases : (1) on marque les états Q_p vérifiant p, cette information est donnée par la fonction l de $\mathcal{M}$; (2) on marque les états $Q_{\mathbf{EX}p}$ vérifiant $\mathbf{EX}p$, ce sont ceux dont un successeur par $\rightarrow$ est dans Q_p. (3) on marque les états Q_φ vérifiant $\neg \mathbf{EX} p$, ce sont ceux qui ne sont pas dans $Q_{\mathbf{X}p}$.

Pour limiter le nombre de cas à traiter dans l'algorithme et dans les preuves, on va se restreindre aux connecteurs $p, \wedge, \neg, \mathbf{EX}, \mathbf{EU}$ et $\mathbf{AU}$. Avant d'appliquer l'algorithme, on passera donc d'abord par une phase de traduction de la formule φ.

Le schéma global de l'algorithme est le suivant. La procédure MARKING est détaillée après.

Algorithme MC-CTL
input : φ, $\mathcal{M} = \langle Q, \rightarrow, P, l, s_0 \rangle$
1: $\varphi' :=$ NORMALIZE(φ) ;
2: Do MARKING(φ',$\mathcal{M}$) ;
3: RETURN $s_0.\varphi'$

Algorithme 1: Model Checking de CTL par étiquetage

Théorème 4.2.1 (Correction)**.** *L'algorithme* MC-CTL *termine et retourne vrai ssi* $\mathcal{M} \models \varphi$.

On s'intéresse maintenant à la complexité de l'algorithme. Pour cela on va définir la taille d'une structure de Kripke $|\mathcal{M}|$ comme la somme de son nombre d'états $|Q|$ et son nombre de transitions $|\rightarrow|$, et la taille d'une formule $|\varphi|$ comme son nombre de connecteurs. On a alors le résultat suivant.

Théorème 4.2.2 (Complexité)**.** *L'algorithme 1 est linéaire en la taille de la formule et la taille de la structure de Kripke, soit* $\mathcal{O}(|\mathcal{M}| \cdot |\varphi|)$.

Exercice 42. *On a ici utilisé un nombre restreint de connecteurs, et la formule initiale* φ_0 *est déjà transformée en une formule* φ' *n'utilisant que ces connecteurs.*

Algorithme MARKING
input : formule φ normalisée, $\mathcal{M} = \langle Q, \rightarrow, P, l, s_0 \rangle$

1:
2: Case 1 : $\varphi = p$
3: for all s $\in Q$ do
4:     if $p \in l(s)$ then s.φ := true
5:     else s.φ := false
6: end for
7:
8: Case 2 : $\varphi = \neg\varphi'$
9: do MARKING(φ',$\mathcal{M}$) ;
10: for all s $\in Q$ do s.φ := not(s.φ') end for
11:
12: Case 3 : $\varphi = \varphi' \wedge \varphi''$
13: do MARKING(φ',$\mathcal{M}$) ; MARKING(φ'',$\mathcal{M}$) ;
14: for all s $\in Q$ do s.φ := and(s.φ',s.φ'') end for
15:
16: Case 4 : $\varphi = \mathbf{EX}\varphi'$
17: do MARKING(φ',$\mathcal{M}$) ;
18: for all s $\in Q$ do s.φ := false end for
19: for all $(s, s') \in\rightarrow$ do
20:     if s'.φ'=true then s.φ := true
21: end for
22:
23: Case 5 : $\varphi = \mathbf{E}\varphi'\mathbf{U}\varphi''$
24: do MARKING(φ',$\mathcal{M}$) ; MARKING(φ'',$\mathcal{M}$) ;
25: for all s $\in Q$ do
26:     s.φ := false ;
27:     s.seenbefore := false
28: end for
29: L := $\emptyset$
30: for all s $\in Q$ do if s.φ''=true then L :=L + {s} end for
31: while L $\neq \emptyset$ do
32:     choose s $\in$ L ; L := L - {s} ;
33:     s.φ := true ;
34:     For all (s',s) $\in\rightarrow$ do // s' predecessor of s
35:         if s'.seenbefore = false then
36:             s'.seenbefore := true ;
37:             if s'.φ' = true then L := L + {s'} ;
38:         end if
39:     end for
40: end while
41:
42: Case 6 : $\varphi = \mathbf{A}\varphi'\mathbf{U}\varphi''$
43: Do MARKINGAU(φ,$\mathcal{M}$) ;

Algorithme 2: Model Checking de CTL par étiquetage, cas simples

Algorithme MARKINGAU
entrée : φ, $\mathcal{M} = \langle Q, \rightarrow, P, l, s_0 \rangle$

```
1:
2: Case 6 : φ = Aφ'Uφ''
3: fo MARKING(φ',M) ; MARKING(φ'',M) ;
4: L := ∅ ;
5: for all s ∈ Q do
6:    s.nb := degree(s) ; s.φ := false ;
7:    if s.φ'' = true then L := L + {s} ;
8: end for
9: while L ≠ ∅ do
10:    choose s ∈ L ; L := L - {s} ;
11:    s.φ := true ;
12:    for all (s',s) ∈→ do // s' predecessor of s
13:       s'.nb := s'.nb - 1 ;
14:       if (s'.nb = 0) and (s'.φ' = true) and (s'.φ = false) do
15:          L := L + {s'} ;
16:       end if
17:    end for
18: end while
19:
```

Algorithme 3: Model Checking de CTL par étiquetage, cas $\mathbf{A}\varphi_1\mathbf{U}\varphi_2$

- *Rappelez pourquoi on peut se permettre de restreindre les opérateurs.*
- *Quel est l'avantage de cette méthode ? Quel est le désavantage ?*
- *Y a-t-il un saut de complexité ?*

Exercice 43 (*)**.** *Faire les preuves de correction et de complexité.*

On a vu qu'il pouvait être intéressant d'implanter d'autres connecteurs directement dans l'algorithme. Par exemple pour **EG** on peut utiliser l'algorithme 4 à base de recherche de composantes fortement connexes.

Exercice 44 (*)**.** *Modifier l'algorithme pour gérer tous les cas suivants :* $\neg p$, $\wedge$, **AX**, **AG**, **AF**, **EF**, **AR**, **ER**, **AW**, **EW**, $\mathbf{AU}_{\leq \mathbf{k}}$, $\mathbf{EU}_{\leq \mathbf{k}}$. *Adaptez les preuves de correction et de complexité.*

Exercice 45 (Model checking de ECTL)**.** *à faire*

4.3 Model checking de fair CTL par labelling

On a vu que CTL n'était pas assez expressive pour exprimer les contraintes d'équités. Cela pose un gros problème pratique, car souvent on ne veut vérifier une propriété que sur des chemins *fair*, sans tenir compte des chemins *aberrants* jugés très peu probables en pratique. Par exemple, pour un protocole de communication, on ne veut pas considérer les chemins où les messages sont systématiquement perdus.

En LTL ou CTL* ce type de propriété s'exprimerait naturellement en utilisant une implication, par exemple $\varphi = \mathbf{A}(\textit{fairness} \rightarrow \textit{propriété})$. En CTL, on ne peut pas exprimer l'équité, mais on peut ruser en modifiant l'algorithme MARKING de telle manière qu'il ne prenne en compte que les chemins passant infiniment souvent par certains états spécifiés par l'utilisateur. *Ainsi, on n'agit pas au niveau de la formule mais on change plutôt la sémantique du modèle.* On peut ainsi simuler des formules d'équité du type *"tels états sont infiniment souvent visités"*, qui suffisent habituellement en pratique. On obtient alors un algorithme FAIR-MARKING, qui prend en entrée $\mathcal{M}$, φ et un ensemble F d'ensembles d'états $F_i \subseteq Q$ appelé contraintes d'équité.

Algorithme MARKING-EG
entrée : $\varphi = \mathbf{EG}\varphi'$, $\mathcal{M} = \langle Q, \rightarrow, P, l, s_0 \rangle$

```
 1:
 2: Q' := {s | φ' ∈ l(s) } ;
 3: SCC := { C | C non trivial SCC of Q' } ;
 4: L := ⋃_{C∈SCC} { s | s ∈ C } ;
 5: for all s ∈ L do s.φ := true end for
 6: while L ≠ ∅ do
 7:    choose s ∈ L ; L := L - {s} ;
 8:    for all (s',s) ∈→ such that s' ∈ Q' do
 9:       if (s'.φ = false) then
10:          s'.φ := true ;
11:          L := L + {s'} ;
12:       end if
13:    end for
14: end while
15:
```

Algorithme 4: Model Checking de CTL par étiquetage, cas $\mathbf{EG}\varphi$

Dans la suite, on distingue une nouvelle relation de satisfaisabilité $\mathcal{M} \models_F \varphi$ signifiant *"si on se restreint aux chemins fair, $\mathcal{M}$ satisfait φ"*. Formellement, $\models_F$ est défini comme $\models$, sauf pour les propositions de base p, et les quantificateurs $\mathbf{A}, \mathbf{E}$. Ainsi,

- $\mathcal{M}, s \models_F p$ ssi il existe un chemin fair partant de s et $p \in l(s)$;
- $\mathcal{M}, s \models_F \mathbf{E}f$ ssi il existe un chemin fair σ partant de s et $\mathcal{M}, \sigma \models_F f$;
- $\mathcal{M}, s \models_F \mathbf{A}p$ ssi tous les chemins fair σ partant de s vérifient $\mathcal{M}, \sigma \models_F f$.

Fair SCC. Soit une structure de Kripke $\mathcal{M}$ et une contrainte d'équité sur les états $F = \{F_1, \ldots, F_n\}$. On dira qu'une composante fortement connexe C de $\mathcal{M}$ est fair (par rapport à F) si pour chaque F_i, il y a au moins un état commun à C et à F_i.

Algorithme. L'idée de l'algorithme FAIR-MARKING est la suivante :

. on définit d'abord une procédure FAIR-MARKING-EG qui marque les états s tels que $\mathcal{M}, s \models_F \mathbf{EG}f$, en supposant que les états sont déjà marqués pour $\mathcal{M}, s \models_F f$. Cette procédure est très proche de MARKING-EG, la seule différence est de considérer les fair SCC plutôt que les SCC.

. on utilise FAIR-MARKING-EG pour marquer les états s à partir desquels partent des chemins fair. Pour cela on ajoute une nouvelle proposition atomique *fair* valant $\mathbf{EG}$true *en sémantique fair*, et on utilise FAIR-MARKING-EG.

. finalement, on réutilise les anciens algorithmes de marquages pour les autres connecteurs, en remarquant que $\mathcal{M}, s \models_F p$ ssi $\mathcal{M}, s \models p \wedge$fair, $\mathcal{M}, s \models_F \mathbf{EX}\varphi$ ssi $\mathcal{M}, s \models \mathbf{EX}(\varphi \wedge$fair) et $\mathcal{M}, s \models_F \mathbf{E}\varphi_1\mathbf{U}\varphi_2$ ssi $\mathcal{M}, s \models \varphi_1\mathbf{EU}(\varphi_2 \wedge$ fair).

Théorème 4.3.1 (Correction et complexité)**.** *L'algorithme* FAIR-MARKING *termine et retourne vrai ssi $\mathcal{M}, s \models_F \varphi$. L'algorithme est en $\mathcal{O}(|\mathcal{M}| \cdot |\varphi| \cdot |F|)$.*

Exercice 46. *Terminer la preuve de correction. Notamment expliquez pourquoi on peut utiliser* FAIR-MARKING-EG *pour étiqueter* $\mathbf{EG}$*true* en sémantique fair *(point 2), et prouver les équivalences données pour passer de $\models_F$ à $\models$. Exprimer pour tous les connecteurs CTL la relation $\mathcal{M}, s \models_F \varphi$ en fonction de $\models$ et fair.*

Enfin essayer de réexprimer l'algorithme général plus simplement, en ramenant $\mathcal{M}, s \models_F \varphi$ à $\mathcal{M}', s \models \varphi$, où $\mathcal{M}'$ est une autre structure de Kripke et φ est la même formule, sans fair.

Algorithme FAIR-MARKING-EG
entrée : $\varphi = \mathbf{EG}\varphi'$, $\mathcal{M} = \langle Q, \rightarrow, P, l, s_0\rangle$, $F = \{F_1, \ldots, F_n\}$

```
1:
2: Q' := {s | φ' ∈ l(s) } ;
3: SCC := { C | C non trivial fair SCC of Q' } ;
4: L := ⋃_{C∈SCC} { s | s ∈ C } ;
5: for all s ∈ L do s.φ := true end for
6: while L ≠ ∅ do
7:    choose s ∈ L ; L := L - {s} ;
8:    for all (s',s) ∈→ such that s' ∈ Q' do
9:       if (s'.φ = false) then
10:         s'.φ := true ;
11:         L := L + {s'} ;
12:      end if
13:   end for
14: end while
15:
```

Algorithme 5: Fair Model Checking de CTL par étiquetage, cas $\mathbf{EG}\varphi$

4.4 Model checking de LTL par automates

Automates, mots et langages. Un automate $A = \langle \Sigma, Q, \rightarrow, q_0, F\rangle$ est un quintuplet constitué d'un ensemble d'états Q reliés par des transitions $\rightarrow \subseteq Q \times \Sigma \times Q$ étiquetées par les lettres d'un alphabet fini Σ. Parmis les états on distingue l'état initial q_0 et les états finaux F. Un mot w *accepté* par A est une suite fini de lettre $v_1 \ldots v_n$ telle qu'il soit possible d'aller de l'état initial q_0 à un état final $f \in F$ en prenant une transition étiquetée par v_1, puis une autre étiquetée par v_2, puis etc. jusqu'à atteindre f. L'ensemble des mots acceptés par un automate A est noté $\mathcal{L}(A)$. Les langages ainsi définissables s'appellent langages réguliers. Les automates offrent une manière élégante de manipuler les langages réguliers. Ainsi on peut facilement à partir de deux automates A_1 et A_2 calculer un automate qui reconnait $\mathcal{L}(A_1) \cap \mathcal{L}(A_2)$ ou tester si $\mathcal{L}(A) = \emptyset$, ou bien encore tester si $\mathcal{L}(A_1) = \mathcal{L}(A_2)$.

Par exemple, l'automate représenté à la figure 4.2 reconnaît le langage des mots terminant par `a`. Les états finaux sont représentés par des doubles cercles, l'état initial est fléché.

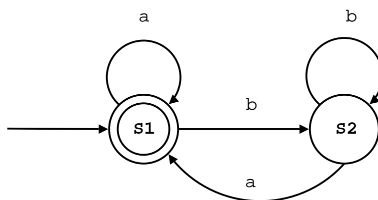


FIG. 4.2: Exemple d'automate.

Exercice 47. *Donnez des algorithmes pour : tester l'appartenance, tester le vide, construire l'union / l'intersection / le complément du langage d'un automate. Discutez la complexité.*

Liens intuitifs avec les systèmes réactifs. On peut voir un système de transitions comme un automate dont tous les états sont acceptants. Les structures de Kripke, en mettant les étiquettes sur les transitions et non plus sur les états, peuvent aussi se ramener à des automates. Dans ce cas, le langage de l'automate est *presque* l'ensemble des exécutions, soit exactement ce sur quoi on veut vérifier des propriétés temporelles. On peut imaginer alors par exemple qu'une question comme *"est-ce que $\mathcal{M}_1$ et $\mathcal{M}_2$ ont mêmes comportements ?"*

se ramène à tester $\mathcal{L}(A_1) = \mathcal{L}(A_2)$. Il y a quand même une différence : on a ici des exécutions finies, alors que nous sommes intéressés par des exécutions (mots) infinies. Pour pallier cette difficulté on va définir alors des automates de mots infinis.

Automates de Büchi. On étend les automates pour travailler sur des mots infinis. Un automate de Büchi est un automate $B = \langle \Sigma, Q, \rightarrow, q_0, F \rangle$, mais la condition d'acceptation est modifiée. L'ensemble F est maintenant appelé l'ensembles des états acceptants. Un mot infini σ est accepté si il part de q_0, respecte la relation de transition et *passe infiniment souvent par un état acceptant* $f \in F$.

Par exemple, si on considère maintenant l'automate de la figure 4.2 comme un automate de Büchi, le langage reconnu est celui des mots ayant une infinité de `a`.

Exercice 48. *Écrire des automates de Büchi sur l'alphabet* $\{a, b\}$ *reconnaissant les langages suivants :* a^w, $b^* a(a, b)^w$.

Exercice 49. *Comment tester l'appartenance d'un mot au langage d'un automate de Büchi ? Comment tester le vide d'un automate de Büchi ? Comment calculer l'union et l'intersection d'automates de Büchi ?*

Remarque 4.4.1. Deux différences notables entre automates de Büchi et automates finis : (1) les automates non déterministes sont strictement plus expressifs que les automates déterministes ; (2) la complémentation est une opération extrêmement coûteuse, en théorie[1] comme en pratique. Il est courant de ne pas pouvoir complémenter un automate d'une centaine d'états.

Model checking de LTL. Le model checking de LTL par automates suit les points suivants :

1. transformer une formule de chemin φ_p en automate de Büchi $B_{\neg\varphi_p}$;
2. transformer $\mathcal{M}$ en automate de Büchi $B_{\mathcal{M}}$;
3. calculer l'automate $B_{\otimes}$ reconnaissant $\mathcal{L}(B_{\mathcal{M}}) \cap \mathcal{L}(B_{\neg\varphi_p})$;
4. tester si le langage reconnu par $B_{\otimes}$ est vide ou non. On a $\mathcal{L}(B_{\otimes}) = \emptyset$ ssi $\mathcal{M} \models \mathbf{A}\varphi_p$.

Intuitivement, $\mathcal{L}(B_{\mathcal{M}})$ va représenter toutes les exécutions (infinies) de $\mathcal{M}$ et $\mathcal{L}(B_{\neg\varphi_p})$ va représenter toutes les exécutions ne satisfaisant pas φ_p. Ainsi $\mathcal{L}(B_{\mathcal{M}}) \cap \mathcal{L}(B_{\neg\varphi_p})$ est vide ssi tous les chemins de $\mathcal{M}$ satisfont φ_p, ssi $\mathcal{M} \models \mathbf{A}\varphi_p$.

Pour la construction de $B_{\neg\varphi_p}$, on procèdera récursivement. Bien entendu, $\wedge, \vee, \neg$ se traduiront par l'intersection, l'union et le complément sur les automates et on fait des constructions ad hoc pour les connecteurs temporels. Par exemple la figure 4.3 présente un automate de Büchi pour $\mathbf{F}^{\infty}p$.

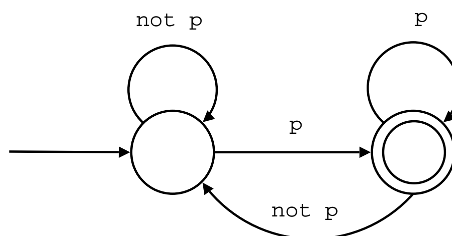


FIG. 4.3: Automate de Büchi pour $\mathbf{F}^{\infty}p$.

Exercice 50. *Transformez les propriétés de chemin suivantes en automates de Büchi sur alphabet* $\{p, q, \neg p, \neg q\}$ *:* $p, \neg p, \mathbf{X}p, \mathbf{F}p, \mathbf{G}p, p\mathbf{U}q, p\mathbf{W}q, \mathbf{F}^{\infty}p, \mathbf{G}^{\infty}p, p\mathbf{U}_{\leq 3}q$.

[1] Pendant longtemps le meilleur algorithme étaient double exponentielle, maintenant on est à $\mathcal{O}(2^{n^2})$, mais avec des constantes importantes et un algorithme très fin à implanter.

Exercice 51. *La complémentation des automates de Büchi est très coûteuse. Proposer une manière de s'en passer.*

Exercice 52 (Model checking de CTL* (**))**.** *Montrez comment adapter l'algorithme de model checking de LTL pour résoudre des formules linéaires du type* $\mathbf{E}\varphi$*, avec* φ *une formule de chemin (on parlera de* **E***-LTL et* **A***-LTL). Montrez ensuite comment modifier l'algorithme de model checking de* **E***-LTL et* **A***-LTL pour marquer tous les états vérifiant une formule, plutôt que savoir si l'état initial satisfait la formule. En déduire un algorithme de model checking pour CTL*. Évaluer sa complexité.*

Remarques sur la complexité. On peut montrer que le problème du model checking de LTL est PSPACE-complet. L'algorithme proposé ici est exponentiel : exponentiel pour la transformation de la formule, polynômial pour le produit et polynômial pour le test du vide. La complexité est en $\mathcal{O}(|\mathcal{M}| \times 2^{|\varphi|})$. Quelques remarques :

1. Même si la complexité peut paraître élevée, on remarque qu'elle est linéaire en la taille de $\mathcal{M}$ et que $|\varphi|$ est souvent petit. Ce résultat n'est donc pas rédhibitoire.
2. Ensuite cet algorithme n'est pas optimal du point de vue complexité, même s'il a l'avantage d'être simple à comprendre. On peut atteindre la complexité optimale en utilisant des automates plus élaborés, dits alternants avec bégaiement[2].
3. On peut faire mieux pour les propriétés de sûreté en utilisant des automates finis, voir ci-après.
4. En pratique le model checking par automate se prête bien a certaines optimisations, par exemple la construction à la volée de $\mathcal{M}$ (plutôt que tout construire d'abord) ou la réduction par ordres partiels. Ça ne change pas la complexité du problème général mais change parfois radicalement les choses en pratique.

Spécifier des propriétés temporelles par des automates. On a vu que nos formules LTL avaient une correspondance avec les automates de Büchi. En fait les automates de Büchi sont un formalisme à la fois plus expressif que LTL et plus facile d'accès pour les ingénieurs. Plusieurs model checkers pour logique linéaire proposent donc aussi de définir des propriétés directement par automate de Büchi, et appliquent le même algorithme que pour LTL. L'approche est séduisante mais a un gros défaut : on est obligé cette fois de passer par la complémentation de l'automate et la complexité augmente énormément. On peut aussi imaginer que l'utilisateur fournisse directement l'automate complémenté (laborieux et pas toujours plus efficace) ou se restreigne aux automates de Büchi déterministes faciles à complémenter mais moins expressifs (pas de $\mathbf{G}^{\infty}$ par exemple).

Sûreté et lien avec les automates observeurs. On peut voir le model checking de LTL comme une amélioration du principe des automates observeurs, où on synchronise avec un automate de Büchi et on teste l'accessibilité d'une composante fortement connexe. Cela permet de vérifier des propriétés de vivacité mais oblige à manipuler des objets plus complexes. En fait on peut montrer que pour les propriétés de sûreté on peut souvent se ramener automatiquement au cas d'un automate fini, ce qui peut permettre d'améliorer la vérification. Le problème reste alors de pouvoir identifier syntaxiquement ces propriétés de sûreté.

Automates et logiques branchantes. On peut utiliser des automates pour le model checking de CTL et CTL*. On utilise alors des automates d'arbres infinis, et là encore des automates alternants permettent des complexités optimales. On peut aussi définir des logiques branchantes à base des automates d'arbres. Cependant ces travaux sont plutôt d'intérêt théorique car CTL* est déjà suffisament expressive et de toute manière peu utilisée en pratique tandis que CTL a des algorithmes optimisés beaucoup plus efficaces.

[2]Le résultat est meilleur, mais reste exponentiel en pratique

Références

[1] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci and P. Schnoebelen. *Vérification de logiciels. Techniques et outils du model-checking.* Vuibert, 1999.

[2] E. M. Clarke, O. Grumberg and D. A. Peled. *Model Checking.* MIT press, 1999.

[3] E. M. Clarke and J. Wing. *Formal Methods : State of the Art and Future Directions.* ACM, 1996.

[4] H. Comon. *Automates et logiques temporelles.* Cours Ens Cachan. `www.lsv.ens-cachan.fr/~comon`

[5] S. Demri. *Temporal logics.* Cours Mpri, Paris 7. `www.lsv.ens-cachan.fr/~demri`

[6] R. P. Kurshan. *Formal Verification In a Commercial Setting.* In *ACM IEEE DA'97*. ACM, 1997.

[7] M. Y. Vardi. *Branching vs linear time : Final showdown.* In *TACAS'2001*. LNCS 2031. Springer, 2001.

[8] M. Y. Vardi. *Automata-theoretic model checking revisited.* In *VMCAI'2007*. LNCS 4349. Springer, 2007.

[9] M. Y. Vardi. *From Church and Prior to PSL.* In *25MC*. 2006.

Annexe A

Rappels de logique

A.1 Définitions

Une logique est la donnée de
- (aspect syntaxique) un langage (récursif) L de formules φ.
- (aspect sémantique) un domaine S d'interprétations $\mathcal{I}$
- (relie les deux) une relation de satisfaction $\models \subseteq S \times L$

Définition A.1.1. On dit que
- $\mathcal{I}$ est un *modèle* de φ si $\mathcal{I} \models \varphi$. Dans ce cas on dit que $\mathcal{I}$ *satisfait* φ.
- φ est *satisfaisable* si il existe une interprétation qui la satisfait.
- φ est *valide* si toute interprétation la satisfait. On note alors $\models \varphi$.
- φ est *contradictoire* si aucune interprétation ne la satisfait.

La sémantique d'une formule φ, notée $[\![\varphi]\!]$, peut être vue comme l'ensemble des interprétations qui la satisfont. C'est à dire que $[\![\varphi]\!] = \{\mathcal{I} \in S | \mathcal{I} \models \varphi\}$.

Définition A.1.2. On dit que :
- Un ensemble $A \subseteq S$ est L-définissable si il existe une formule φ de L telle que $A = [\![\varphi]\!]$.
- Pour une logique donnée L, l'ensemble des ensembles L-définissables donne une idée du pouvoir d'expression de L (cf problème d'expressivité si dessous).

A.2 Problèmes classiques liés aux logiques

Problèmes relatifs à une formule de la logique :

1. (MC) **Model checking**. Entrée : $\mathcal{I}, \varphi$. Réponse : est-ce que $\mathcal{I} \models \varphi$?
2. **Validité**. Entrée : φ. Réponse : est-ce que $\models \varphi$?
3. (SAT) **Satisfaction**. Entrée : φ. Réponse : est-ce que φ est satisfaisable ?
4. **Synthèse**. Entrée : φ. Réponse : donner, si il existe, $\mathcal{I}$ tel que $\mathcal{I} \models \varphi$.

Problèmes relatifs à la logique elle-même :

1. **Expressivité**. Étant donné deux logiques, est-ce qu'elles définissent les mêmes ensembles ?
2. **Concision**. Étant donné deux logiques, est-ce que les formules pour exprimer les mêmes ensembles ont même taille ?
3. **Pouvoir de séparation**. Étant donné une logique L et deux interprétations $\mathcal{I}_1$ et $\mathcal{I}2$, est-ce que L peut les distinguer, c'est à dire est-ce qu'il existe $\varphi \in L$ telle que $\mathcal{I}_1 \models \varphi$ et $\mathcal{I}_2 \not\models \varphi$?

A.3 Quelques logiques

De nombreuses logiques ont été définies, chacune adaptée à exprimer un point de vue particulier. Ce travail prend surtout en compte le pouvoir d'expression et la concision. De plus, dans le cadre du raisonnement automatique, de nombreuses sous-classes ont ensuite été explorées pour gagner en décidabilité et complexité.

- logiques classiques de prédicats (fondements des mathématiques)
- logique classique propositionnelle (électronique)
- logiques intuitionistes (fondements de la programmation)
- logiques linéaires[1] (prise en compte des ressources)
- logiques monadiques (diverses types de raisonnements en IA, par exemple *croyances & savoirs*)

A.4 Exemple : logique classique propositionnelle

On considère le cas le plus simple : la logique des propositions. On se donne un ensemble fini $A_1, \ldots, A_n$ de propositions atomiques. Le langage des formules propositionnelles est défini par la grammaire :

atome : := A_i — $\top$ — $\bot$
formule : := formule $\vee$ formule — formule $\wedge$ formule — $\neg$ formule — atome

Le domaine d'interprétation S est l'ensemble des valuations booléennes des A_i. Une interprétation $\mathcal{I}$ est donc une fonction qui assigne une valeur dans $\{0, 1\}$ à chaque A_i.

La satisfaction $\mathcal{I} \models \varphi$ est définie inductivement par :

$\mathcal{I} \models \top$ pour tout $\mathcal{I}$,
$\mathcal{I} \not\models \bot$ pour tout $\mathcal{I}$,
$\mathcal{I} \models A_i$ si $\mathcal{I}(A_i) = 1$,
$\mathcal{I} \models f_1 \wedge f_2$ si $\mathcal{I} \models f_1$ **ou** $\mathcal{I} \models f_2$,
$\mathcal{I} \models f_1 \vee f_2$ si $\mathcal{I} \models f_1$ **et** $\mathcal{I} \models f_2$,
$\mathcal{I} \models \neg f$ si $\mathcal{I} \not\models f$.

Exercice 53 (Logique des propositions)**.**

1. *Dites pour chaque interprétation $\mathcal{I}$ si elle est un modèle de la formule $A \vee (\neg B)$:*
   $\mathcal{I}_1 : (A, B) \longrightarrow (0, 0)$, $\mathcal{I}_2 : (A, B) \longrightarrow (1, 1)$, $\mathcal{I}_3 : (A, B) \longrightarrow (1, 0)$.
2. *Que dire des formules suivantes (satisfaisable, valide, contradictoire) : $A \wedge \neg A$, $A \vee \neg A$, $A \vee B$*
3. *Ajouter à la logique les connecteurs $\rightarrow$, $\leftrightarrow$ et* `xor`*.*
4. *Exprimer ces connecteurs en fonction des anciens.*
5. *Exprimer $\vee, \top, \bot$ en fonction de $\wedge, \neg$.*
6. *Montrer que tous les connecteurs peuvent s'obtenir à partir de $\neg p \wedge \neg q$.*

Exercice 54. *Quel lien y a-t-il entre satisfaisabilité de f et validité de $\neg f$?*

Exercice 55. *On définit la relation $\equiv$ sur les formules logiques par $\varphi_1 \equiv \varphi_2$ ssi φ_1 et φ_2 ont les mêmes modèles. Donnez une définition formelle de "ont les mêmes modèles". Quel est le lien entre $\varphi_1 \equiv \varphi_2$ et $\varphi_1 \leftrightarrow \varphi_2$?*

Exercice 56 (**)**.** *Soit F un ensemble fini de formules de logique classique propositionnelle sur des propositions atomiques $p_1, \ldots, p_n$. À partir de quelle valeur de $|F|$ est-on sûr d'avoir au moins deux $\varphi_1, \varphi_2 \in F$ telles que $\varphi_1 \equiv \varphi_2$?*

Exercice 57 (Forme normale (*))**.** *Montrer que toute formule φ peut se mettre sous une forme $\bigvee_i \bigwedge_j \bar{p_i}$, où $\bar{p_i}$ vaut soit p_i soit $\neg p_i$.*

[1] Rien à voir avec LTL

Exercice 58 (QBF (*)). *On appelle QBF (* Quantified boolean Formulas*) la logique des propositions à laquelle on ajoute les quantificateurs* $\exists$ *et* $\forall$. *On pourra ainsi écrire des formules comme :* $\exists x, x \wedge y$.

1. *Définissez* $\models$ *pour ces nouveaux opérateurs.*
2. *Montrer que toute formule* φ *de QBF peut se traduire en une formule* $\widetilde{\varphi}$ *de logique des propositions.*
3. *Quel lien y a-t-il entre* $|\varphi|$ *et* $|\widetilde{\varphi}|$ *?*

Annexe B

Notions de calculabilité et complexité

B.1 Calculabilité

Calculabilité. Ce domaine de recherche s'intéresse à distinguer les problèmes solubles par ordinateur (avec des ressources arbitrairement grandes, mais finies[1]) de ceux qui ne le sont pas. On modélise un problème par un langage (des couples entrées-solution). On ramène alors le but initial à la définition de langages reconnus par un certain modèle de machine. Ce modèle se doit d'être le plus près possible de la notion de "fonction humainement calculable", sinon dès qu'un formalisme plus puissant serait trouvé, il faudrait tout refaire. Le formalisme choisit est celui des **machines de Turing**. Ce sont globalement des automates étendus par un ruban infini et des opérations de lecture / écriture. On se limite aux problèmes de décisions. On définit entre autre les trois classes suivantes de problèmes / langages :

- problème décidable : soluble par un programme.
- problème indécidable : il ne peut exister de programme qui le résoud.
- problème semi-décidable : il existe un programme qui répond toujours 1 quand il faut, mais qui peut soit répondre 0 soit ne pas terminer quand la réponse est "faux".

Exemples de problèmes décidables. La plupart des questions raisonnables sur les automates finis sont décidables, par exemple savoir si un mot est accepté ou si le langage reconnu est vide. Les questions relatives à des logiques simples, typiquement la logique des propositions, sont souvent aussi décidables.

Exemples de problèmes indécidables. On arrive rapidement à des problèmes indécidables quand on commence à utiliser les machines de Turing (comme outil de décision) pour résoudre des problèmes sur les machines de Turing (comme objet d'étude). En fait *"Tout problème non trivial sur les machines de Turing est indécidable"* (théorême de Rice). Par exemple on ne peut pas décider si une machine de Turing termine, peut atteindre un certain point de son programme ou fait ce qu'elle devrait faire. Ce dernier point est important car il implique que la vérification automatique de vrais programmes, écrits dans de vrais langages et tournant sur de vrais ordinateurs, est impossible.

Pour regagner en décision, il faut en fait s'intéresser à des formalismes moins puissants.

Thèse de Church. *La thèse de Church déclare qu'aucun système de calcul automatisé ne peut avoir une puissance supérieure à celle des machines de Turing, du point de vue de la calculabilité.*

Cela signifie que nos ordinateurs actuels, tout comme les ordinateurs qui seront inventés dans 1000 ans ou ceux qui auraient été utilisés *a long time ago in a galaxy far, far away* ont fondamentalement les mêmes possibilités et limites théoriques que les machines de Turing. Ainsi si on accepte cette thèse, le caractère indécidable d'un problème n'est plus spécifique aux machines de Turing mais est intrinsèque au problème lui-même.

[1] C'est cohérent puisqu'on ne peut guère prévoir l'évolution future des moyens de calcul.

Deux remarques importantes : quand on dit *"la même puissance"* c'est en terme de décision. La conjecture ne dit rien sur la vitesse de calcul, et donc sur la possibilité pratique de mener la résolution du problème à bien. Ensuite c'est une conjecture et pas un théorème. Même si on a de fortes raisons d'y croire, rien ne dit qu'elle ne sera jamais remise en cause.

Les preuves de la conjecture reposent principalement sur le fait que d'une part toutes les extensions tentées sur les machines de Turing (probabiliste, quantique, non déterministe) n'ont au final pas ajouté de pouvoir d'expression, et d'autre part que toutes les autres modélisations de la notion de fonction calculable ont abouti à des formalismes de même pouvoir que celui des machines de Turing, bien que partant parfois de concepts très différents. On peut citer par exemple le λ-calcul de Church, les fonctions récursives de Kleene et les grammaires génératives de Chomsky.

Remarquons enfin que la conjecture dite *thèse forte de Church-Turing* (ils n'en sont pas auteurs) qui stipule que tout système de calcul automatisé est aussi rapide à un facteur polynômial près qu'une machine de Turing pourrait bien s'effondrer à cause des ordinateurs quantiques (quantum computing) et biologiques (DNA computing). Cependant la conjecture n'est pas encore formellement réfutée, même si on la pense maintenant assez improbable.

B.2 Complexité

La complexité s'interesse à distinguer parmi les problèmes décidables ceux qui le sont en temps et en mémoire raisonnables, de ceux qui ne le sont pas[2]. On classe ainsi les problèmes décidables selon leur classe de complexité, c'est à dire selon le minimum de ressources qu'un algorithme qui résoud ces problèmes doit dépenser. On définit par exemple les classes suivantes :

- P : problèmes solubles en temps polynômial.
- NP : problèmes solubles en temps polynômial *sur une machine non déterministe*. En pratique ils peuvent nécessiter un temps exponentiel.
- P^{NP} : problèmes solubles *avec un nombre d'appels polynômial à une machine indéterminste*. En pratique ils peuvent nécessiter un temps exponentiel.
- $PSPACE$: problèmes solubles en espace polynômial. En pratique ils peuvent nécessiter un temps exponentiel.
- $NPSPACE$: comme $PSPACE$ pour machines non déterministes.
- $EXPTIME$, $NEXPTIME$, $EXPSPACE$, $NEXPSPACE$, $2-EXPTIME$,...

Les classes sont présentées en ordre d'inclusion croissant. Il est acquis que $PSPACE = NPSPACE$, $EXPSPACE = NEXPSPACE$, etc. Peu d'inclusions strictes sont connues. On est certain des inclusions strictes entre P, $EXPTIME$, $2-EXPTIME$, etc. Idem pour $PSPACE$, $EXPSPACE$, $2-EXPSPACE$, etc. On sait aussi que $PSPACE$ est différent de $EXPTIME$.

Par contre on ne sait par exemple si $P = PSPACE$ ou si $P = NP$.

Pour donner une idée de la hiérarchie, typiquement, si on a un système d'inéquations linéaires : vérifier une solution est dans P, trouver une solution est dans NP et trouver une solution minimale est dans P^{NP}.

[2]Des milliards d'année pour répondre 42 n'est pas considéré comme raisonnable.

Annexe C

Divers problèmes algorithmiques

C.1 Composantes fortement connexes

Formellement un graphe orienté G est une paire $G = \langle Q, T \rangle$ où Q est un ensembles d'états (ou nœuds) et $T \subseteq Q \times Q$ un ensemble de transitions. Comme d'habitude, $(q, q') \in T$ signifie qu'on peut aller de q à q' en prenant une transition du graphe. On dit que q' est atteignable à partir de q, noté $q \xrightarrow{*} q'$ si il existe un chemin (une suite de transitions) de q à q'.

Définition C.1.1. Soit $G = \langle Q, T \rangle$ un graphe orienté. On appelle composante connexe de G tout sous ensemble non vide $C \subseteq Q$ tel que pour tout $c_i, c_j \in C$, c_j est atteignable à partir de c_i *en restant dans* C.

Une composante fortement connexe est une composante connexe maximale : si on lui rajoute un nouveau nœud, elle n'est plus connexe. Un composante fortement connexe non triviale a soit au moins deux nœuds, soit un seul nœud avec une transition sur lui-même, c-à-d $(q, q) \in T$. On s'intéresse à la décomposition en composantes fortement connexes non triviales d'un graphe orienté.Un exemple de décomposition est donné à la figure C.1.

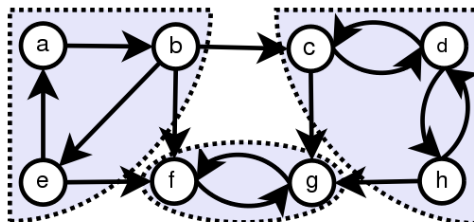


Fig. C.1: Exemple de décomposition en composantes fortement connexes

Algorithme de Kosaraju (1978). Intuitivement l'algorithme est en trois passes. (1) On fait une recherche en profondeur d'abord[1] dans le graphe G, et on marque chaque nœud dans l'ordre où il est traité (les feuilles ont les plus bas indices, la racine a le plus haut). (2) On inverse G en G_r, en inversant le sens des transitions. (3) On part du nœud d'indice le plus haut, et on fait une DFS dans G_r. Tous les nœuds rencontrés forment la première composante fortement connexe. Si tous les nœuds n'ont pas été rencontrés, on recommence avec l'indice le plus haut restant.

Exercice 59. *Prouvez la correction de l'agorithme de Kosaraju. Pour cela, vous procéderez en deux phases. D'abord (1) montrez que si x, y sont dans la même SCC alors ils sont dans le même arbre calculé par la DFS sur G_r. Puis (2) montrez le sens inverse.*

[1] DFS pour *depth first search*.

Algorithme KOSARAJU

input : graph $G = \langle Q, T \rangle$, initial state s0

```
 1: depth := 0 ;
 2: L := Q ;
 3: SCC := ∅ ; // set of SCCs
 4: marking_dfs(s0) ;
 5: L := Q ;
 6: while L ≠ ∅ do
 7:    choose s ∈ L with highest s.depth ;
 8:    CurrentSCC := ∅ ;
 9:    drawing_scc(s) ;
10:    SCC := SCC + {CurrentSCC}
11: end while
12: RETURN(SCC) ;
13:
14: procedure marking_dfs(s) // DFS transversal and marking
15:    L := L - {s} ;
16:    forall (s, s') in T do
17:       if (s' ∈ L) do marking_dfs(s') ; end if
18:    end for
19:    s.depth := depth ; depth := depth + 1
20: end proc
21:
22: procedure drawing_scc(s) // DFS on Gr to recover current SCC
23:    CurrentSCC := CurrentSCC + {s} ;
24:    L := L - {s} ;
25:    forall (s', s) in T do
26:       if (s' ∈ L) do drawing_scc(s') ; end if
27:    end for
28: end proc
```

Algorithme 6: Décomposition en SCC, algorithme de Kosaraju

Annexe D

Sujets de partiel

D.1 ENSTA, année 2006-2007

Exercice 60.

1. *Qu'est-ce que le model checking ?(2 lignes max)*
2. *Citer deux domaines d'application du model checking.*
3. *Quelles sont les caratéristiques d'un système réactif ? (5 lignes max)*
4. *Quels sont les liens entre machine à états, systèmes de transitions et structure de Kripke ? (5 lignes max)*
5. *Qu'est-ce qui distingue les logiques temporelles de la logique classique ? (2 lignes max)*
6. *Qu'est-ce qu'un connecteur temporel ? Quels sont les connecteurs temporels vus en cours ?*
7. *Qu'est-ce qu'un quantificateur de chemin ? Quels sont les quantificateurs de chemin vus en cours ?*
8. *Quelle est la différence entre une logique branchante et une logique linéaire ?*
9. *Quelles sont les différences (au niveau définitions) entre CTL*, LTL et CTL ?*
10. *À quelles grandes classes de propriétés temporelles (accessibilité, invariance, etc.) appartiennent ces formules :* $\mathbf{AG}p$, $\mathbf{EF}p$, $\mathbf{A}((\neg p\mathbf{U}q) \vee \mathbf{G}\neg p)$, $\mathbf{A}(\mathbf{GF}p \to \mathbf{GF}q)$, $\mathbf{AG}(p \to \mathbf{F}q)$.
11. *Exprimer* $\vee, \mathbf{F}, \mathbf{G}$ *en fonction de* $\neg, \wedge, \mathbf{U}$.
12. *Quel est l'intérêt de restreindre ainsi le nombre de connecteurs ? Quel est le désavantage ?*
13. *Quelles logiques parmi LTL, CTL et CTL* peuvent exprimer l'équité ?*
14. *Faites un schéma indiquant les relations d'inclusion entre LTL, CTL, CTL* et ACTL* (vue en TD). Quand deux logiques sont incomparables, donnez une propriété définissable dans la première et pas dans la seconde, et vice versa. Quand une logique est incluse dans une autre, dites pourquoi.*
15. *Comment vérifieriez-vous les types de propriétés suivants si vous aviez un model checker pour (1) CTL, (2) LTL, (3) CTL* ? (a) accessibilité, (b) invariance, (c) équité.*

Exercice 61.

1. *Quels sont les algorithmes de model checking présentés dans le cours ? Donnez l'idée de chacun des algorithmes. (15 lignes max en tout)*
2. *Pour l'algorithme de marquage de CTL, écrivez les cas* $\mathbf{AX}\varphi$ *et* $\mathbf{AG}\varphi$, sans passer par une traduction dans d'autres opérateurs.
3. *Dessinez les automates de Büchi représentant les propriétés* $\mathbf{GF}p$ *et* $\mathbf{FG}p$.

Exercice 62. *Montrer que le problème du model checking de LTL se ramène au problème de la validité de LTL. Plus précisément, on se donne une formule LTL* φ *et une structure de Kripke* $\mathcal{M} = \langle Q, \rightarrow, P, l, s_0 \rangle$*. On va construire une formule* φ'' *telle que* $\mathcal{M}, s_0 \models \varphi$ *ssi* φ'' *est valide. Pour cela on va construire une formule* φ' *telle que* $\sigma \models \varphi'$ *ssi* $\sigma \in \mathcal{L}(\mathcal{M})$*. On commence par rajouter une variable propositionnelle* p_{s_i} *pour chaque état* s_i *de* $\mathcal{M}$*.*

1. *Construisez les formules suivantes :* $PROP_{s_i}$ *qui indique les propriétés atomiques vérifiées par* s_i *et* $NEXT_{s_i}$ *qui mime la relation de transition de* $\mathcal{M}$*.*
2. *Servez-vous des résultats précédents pour construire* φ'*.*
3. *Concluez en construisant la formule* φ'' *cherchée à partir de* φ' *et* φ*.*

D.2 ENSTA, rattrapages, année 2006-2007

Exercice 63 (Questions de Cours)**.**

1. *Qu'est-ce que le model checking ?*
2. *Qu'est-ce qu'une logique temporelle ?*
3. *Détailler les entrées-sorties d'un algorithme de model checking et expliquer le rôle de chacune des entrées.*
4. *Que signifient* logique linéaire *et* logique branchante *?*
5. *Donner une définition de la logique CTL, par exemple en citant les opérateurs permis ou en donnant la grammaire des formules CTL.*

Exercice 64. *Pour l'algorithme de model checking pour CTL :*

1. *Donner les entrées-sorties.*
2. *Expliquer le principe de l'algorithme.*
   Vous pourrez par exemple illustrer le fonctionnement sur un petit exemple ou vous appuyer sur du pseudo-code de haut niveau.
3. *Donner un programme pour gérer les cas* $\mathbf{EX}\varphi$ *et* $\mathbf{AX}\varphi$*.*
   Vous pourrez utiliser du pseudo-code, par exemple des opérations ensemblistes plutôt que des manipulations de structures de données.

Exercice 65. *Expliquer la notion de sémantique* fair *utilisée dans l'algorithme de model checking de fair CTL. Quelles formules peut-on alors exprimer dans fair CTL ?*

Exercice 66. *Soit deux structures de Kripke* $\mathcal{M}_1, \mathcal{M}_2$ *ayant même langage* $\mathcal{L}(\mathcal{M}_1) = \mathcal{L}(\mathcal{M}_2)$*, et* φ *une formule LTL. Que peut-on dire si* $\mathcal{M}_1 \models \varphi$ *? Que se passe-t-il maintenant si* $\mathcal{L}(\mathcal{M}_1) \subseteq \mathcal{L}(\mathcal{M}_2)$ *? Ces résultats sont-ils valides pour* φ *dans CTL ?*

Exercice 67 (Questions de Cours)**.**

1. *Qu'est-ce qu'une logique temporelle ?*
2. *Pourquoi utilise-t-on des logiques temporelles ? Quels sont les avantages par rapport au langage naturel et à la logique classique ?*
3. *Définissez LTL,* CTL^**, CTL.*

Exercice 68. *Model checking de LTL*

1. *Définition formelle des automates de Büchi et du langage associé*
2. *Lien LTL - automates de Büchi*
3. *Illustrer ce lien pour les formules* $\mathbf{FG}p$ *et* $\mathbf{GF}p$*.*
4. *Schéma de l'algorithme de model checking de LTL*

Exercice 69. *À propos des automates de Büchi :*

1. *Comment tester si un mot infini appartient au langage d'un automate de Büchi ?*
2. *Comment tester si le langage d'un automate de Büchi est vide ?*

Exercice 70 (Questions de Cours)**.**

1. *Donner une définition de chacune des classes de propriétés temporelles suivantes : accessibilité, invariance, vivacité, équité, équivalence comportementale.*
2. *Quels sont les liens entre accessibilité et invariance ?*
3. *Qu'est-ce que l'équivalence comportementale ? Quel est l'intérêt ?*

Exercice 71.

1. *Donner la définition formelle de la relation $\models$ pour un chemin infini σ.*
2. *Montrer que* $\mathbf{U}$ *suffit à exprimer* $\mathbf{F}$ *et* $\mathbf{G}$*.*
3. *Transformer les formules suivantes (si possible) en formules CTL*
   - $\mathbf{E}(p \vee \mathbf{F}q)$
   - $\mathbf{AGF}p$
4. *Montrer que* $\mathbf{AFG}p \neq \mathbf{AFAG}p$*.*