Vérification Qualitative

Franck Cassez

IRCCyN/CNRS UMR 6597
BP 92101
1 rue de la Noë
44321 Nantes Cedex 3 France,
email : Franck.Cassez@irccyn.ec-nantes.fr

RÉSUMÉ. *Ce document est une introduction rapide au domaine de la modélisation et de la vérification des applications concurrentes. On y présente un modèle permettant de décrire de telles applications : les systèmes de transitions. Les logiques temporelles LTL et CTL sont ensuite introduites pour spécifier les propriétés de ces systèmes. Enfin, les principes des algorithmes de model-checking pour LTL et CTL sont donnés.*

ABSTRACT. *In this paper we give a quick overview of the area of modelling and verification of concurrent systems. We take transition systems as the basic model of concurrent systems, and introduce the temporal logics LTL and CTL. Moreover we give some hints for the model-checking algorithms for LTL and CTL.*

MOTS-CLÉS : *système de transitions, logique temporelle, model-checking.*

KEYWORDS: *transition systems, temporal logics, model-checking*

1. Introduction

Dans ce document on introduit le domaine de la *modélisation et de la vérification* des *applications concurrentes*. Le contenu est fortement inspiré de [SCH 99] et le lecteur intéressé pourra consulter cet excellent livre où il trouvera toutes les informations lui permettant d'aller plus loin.

Le but de la *vérification* est d'assurer qu'un programme (ou système) satisfait un certain nombre de *propriétés*. Pour cela il est nécessaire de *modéliser formellement* le comportement du programme : nous introduisons dans la Section 2 les systèmes de transitions (finis) qui permettent de décrire des applications concurrentes. Dans la Section 3 on introduit les *logiques temporelles* LTL et CTL qui sont des langages de *spécification des propriétés*. Une fois le système décrit par un système de transitions, et les propriétés souhaitées spécifiées en logique temporelle, il existe un algorithme dit de *model-checking* permettant de répondre automatiquement à la question : «est-ce que le système satisfait les propriétés souhaitées ?». La Section 4 décrit les principes des algorithmes de model-checking pour LTL et CTL.

2. Modélisation des systèmes concurrents

La première étape dans la vérification formelle d'un système est la construction d'un *modèle formel* du comportement de l'application. Des langages de spécification de haut niveau pour les applications concurrentes existent depuis les années 80 [BOU 91, CAS 87, Le 91, ELL 85]. Certains de ces langages sont maintenant utilisés dans le cadre de développements industriels. Cependant la *sémantique* de ces langages est très souvent donnée en terme de *systèmes de transitions* [ARN 92] qui sont des machines à états dont le fonctionnement est très simple. On va donc décrire ces systèmes de base. En composant ces systèmes (par synchronisation) il est possible de décrire des comportements complexes d'activités concurrentes.

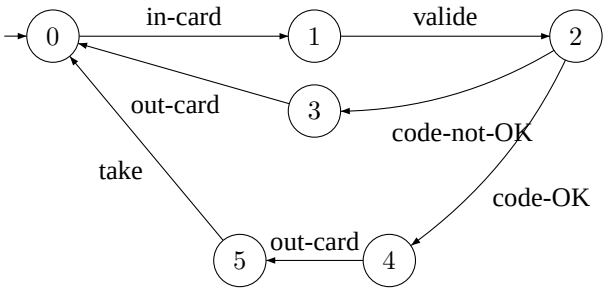


Figure 1. *Système de transitions représentant le fonctionnement d'un GAB*

2.1. *Systèmes de transitions*

Définition 1 (Système de transitions [ARN 92]) *Un* système de transitions S *est un tuple* $(Q, s_0, A, \longrightarrow)$ *tel que :*

– Q *est un ensemble (fini)* d'états,
– $s_0 \in Q$ *est l'*état initial,
– A *est un* alphabet (fini) d'actions,
– $\longrightarrow \subseteq Q \times A \times Q$ *est la* relation de transition *de* S. *On note* $s \xrightarrow{a} s'$ *si* $(s, a, s') \in \longrightarrow$.

Une exécution σ *de* S *est une séquence de transitions* $\sigma = s_0 \xrightarrow{a_0} s_1 \dots s_n \xrightarrow{a_n} s_{n+1} \dots$ *telle que* $\forall i \geq 0, (s_i, a_i, s_{i+1}) \in \longrightarrow$. $a_0 a_1 \cdots a_n \cdots$ *est la* trace *de* σ *notée* $Tr(\sigma)$. *Une exécution est finie si la séquence qui la compose est finie. On note* $s_0 \xrightarrow{w} s$, $w \in A^*$ *s'il existe une exécution finie dans* S *menant de* s_0 *à* s *et de trace* w. $Run^*(S)$ *est l'ensemble des exécutions finies de* S *et* $Run^\omega(S)$ *l'ensemble des exécutions infinies. Un état* $s \in Q$ *est* accessible *dans* S *ssi* $\exists w \in A^*, s_0 \xrightarrow{w} s$. $Reach(S)$ *est l'ensemble des* états accessibles *dans* S *à partir de* s_0 *:* $Reach(S) = \{s \in Q, \exists w \in A^*, s_0 \xrightarrow{w} s\}$. *Un système de transitions* étiqueté *est un système de transitions muni d'une fonction* $L : Q \to 2^{AP}$ *où* AP *est un ensemble de* propositions atomiques *;* $L(s)$ *donne l'ensemble des propositions que satisfait l'état* s. *On note* $(Q, s_0, A, \longrightarrow, L)$ *un système de transitions étiqueté.* □

Un exemple de système de transitions modélisant un guichet automatique de banque est donné Figure 1. Dans ce système l'état initial est 0.

On peut bien entendu ajouter des variables entières ou plus généralement des variables à valeurs dans un ensemble fini [MAN 91] pour étendre le modèle de la Définition 1. Dans ce cas le nombre d'états du système est toujours fini. Un exemple de l'utilisation de variables entières est donné dans la Figure 2.

2.2. *Synchronisation de systèmes de transitions*

Un système concurrent est en général constitué de plusieurs composants interagissants. Pour décrire les interactions on utilise la notion de *produit synchronisé* à la Arnold-Nivat [ARN 92]. Pour modéliser le fonctionnement *asynchrone* des systèmes on introduit une action spéciale $\bullet$ qui indique qu'un système ne «fait rien pendant que d'autres changent d'état».

Définition 2 (Produit synchronisé [ARN 92]) *Soient* n *systèmes de transitions* $S_i = (Q_i, s_0^i, A_i, \rightarrow_i), i \in [1, n]$, *et* $I \subseteq A_1 \cup \{\bullet\} \times A_2 \cup \{\bullet\} \times \cdots \times A_n \cup \{\bullet\}$ *une* contrainte de synchronisation. *Un élément* $(a_1, a_2, \dots, a_n)$ *de* I *est appelé un* vecteur de synchronisation. *Le* produit synchronisé $(S_1 \parallel S_2 \cdots \parallel S_n)_I$ *des* n *systèmes de transitions* S_i *avec la contrainte* I *est le système de transitions* $S = (Q, s_0, A, \rightarrow)$ *défini par :*

– $Q = Q_1 \times Q_2 \times \cdots \times Q_n$,
– $s_0 = (s_0^1, s_0^2, \dots, s_0^n)$,
– $A = A_1 \cup \{\bullet\} \times A_2 \cup \{\bullet\} \times \cdots \times A_n \cup \{\bullet\}$,

– $\longrightarrow \subseteq Q \times A \times Q$ *est telle que :*

$$(q_1, \ldots, q_n) \xrightarrow{(a_1,\ldots,a_n)} (q'_1, \ldots, q'_n) \Longleftrightarrow \forall i \in [1, n], \begin{cases} \textit{si } a_i \in A_i, q_i \xrightarrow{a_i} q'_i \\ \textit{sinon } a_i = \bullet \textit{ et } q_i = q'_i \end{cases}$$

□

Ce type de synchronisation permet de décrire des envois/réceptions de messages (par exemple sur la Figure 2 et Figure 3 les ! et ? représentent les émissions et réceptions de messages). Un exemple de contrainte de synchronisation pour les automates des Figure 2 et Figure 3 est donné dans la Table 1. Sur cet exemple on choisit de renommer les vecteurs de synchronisation : on a ainsi une synchronisation à la Arnold-Nivat avec *renommage*. Formellement il suffit d'étendre la Définition 2 ainsi : au lieu de I, on utilise une fonction partielle f de synchronisation $f : A_1 \cup \{\bullet\} \times A_2 \cup \{\bullet\} \times \cdots \times A_n \cup \{\bullet\} \longrightarrow A$ (où A est un nouvel alphabet) et

$$(q_1, \ldots, q_n) \xrightarrow{b} (q'_1, \ldots, q'_n) \Longleftrightarrow \begin{cases} f(a_1, \ldots, a_n) = b \\ \wedge \forall i \in [1, n], \begin{cases} \textit{si } a_i \in A_i, q_i \xrightarrow{a_i} q'_i \\ \textit{sinon } a_i = \bullet \textit{ et } q_i = q'_i \end{cases} \end{cases}$$

Avec une fonction de renommage comme celle de la Table 1 le produit synchronisé des automates des Figure 2 et Figure 3 est celui de la Figure 4.

$f($ *?in-card, !in-card*$) =$ *in*
$f($ *?valide, !valide*$) =$ *valide*
$f($ *!out-card, ?out-card*$) =$ *out*
$f($ *?take, !take*$) =$ *take*
$f($*code-OK*, $\bullet) =$ *code-OK*
$f($ *code-not-OK*, $\bullet) =$ *code-not-OK*
$f($ *keep-card*, $\bullet) =$ *keep-card*

Tableau 1. *Contrainte de synchronisation pour le GAB*

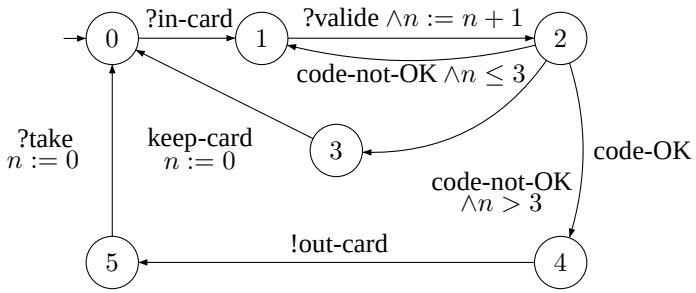


Figure 2. *Un GAB avec messages*

2.3. *Propriétés des exécutions d'un système de transitions*

Blocage.

Un état *deadlock* (bloquant) d'un système de transitions S est un état qui n'est source d'aucune transition. Une propriété importante que l'on souhaite souvent assurer dans un système de transitions est l'absence de deadlock

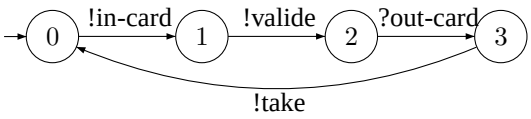


Figure 3. *Un utilisateur du GAB*

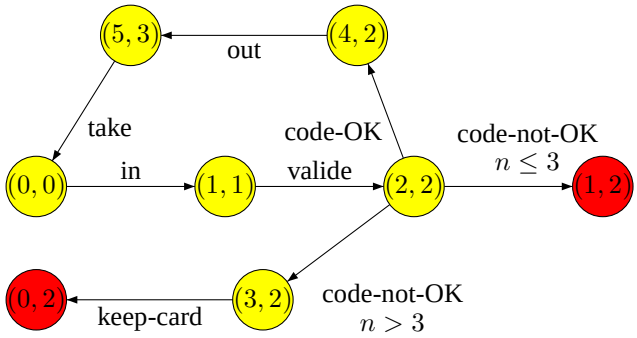


Figure 4. *Produit synchronisé GAB* $\times$ *U – contrainte I*

(car ces états correspondent à des blocages non voulus). Mais certains états du système correspondant à la fin de l'exécution de l'application sont alors considérés comme des deadlocks. Il est donc nécessaire de distinguer les blocages non souhaités de ceux qui correspondent à la fin de l'exécution du système. Ceci est possible en utilisant la fonction d'étiquetage (Cf. Définition 1) et en étiquetant les états de fin d'exécution par une propriété *terminal*. Les vrais états deadlock sont alors les états bloquants qui n'ont pas la propriété *terminal*.

Équité.

Une autre propriété importante d'une exécution d'un système de transitions est l'*équité* [ARN 92, SCH 99, EME 90]. On considère dans ce cas que toutes les exécutions sont infinies (au besoin on les complète par des transitions $\bullet$). On dit qu'une transition d'étiquette a est *tirable* à partir d'un état s si $\exists s' | (s, a, s') \in \longrightarrow$. La notion d'équité vise à définir un sous ensemble de l'ensemble des exécutions d'un système de transitions de façon à éliminer les exécutions non réalistes. Par exemple si l'on considère deux systèmes de transitions S_1 et S_2 que l'on synchronise, dans le produit synchronisé il peut y avoir des exécutions qui à partir d'un certain point ne contiennent que des transitions d'étiquette $(a, \bullet)$: le système S_2 ne fait rien alors que potentiellement une transition (x, b) est tirable. Si l'on considère que cette exécution est irréaliste il faut pouvoir la caractériser.

Un premier critère d'équité est celui de l'*équité forte* : une exécution σ satisfait le critère d'équité forte ssi toute transition *infiniment souvent tirable* est *infiniment souvent tirée*. Un critère plus faible est celui de l'*équité faible* : une exécution σ satisfait le critère d'équité faible ssi toute transition *toujours tirable à partir d'un certain point* est *infiniment souvent tirée*. L'équité forte implique l'équité faible.

2.4. *Propriétés des systèmes de transitions*

Déterminisme.

Un système de transitions $S = (Q, s_0, A, \rightarrow)$ est *déterministe* ssi

$$\forall s, s', s'' \in S, \forall a \in A, s \xrightarrow{a} s' \wedge s \xrightarrow{a} s'' \Longrightarrow s' = s''$$

Sinon il est *non déterministe*. Ceci implique que pour une trace donnée il n'y a qu'une seule exécution possible. Il n'est pas nécessaire qu'un système de transitions soit déterministe pour être un modèle correct. Le non déterminisme permet de représenter plusieurs choix possibles ou des parties de l'application pour lesquelles plusieurs comportements sont possibles sans que l'on sache ou que l'on veuille les modéliser de façon précise.

Équivalence de systèmes de transitions.

A partir d'un système de transitions $S = (Q, s_0, A, \longrightarrow)$ on définit l'ensemble des traces finies de S (le langage de S) par $L^*(S) = \{Tr(\sigma), \sigma \in Run^*(S)\}$. L'ensemble des traces infinies $L^\omega(S)$ es l'ensemble des traces des exécutions infinies : $L^\omega(S) = \{Tr(\sigma), \sigma \in Run^\omega(S)\}$.

Un autre point de vue peut être adopté : un système de transitions définit implicitement un *arbre d'exécution* qui correspond au dépliage de ce système à partir de l'état initial. L'arbre d'exécution $A(S)$ de S est défini inductivement par :

1) s_0 est la *racine* de $A(S)$;

2) si s est un nœud de $A(S)$ et $s \xrightarrow{a} s'$ alors s' est un fils de s.

Un tel arbre[1] est en général infini (sauf si toutes les exécutions sont finies). Tout chemin dans l'arbre à partir de la racine correspond à une exécution de S.

A partir de ces notions on peut parler d'*équivalences* de systèmes de transitions.

Deux systèmes de transitions $S_1 = (Q_1, s_0^1, A, \longrightarrow_1)$ et $S_2 = (Q_2, s_0^2, A, \longrightarrow_2)$ sont *trace-équivalents* ssi $L^*(S_1) = L^*(S_2)$ (une définition similaire est obtenue avec L^ω). Ils sont (fortement) *bisimilaires* ssi $A(S_1)$ et $A(S_2)$ sont isomorphes.

La notion de bisimulation est très importante et indique que si S_1 et S_2 sont bisimilaires, S_1 peut mimer chaque action de S_2 (et réciproquement) et donc ces deux systèmes sont indiscernables par un observateur qui ne voit que les actions.

Définition 3 (Relation de bisimulation) *Soit $S = (Q, s_0, A, \rightarrow)$ et $S' = (Q', s'_0, A, \rightarrow)$ deux systèmes de transitions*[2] *et $\mathcal{R} \subseteq S \times S'$. $\mathcal{R}$ est une* relation de bisimulation *ssi $\forall s \in S, s' \in S'$, $s\mathcal{R}s'$ implique*

$$\forall a \in A, s \xrightarrow{a} s_1 \Longrightarrow s' \xrightarrow{a} s'_1 \wedge s_1 \mathcal{R} s'_1 \quad (1)$$

$$\forall a \in A, s' \xrightarrow{a} s'_1 \Longrightarrow s \xrightarrow{a} s_1 \wedge s_1 \mathcal{R} s'_1 \quad (2)$$

S et S' sont en bisimulation *ssi il existe une relation de bisimulation $\mathcal{R}$ entre S et S' avec $s_0 \mathcal{R} s'_0$. Si on a uniquement la contrainte (1) de la Définition 3 alors $\mathcal{R}$ est une relation de* simulation *entre S' et S et S'* simule *S.* □

Si S simule S' et S' simule S, S et S' ne sont pas forcément bisimilaires : il est possible que ce soient deux relations différentes qui permettent les simulations.

La bisimilarité implique l'équivalence de traces. Si les systèmes S_1 et S_2 sont déterministes, l'équivalence de trace implique la bisimilarité. Un exemple de deux systèmes de transitions trace-équivalents mais non bisimilaires est donné Figure 5.

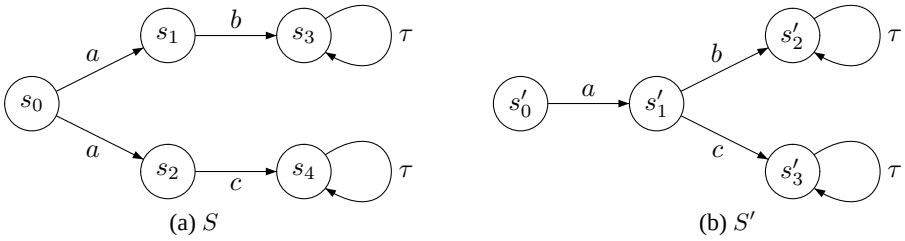


Figure 5. *Deux systèmes de transitions non bisimilaires*

1. Pour être tout à fait formel, il faudrait distinguer les noms des nœuds des noms des états correspondant à ces nœuds.
2. On utilise le même symbole $\rightarrow$ pour les deux systèmes bien que ces deux relations soient différentes : le contexte indique clairement de quelle relation il s'agit.

3. Spécification des propriétés

3.1. *Logiques temporelles*

Une fois l'étape de modélisation terminée, on souhaite vérifier que l'application satisfait un certain nombre de propriétés. Des propriétés comme le deadlock peuvent être vérifiées en explorant le (graphe du) système de transitions. Néanmoins pour des propriétés plus complexes concernant les comportements dynamiques de l'application on utilise des langages logiques adaptés : les *logiques temporelles* [SCH 99, EME 90, HUT 00]. L'intérêt des ces logiques est qu'elles permettent de spécifier des propriétés dans un langage logique le long des exécutions (par exemple d'un système de transitions). On n'a donc pas besoin de connaître la structure interne du système à vérifier pour spécifier des propriétés. La phase de spécification peut ainsi être réalisée distinctement de celle de modélisation du comportement du système.

Dans cette section on note tt la valeur booléenne *vraie* et ff la valeur booléenne *fausse*.

On a vu dans la Section 2.4 qu'on pouvait associer un langage ou un arbre d'exécution à un système de transitions. Ces structures modélisent implicitement la notion de temps sous une forme logique : dans une exécution passant successivement par les états $s_0, s_1 \ldots$ il y a un instant *logique* associé à l'état s_0 qui précède l'instant logique associé à l'état s_1. De même pour la structure arborescente, l'instant logique d'un nœud est le précédent de celui de ces fils. Les logiques temporelles permettent d'exprimer des propriétés dynamiques, faisant référence au temps discret. Pour spécifier des propriétés des exécutions, on utilise une *logique temporelle linéaire* (l'état du système à l'instant suivant de l'instant courant est complètement déterminé). Dans le cas de l'arbre des exécutions, on utilise une *logique temporelle arborescente* (l'état du système à l'instant suivant de l'instant courant n'est pas unique).

Une formule de logique est interprétée sur un modèle. C'est aussi le cas pour les logiques temporelles. Dans le cas de la logique temporelle linéaire les modèles sont des séquences de *propriétés atomiques*. Soit AP un ensemble fini de propriétés atomiques. Une *séquence* de propriétés atomiques $\pi = p_0 p_1 \ldots p_n \ldots$ est une séquence telle que $\forall i \geq 0, p_i \in 2^{AP}$. Une *propriété* P est un ensemble de séquences de propriétés atomiques. On dira que la séquence π satisfait la propriété P ssi $\pi \in P$. On note dans ce cas $\pi \models P$. Dans le cas arborescent, on considère des *arbres* de propriétés atomiques : les nœuds sont des éléments de 2^{AP}. Une propriété P est un ensemble d'arbres et un arbre a satisfait P ssi $a \in P$. On note aussi dans ce cas $a \models P$. Les logiques temporelles linéaires et arborescentes permettent de définir respectivement des ensembles de séquences et des ensembles d'arbres de propriétés. Les définitions précises des logiques temporelles linéaires et arborescentes sont données en Section 3.3 et Section 3.4 .

3.2. *Sûreté et Vivacité*

Les propriétés d'un système sont souvent classées en deux catégories :

– les propriétés de *sûreté* qui expriment que «quelque chose de mauvais n'arrive jamais»,

– les propriétés de *vivacité* qui expriment que «quelque chose de bon est toujours possible».

D'un point de vue formel, une propriété P définit un ensemble de séquences (ou d'arbres). Un tel ensemble est une propriété de sûreté ssi pour toute séquence $\pi \in P$ tous les préfixes finis de π sont aussi dans P. Par exemple : «on n'atteint jamais un état deadlock». Une propriété P et une propriété de vivacité ssi toute séquence finie (du système à étudier) peut être étendue en une séquence de P. Par exemple : «il est toujours possible d'atteindre un état deadlock». Cette classification a un intérêt théorique : dans [ALP 85], les auteurs montrent que n'importe quelle propriété est la conjonction (intersection) d'une propriété de sûreté et d'une propriété de vivacité.

3.3. *LTL : Linear Temporal Logic*

La logique temporelle *linéaire* [EME 90, AUD 90, WOL 90] permet de spécifier des propriétés d'exécutions. On considère un ensemble fini de *propriétés atomiques* AP. La syntaxe de LTL est la suivante :

Définition 4 (Formules de LTL) *Les* formules de LTL *sont définies inductivement par :*

– $\forall p \in AP, p \in LTL$

– *si* $p, q \in LTL$, *alors* $p \vee q, \neg p \in LTL$,

– *si* $p, q \in LTL$*, alors* $\mathsf{X}p, p\,\mathcal{U}\,q \in LTL$*.* □

La signification intuitive des ces formules pour les modalités $\mathsf{X}p, p\,\mathcal{U}\,q$ est la suivante : $\mathsf{X}p$ est vraie sur une séquence $\pi = p_0p_1p_2 \dots p_n \dots$ ssi la sous séquence $p_1p_2 \dots p_n \dots$ satisfait p. $p\,\mathcal{U}\,q$ est vraie sur π ssi p est vraie depuis le début de π jusqu'à un instant où le suffixe de la séquence satisfait q.

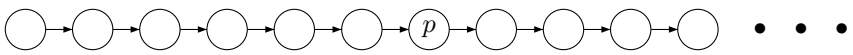
Dans la suite on considère des séquences infinies de propriétés obtenues à partir (des propriétés des états) des exécutions d'un système de transitions étiqueté $S = (Q, s_0, A, \longrightarrow, L)$: une exécution $\sigma = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_1} s_2 \cdots \xrightarrow{a_n} s_{n+1} \cdots$ de S induit une séquence d'états $\rho = s_0s_1 \cdots s_{n+1} \cdots$ à laquelle est associée une séquence de propriétés $L(s_0)L(s_1) \cdots L(s_{n+1}) \cdots$. On note $\rho_i = s_is_{i+1} \cdots$ le *suffixe* de ρ commençant au ième état.

Définition 5 (Sémantique de LTL) *Soit* $\rho = q_0q_1 \cdots q_n \cdots$ *une séquence d'états et* L *telle que* $\forall i \geq 0, L(q_i) \subseteq AP$*. La relation «*$\rho \models \phi$*» se lit «la séquence* ρ *satisfait* ϕ*». Cette relation est définie inductivement et donne la* sémantique *des formules de LTL :*

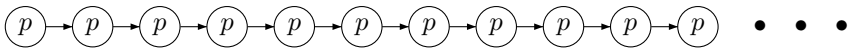
– $p \in AP, \rho \models p \iff p \in L(q_0)$ *;*
– $\rho \models p \vee q \iff \rho \models p$ *ou* $\rho \models q$ *;*
– $\rho \models \neg p \iff \rho \not\models p$ *;*
– $\rho \models \mathsf{X}p \iff \rho_1 \models p$ *;*
– $\rho \models p\,\mathcal{U}\,q \iff \exists j \geq 0, \rho_j \models q$ *et* $(\forall k < j, \rho_k \models p)$*.* □

Certaines formules de LTL sont souvent utilisées car elles correspondent à des propriétés standard et les abréviations suivantes sont souvent rencontrées :

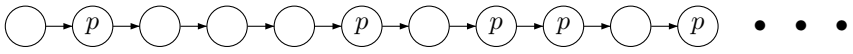
– **fatalement (ou inévitablement)** p **:** $\mathsf{F}p \equiv tt\,\mathcal{U}\,p$



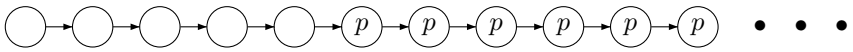
– **toujours** p **:** $\mathsf{G}p \equiv \neg\mathsf{F}\neg p$ (p est un *invariant*)



– **infiniment souvent** p **:** $\mathsf{GF}p$



– **presque partout** p **:** $\mathsf{FG}p$



Par exemple sur le produit synchronisé de la Figure 4 on peut exprimer les propriétés suivantes en LTL :

– «la carte n'est pas rendue si le code tapé est mauvais» : $\mathsf{G}\neg(U.3 \wedge n > 3)$ où $U.3$ est vraie quand le système de transitions U est dans l'état 3 ;

– «après avoir validé on obtient forcément de l'argent» : $\mathsf{G}(U.2 \Longrightarrow \mathsf{F}U.3)$.

Une des propriétés importantes de LTL est qu'elle permet d'exprimer l'équité (Cf. Section 2.3). De plus si deux système sont trace-équivalents ils satisfont les mêmes formules LTL.

3.4. *CTL : Computation Tree Logic*

La logique LTL ne permet pas d'exprimer des propriétés du type «il est possible dans le futur que ...». Il faut pour cela avoir plusieurs futurs possibles dans le modèle de la logique. La logique temporelle arborescente CTL [SCH 99, HUT 00, EME 90] (Computation Tree Logic) permet d'exprimer des propriétés des arbres d'exécution (correspondant éventuellement à un système de transitions).

Définition 6 (Formules de Computation Tree Logic) *Les* formules de CTL *sont les* formules d'états *définies inductivement par :*

– $\forall p \in AP$, p *est une* formule d'état *;*

– *si* p, q *sont des* formules d'état, *alors* $p \vee q$ *et* $\neg p$ *sont des* formules d'état,

– *si* p *est une* formule de chemin, *alors* $\mathsf{E}p$ *et* $\mathsf{A}p$ *sont des* formules d'états,

– *si* p *et* q *sont des* formules d'état, *alors* $\mathsf{X}p$ *et* $p \mathcal{U} q$ *sont des* formules de chemin. □

On considère pour CTL des arbres infinis de propriétés obtenues à partir de l'arbre d'exécution d'un système de transitions étiqueté $S = (Q, s_0, A, \longrightarrow, L)$: à l'arbre d'exécution $A(S)$ on associe l'arbre des propriétés a' étiqueté par les propriétés $L(q)$ des états q de chaque nœud de $A(S)$.

La sémantique de CTL est définie sur les arbres d'exécution. Un arbre d'exécution d'un système de transitions est complètement défini par son état d'origine et par la relation de transition $\longrightarrow$. C'est pourquoi la sémantique de CTL est définie sur les *états* d'un système de transitions (qui caractérise les arbres issus de ces états).

Définition 7 (Sémantique de CTL) *Soit* q_0 *un état de* S. *La* sémantique *de CTL est définie inductivement par :*

– $p \in AP, q_0 \models p \iff p \in L(q_0)$ *;*

– $q_0 \models p \vee q \iff q_0 \models p$ *ou* $q_0 \models q$ *;*

– $q_0 \models \neg p \iff q_0 \not\models p$ *;*

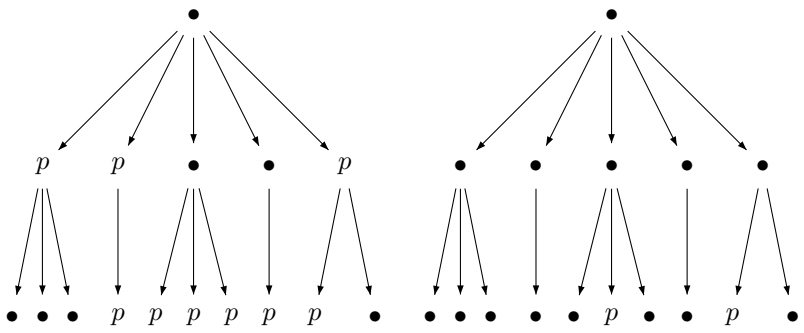
– $q_0 \models \mathsf{E}p \iff \exists \sigma = q_0 \cdots q_n \cdots$ *telle que* $\sigma \models p$ *;*

– $q_0 \models \mathsf{A}p \iff \forall \sigma = q_0 \cdots q_n \cdots$ *on a* $\sigma \models p$ *;*

– $\sigma \models \mathsf{X}p \iff \sigma_1 \models p$ *;*

– $\sigma \models p \mathcal{U} q \iff \exists j \geq 0, \sigma_j \models q$ *et* $(\forall k < j, \sigma_k \models p)$. □

Comme pour LTL, on utilise souvent certaines formules de CTL : $\mathsf{AF}p \equiv \mathsf{A}(\mathsf{tt}\mathcal{U}p)$ (Figure 6.(a)), $\mathsf{EF}p \equiv \mathsf{E}(\mathsf{tt}\mathcal{U}p)$ (Figure 6.(b)), $\mathsf{AG}p \equiv \neg\mathsf{EF}\neg p$ (Figure 7.(a)) et $\mathsf{EG}p \equiv \neg\mathsf{AF}\neg p$ (Figure 7.(b)).



(a) $\mathsf{AF}p \equiv \mathsf{A}(\mathsf{tt}\mathcal{U}p)$ (b) $\mathsf{EF}p \equiv \mathsf{E}(\mathsf{tt}\mathcal{U}p)$

Figure 6. *Abréviations CTL (1)*

Voici quelques exemples de propriétés CTL pour le système de transitions de la Figure 4 :

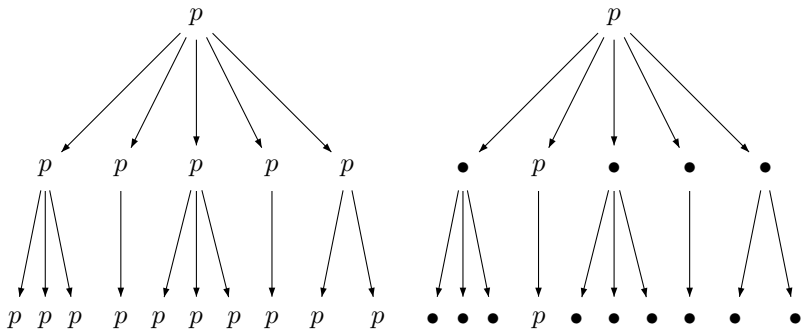
– «la carte n'est jamais rendue quand le code est mauvais» : $\mathsf{AG}\neg(U.3 \wedge n > 3)$;

– «il est possible d'obtenir de l'argent après chaque demande» : $\mathsf{EG}(U.2 \Longrightarrow \mathsf{AF}U.3)$;

– «on obtient toujours de l'argent après une demande» : $\mathsf{AG}(U.2 \Longrightarrow \mathsf{AF}U.3)$;

– «de tout état, on peut revenir à l'état initial» : $\mathsf{AG}\,(\mathsf{EF}\mathit{init})$ où *init* est vraie uniquement pour l'état initial du système.

Une caractéristique importante de CTL est qu'elle «correspond» dans le sens suivant à la notion de bisimulation : deux systèmes de transitions S et S' sont bisimilaires ssi ils satisfont les mêmes formules de CTL.



(a) $\mathsf{AG}p \equiv \neg\mathsf{EF}\neg p$ (b) $\mathsf{EG}p \equiv \neg\mathsf{AF}\neg p$

Figure 7. *Abréviations CTL (2)*

3.5. *LTL vs. CTL*

L'une des différences entre CTL et LTL est que CTL ne permet pas d'exprimer l'équité. LTL par contre ne permet pas d'exprimer la notion de possibilité dans le futur. En fait LTL et CTL ne sont pas comparables.

CTL* est un extension de CTL qui contient aussi LTL.

3.6. *CTL**

Définition 8 (Formules de CTL*) *Les* formules de CTL* *sont les formules d'états définies inductivement par :*

(s_1) $\forall p \in AP, p$ *est une* formule d'état,
(s_2) si p *et* q *sont des formules d'état, alors* $p \vee q$ *et* $\neg p$*, sont des* formules d'état,
(s_3) si p *est une* formule de chemin, *alors* $\mathsf{E}p$ *et* $\mathsf{A}p$ *sont des* formules d'état,
(p_1) si p *est une* formules d'état, *alors* p *est une* formule de chemin,
(p_2) si p *et* q *sont des* formules de chemin, *alors* $p \vee q$ *et* $\neg p$ *sont des* formules de chemin,
(p_3) si p *et* q *sont des* formules de chemin, *alors* $\mathsf{X}p$ *et* $p\,\mathcal{U}\,q$ *sont des* formules de chemin. □

La sémantique de CTL* est une combinaison des sémantiques de LTL et CTL :

Définition 9 (Sémantique de CTL*) *La* sémantique *des* formules de CTL* *est définie inductivement par :*

(s_1) si $p \in AP$ *est une formule d'état,* $q_0 \models p \iff p \in L(q_0)$,
(s_2) $q_0 \models p \vee q \iff q_0 \models p$ *ou* $q_0 \models q$,
(s_2') $q_0 \models \neg p \iff q_0 \not\models p$,
(s_3) $q_0 \models \mathsf{E}p \iff \exists \sigma = q_0 \cdots q_n \cdots$ *telle que* $\sigma \models p$,
(s_3') $q_0 \models \mathsf{A}p \iff \forall \sigma = q_0 \cdots q_n \cdots$ *telle que* $\sigma \models p$,
(p_1) $\sigma = q_0 \cdots q_n \cdots \models p \iff q_0 \models p$,
(p_2) $\sigma \models p \vee q \iff \sigma \models p$ *ou* $\sigma \models q$,
(p_3) $\sigma \models \mathsf{X}p \iff \sigma_1 \models p$,
(p_3') $\sigma \models p\,\mathcal{U}\,q \iff \exists j \geq 0, \sigma_j \models q$ *et* $(\forall k < j, \sigma_k \models p)$. □

Il existe d'autres logiques pour le temps discret : par exemple on peut se reporter à [ARN 92] pour le μ-calcul, ou [LAM 94] pour TLA.

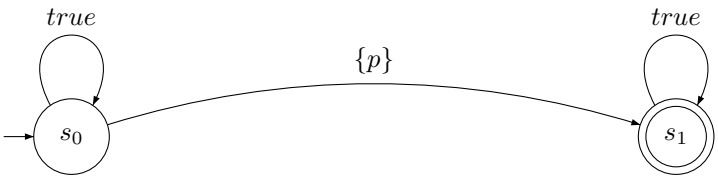


Figure 8. *Automate acceptant* Fp

4. Model-Checking

Le problème du *model-checking* (vérification sur modèle) est le suivant : «étant donnés une formule ϕ (de LTL, CTL, etc) et un système de transitions S, est-ce que S satisfait ϕ ?» (on dit aussi «est-ce que S est un modèle pour ϕ» d'où le nom de model-checking).

Si ϕ est une formule de LTL, le problème du model-checking est précisément : «est-ce que $\forall \sigma \in \mathit{Run}^{\omega}(S), \sigma \models \phi$?»

Dans le cas de CTL il devient : «est-ce que $s_0 \models \phi$?» (où s_0 représente l'arbre d'exécution $A(S)$ issu de l'état initial s_0.)

Dans cette section on donne quelques principes des algorithmes de model-checking de LTL et CTL.

4.1. *Model-checking de LTL*

Le model-checking de LTL est basé sur le principe suivant :

– à une formule ϕ de LTL, on associe un *automate de Büchi* [THO 90] $B_{\neg\phi}$ qui accepte toutes les exécutions ne satisfaisant pas ϕ,

– on synchronise de façon adéquate les système de transitions S et $B_{\neg\phi}$,

– on répond à la question «est-ce que $S \times B_{\neg\phi}$ accepte un langage non vide ?». Si «oui» il y a un mot qui représente une exécution de S qui ne satisfait pas ϕ. Si «non» la propriété ϕ est satisfaite sur le système S.

Automates de Büchi.

Définition 10 (Automate de Büchi [THO 90]) *Un* automate de Büchi A *est tuple* $(Q, s_0, \Sigma, \longrightarrow, F)$ *tel que :*

– Q *est un ensemble* fini *d'états,*

– s_0 *est l'état* initial,

– Σ *est un* alphabet fini,

– $\longrightarrow \subseteq Q \times \Sigma \times Q$ *est la* relation de transition,

– $F \subseteq Q$ *est un ensemble* d'états accepteurs.

Une exécution *de* A *est une* séquence infinie *de transitions. Une* exécution admissible *est une* exécution *passant* infiniment souvent *par un état de* F. *Un mot* $w \in \Sigma^{\omega}$ *est* accepté par A *ssi* w *est la* trace *d'une exécution admissible de* A. *Le* langage accepté par A *est* $L(A)$ *constitué de l'ensemble des mots infinis acceptés par* A. □

Les automates de Büchi permettent de représenter des formules de LTL : il est possible de construire un automate de Büchi acceptant uniquement les séquences qui satisfont une formule LTL. Des exemples sont donnés sur les Figure 8, Figure 9 et Figure 10.

Principe du model-checking de LTL

Comme indiqué précédemment le model-checking de LTL [SCH 99, CLA 99, MER 00] consiste à construire un automate de Büchi acceptant les séquences non valides et à le synchroniser avec le système à vérifier.

Soit ϕ une propriété de LTL et $S = (Q, s_0, A, \longrightarrow, L)$ un système de transitions étiqueté. On construit l'automate de Büchi $B_{\neg\phi}$ acceptant les exécutions ne satisfaisant pas ϕ : les actions de $B_{\neg\phi}$ sont des propriétés des

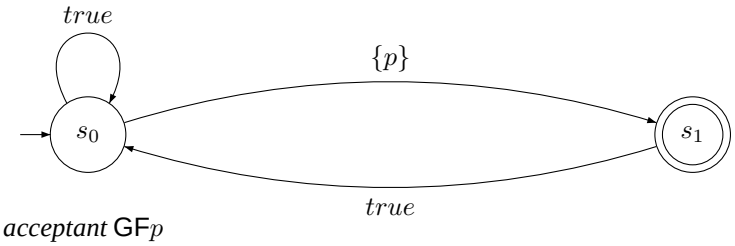


Figure 9. *Automate acceptant* $\mathsf{GF}p$

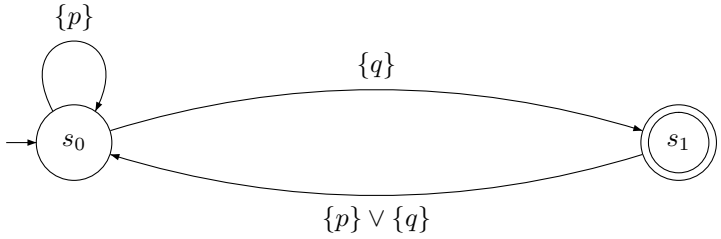


Figure 10. *Automate acceptant* $\mathsf{G}(p\,\mathcal{U}\,q)$

états de S. On «transforme» ensuite le système de transitions S à vérifier en un automate de Büchi où tous les états sont accepteurs. On synchronise les 2 automates de Büchi obtenus de la manière suivante : $(q,s) \xrightarrow{L(q)} (q',s')$ ssi $q \xrightarrow{a} q'$ dans S pour $a \in A$ et $s \xrightarrow{L(q)} s'$ dans $B_{\neg\phi}$. Dans le produit synchronisé $S \times B_{\not\phi}$ si S suit une exécution ρ dont les états sont $q_0 q_1 \cdots q_n$, $B_{\neg\phi}$ suit une exécution de trace $L(q_0)L(q_1)\cdots L(q_n)$ ($B_{\neg\phi}$ n'est pas forcément déterministe). Le produit synchronisé $S \times B_{\neg\phi}$ est un automate de Büchi où (q,s) est un état accepteur ssi s est accepteur dans $B_{\neg\phi}$.

Il suffit ensuite de déterminer si $L(S \times B_{\neg\phi}) = \emptyset$ pour savoir si $S \models \phi$. L'algorithme pour savoir si $L(S \times B_{\neg\phi}) = \emptyset$ est le suivant :

1) on cherche les états accepteurs Z, qui sont sur des cycles dans $S \times B_{\neg\phi}$;

2) on regarde ensuite si l'un des états de Z est accessible à partir de l'état initial de $S \times B_{\neg\phi}$.

La construction dans le cas général de l'automate de Büchi associé à une formule de LTL et d'autres méthodes de model-checking de LTL sont est données dans [MER 00].

Complexité du model-checking de LTL.

Soit $S = (Q, s_0, A, \longrightarrow, L)$ un système de transitions : on note $|S| = |Q| + |\longrightarrow|$. Soit ϕ une formule de LTL : $|\phi|$ correspond au nombre de sous-formules composant ϕ (en décomposant ϕ avec la grammaire abstraite de la Définition 4). L'automate $B_{\neg\phi}$ a une taille de l'ordre de $2^{|\phi|}$. Le produit synchronisé a une taille qui est le produit du nombre d'états de S et de $B_{\neg\phi}$. L'algorithme permettant de tester si $L(K) = \emptyset$ pour un automate de Büchi K est en $\mathcal{O}(|K|)$. Ainsi le model-checking de LTL est en $\mathcal{O}(|S|.2^{|\phi|})$. Le model-checking de LTL est PSPACE-complet.

Des raffinements existent pour minimiser la taille de $B_{\neg\phi}$ [GAS 01]. Enfin les méthodes d'ordres partiels utilisent des propriétés de symétrie des systèmes pour diminuer le nombre de transitions à explorer (voir [GOD 91]).

4.2. *Model-checking de CTL*

Le model-checking de CTL consiste à étiqueter les états d'un système de transitions par les formules qu'il satisfait. Soit ϕ une formule de CTL et $S = (Q, s_0, A, \longrightarrow, L)$ un système de transitions étiqueté.

Étiquetage des états.

L'*étiquetage* d'un état $s \in Q$ consiste à ajouter à $L(s)$ les sous-formules que s satisfait de la façon suivante :

– pour $p \in AP$ l'étiquetage est donné par $L(s)$,
– pour une formule $p \wedge q$ on a $L(s) := L(s) \cup \{p \wedge q\}$ pour tout s tel que $p, q \in L(s)$,
– pour $\neg p$ on a $L(s) := L(s) \cup \{\neg p\}$ pour tout s tel que $p \notin L(s)$,
– pour $\mathsf{EX}q$ on a $L(s) := L(s) \cup \{\mathsf{EX}q\}$ pour tout s tel que $\exists a \in A$ tel que $s \xrightarrow{a} t \wedge q \in L(t)$,
– pour $\mathsf{AX}q$ on a $L(s) := L(s) \cup \{\mathsf{AX}q\}$ pour tout s tel que $\forall a \in A$ si $s \xrightarrow{a} t$ alors $q \in L(t)$,
– pour $p = \mathsf{E}(q\,\mathcal{U}\,r)$ on fait 1) puis $|Q|$ fois le point 2) :

1) si $r \in L(s)$, $L(s) := L(s) \cup \{\mathsf{E}(q\,\mathcal{U}\,r)\}$,

2) $\forall s \in Q$ tel que $q \in L(s)$, si $\exists a \in A$ tel que $s \xrightarrow{a} t$ et $\mathsf{E}(q\,\mathcal{U}\,r) \in L(t)$ alors $L(s) := L(s) \cup \{\mathsf{E}(q\,\mathcal{U}\,r)\}$,

– pour $p = \mathsf{A}(q\,\mathcal{U}\,r)$ on fait 1) puis $|Q|$ fois 2) :

1) si $r \in L(s)$, $L(s) := L(s) \cup \{\mathsf{A}(q\,\mathcal{U}\,r)\}$,

2) $\forall s \in Q$ tel que $q \in L(s)$, si $\forall a \in A$ tel que $s \xrightarrow{a} t$ on a $\mathsf{A}(q\,\mathcal{U}\,r) \in L(t)$ alors $L(s) := L(s) \cup \{\mathsf{A}(q\,\mathcal{U}\,r)\}$.

On peut prouver que lorsque cet algorithme d'étiquetage termine qu'à la fin les états sont étiquetés par les sous-formules de ϕ qu'ils satisfont. Pour déterminer si $S \models \phi$ il suffit de regarder si $\phi \in L(s_0)$.

Complexité du model-checking de CTL.

Pour une formule ϕ de CTL, l'algorithme d'étiquetage pour ϕ est en $\mathcal{O}(|Q| + |T|) = \mathcal{O}(|S|)$. Si l'on fait cette opération pour toutes les sous-formules de ϕ on obtient que le model-checking de CTL est de complexité $\mathcal{O}(|S| \times |\phi|)$. Le model-checking de CTL est polynomial.

4.3. *Model-checking symbolique*

Même si l'algorithme de model-checking de CTL est linéaire en la taille du système de transitions et de la formule à vérifier, les applications pratiques donnent lieu à des modèles avec un nombre considérable d'états : on parle du *problème de l'explosion combinatoire*. Par exemple dans le cas d'un produit synchronisé, le nombre d'états du produit est le produit (au maximum) des nombres d'états de chaque système de transitions.

Un moyen de repousser les limites du model-checking (de CTL par exemple) est de faire du *model-checking symbolique* [MCM 93]. Cette approche consiste à représenter d'une manière *symbolique* l'espace d'états (par ex. «tous les états sources d'une transition a») et de faire des calculs sur ces représentations symboliques.

Principe du model-checking symbolique.

Le principe du model-checking symbolique est de manipuler des ensembles d'états et de réaliser sur ces ensembles des opérations représentant le calcul de l'espace atteignable ou l'ensemble des états satisfaisant une formule de CTL par exemple.

Il est donc nécessaire de trouver une structure de données permettant la manipulation symbolique, et ensuite d'exprimer les algorithmes (atteignabilité, model-checking) à l'aide d'opérations sur ces structures.

Calcul symbolique pour le model-checking de CTL.

On considère ici le model-checking symbolique de CTL. Soit ϕ une formule de CTL et $S = (Q, s_0, A, \longrightarrow, L)$ un système de transitions étiqueté.

On définit l'opérateur *Pre* de la façon suivante : soit $X \subseteq Q$, $\mathit{Pre}(X) = \{q \in Q, \exists a \in A |\ q \xrightarrow{a} q' \wedge q' \in X\}$.

Pour $i \geq 0$, $\mathit{Pre}^{i+1}(X) = \mathit{Pre}(\mathit{Pre}^i(X))$ avec $\mathit{Pre}^0(X) = X$. A l'aide de l'opérateur *Pre* on peut définir inductivement l'ensemble $\mathit{Sat}(\varphi)$ des états satisfaisant φ :

– si $\varphi \in AP$, $\mathit{Sat}(\varphi) = \{q \in Q, \varphi \in L(s)\}$,
– $\mathit{Sat}(\neg\varphi) = Q \setminus \mathit{Sat}(\varphi)$,
– $\mathit{Sat}(\varphi \vee \psi) = \mathit{Sat}(\varphi) \cup \mathit{Sat}(\psi)$,
– $\mathit{Sat}(\mathsf{EX}\varphi) = \mathit{Pre}(\mathit{Sat}(\varphi))$,
– $\mathit{Sat}(\mathsf{AX}\varphi) = Q \setminus \mathit{Pre}\big(Q \setminus \mathit{Sat}(\varphi)\big)$,

– les opérateurs $\mathsf{E}\varphi_1\mathcal{U}\varphi_2$ et $\mathsf{A}\varphi_1\mathcal{U}\varphi_2$ sont plus difficiles à définir ; on peut d'abord remarquer qu'on a les équivalences suivantes :

$$\begin{aligned}\mathsf{E}\varphi_1\mathcal{U}\varphi_2 &\equiv \varphi_2 \vee \big(\varphi_1 \wedge \mathsf{EX}(\mathsf{E}\varphi_1\mathcal{U}\varphi_2)\big)\\ \mathsf{A}\varphi_1\mathcal{U}\varphi_2 &\equiv \varphi_2 \vee \big(\varphi_1 \wedge \mathsf{AX}(\mathsf{A}\varphi_1\mathcal{U}\varphi_2)\big)\end{aligned}$$

Ceci nous permet de définir $\mathit{Sat}(\mathsf{E}\varphi_1\mathcal{U}\varphi_2)$ et $\mathit{Sat}(\mathsf{A}\varphi_1\mathcal{U}\varphi_2)$ comme les plus petits points fixes des équations :

$$\begin{aligned}f(Y) &= Sat(\varphi_2) \cup \big(Sat(\varphi_1) \cap Sat(\mathsf{EX}Y)\big)\\ f'(Y) &= Sat(\varphi_2) \cup \big(Sat(\varphi_1) \cap Sat(\mathsf{AX}Y)\big)\end{aligned}$$

Les fonctions f et f' sont *monotones* et l'espace d'états est fini : on peut montrer dans ce cas que le calcul itératif du plus petit point fixe termine.

Le calcul des ensembles d'états satisfaisant une formule nous permet de répondre à la question «est-ce que $S \models \phi$?» car il suffit de regarder si $s_0 \in \mathit{Sat}(\phi)$: on a ainsi un algorithme de model-checking symbolique.

Les *structures de données* utilisées par l'algorithme symbolique doivent supporter les opérations suivantes :

– l'opération *Pre*,

– les opérations $\cap, \cup, \setminus$,

– un *test d'égalité* pour l'arrêt du calcul (itératif) des points fixes.

Binary Decision Diagrams.

Les *Binary Decision Diagrams (BDD)* sont des structures de données permettant de représenter de façon compacte des expressions booléennes. Il est possible de définir les opérations nécessaires au calcul de *Sat* sur ces structures et ceci de manière efficace. Il existe de nombreux travaux sur l'utilisation des BDDs pour la vérification des systèmes de transitions. On peut consulter [HUT 00] pour un exposé détaillé de ces structures.

5. Outils de model-checking

Il existe de nombreux outils de model-checking disponibles gratuitement. Parmi les plus connus on trouve SPIN [HOL 91, HOL 97] et SMV [MCM 93]. On peut consulter [SCH 99] pour une introduction à ces outils.

Merci à François Laroussinie (LSV, ENS-Cachan) pour ses remarques judicieuses.

6. Bibliographie

[ALP 85] ALPERN B., SCHNEIDER F., « Defining Liveness », *Information Processing Letters*, vol. 21, 1985, p. 181–185.

[ARN 92] ARNOLD A., *Systèmes de transitions et sémantique des processus communicants*, Masson, 1992.

[AUD 90] AUDUREAU E., ENJALBERT P., FARIÑAS DEL CERRO L., *Logique temporelle – Sémantique et validation de programmes parallèles*, E.R.I, MASSON, 1990.

[BOU 91] BOUSSINOT F., DE SIMONE R., « The ESTEREL language », *Proceedings of the IEEE*, vol. 79, n° 9, 1991, p. 1293–1304.

[CAS 87] CASPI P., PILAUD D., HALBWACHS N., PLAICE J. A., « LUSTRE : a Declarative language for Programming Synchronous Systems », *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, Munich (GERMANY), january 1987.

[CLA 99] CLARKE E., GRUMBERG O., PELED D., *Model-cheking*, MIT PRESS, 1999.

[ELL 85] ELLOY J.-P., ROUX O., « ELECTRE : A Language for Control Structuring in Real Time », *The Computer Journal*, vol. 28, n° 5, 1985.

[EME 90] EMERSON E., « Temporal and Modal Logic », VAN LEEUWEN J., Ed., *Handbook of Theoretical Computer Science*, vol. B Formal Models and Semantics, chapitre 16, p. 994–1072, Elsevier Science B.V., 1990.

[GAS 01] GASTIN P., ODDOUX D., « Fast LTL to Büchi Automata Translation », BERRY G., COMON H., FINKEL A., Eds., *Proceedings of the 13th Conference on Computer Aided Verification (CAV'01)*, n° 2102 Lecture Notes in Computer Science, Springer Verlag, 2001, p. 53-65.

[GOD 91] GODEFROID P., WOLPER P., « A Partial Approach to Model Checking », *Logic in Computer Science*, 1991, p. 406-415.

[HOL 91] HOLZMANN G. J., *Design and Validation of Computer Protocols*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[HOL 97] HOLZMANN G. J., « The Model Checker SPIN », *IEEE Transactions on Software Engineering*, vol. 23, n° 5, 1997, p. 279–295, Special Issue : Formal Methods in Software Practice.

[HUT 00] HUTH M. R. A., RYAN M. D., *Logic in Computer Science : Modelling and Reasoning about Systems*, Cambridge University Press, Cambridge, England, 2000.

[LAM 94] LAMPORT L., « The Temporal Logic of Actions », *ACM Transactions On Programming Languages and Systems*, vol. 16, n° 3, 1994, p. 872–923.

[Le 91] LE GUERNIC P., LE BORGNE M., GAUTIER T., LE MAIRE C., « Programming Real-Time Applications with SIGNAL », *Proceedings of the IEEE*, vol. 79, n° 9, 1991, p. 1321–1336.

[MAN 91] MANNA Z., PNUELI A., *The Temporal Logic of Reactive and Concurrent Systems, Specification*, Springer, 1991.

[MCM 93] MC MILLAN K. L., *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.

[MER 00] MERZ S., « Model Checking », CASSEZ F., JARD C., RYAN M., ROZOY B., Eds., *Modeling and Verification of Parallel Processes (MOVEP'00) – Summer School*, CNRS/IRCCyN, Ecole Centrale de Nantes, 2000, p. 51–70.

[SCH 99] SCHNOEBELEN P., *Vérification de logiciles – Techniques et outils du model-checking*, Vuibert, Paris, 1999, Ouvrage collectif.

[THO 90] THOMAS W., « *Handbook of theoretical computer science* », chapitre 4, Automata on infinite objects, Elsevier Science, 1990.

[WOL 90] WOLPER P., « *Approche logique de l'intelligence artificielle Logique Temporelle* », vol. 2 – De la logique modale à la logique des bases de données, chapitre 4, Logique Temporelle, p. 179–, DUNOD, 1990.