Laboratoire Spécification et Vérification
ENS Cachan

Samuel Hornus
DEA algorithmique

Requêtes en logique temporelle

stage effectué sous la direction de Philippe Schnoebelen

Table des matières

LSV, CNRS UMR 8643
ENS de Cachan, 61 av. du Pdt Wilson
94235 CACHAN Cedex, FRANCE
Tél :+33 1 47 40 24 04, Fax :+33 1 47 40 24 64

1 Introduction

La vérification de logiciel devient de plus en plus nécéssaire au fur et à mesure que croît la complexité, la quantité et l'importance économique des systèmes informatiques dans notre quotidien. Différentes techniques sont mises en œuvre pour la vérification de logiciels (plus couramment, on vérifie des abstractions d'algorithmes, rarement des programmes complets). La première n'est pas un moyen de vérifier formellement un logiciel ; c'est le test. On soumet le programme à un jeu de données d'entrée correctement construit pour couvrir le plus grand nombre de cas possibles, ou les cas les plus représentatifs en fonction du genre de bug que l'on souhaite débusquer, et on examine les valeurs de retour du logiciel. Les deux autres sont des méthodes formelles de vérification : la preuve de programme et le model-checking. La preuve de programme s'attache à démontrer, par exemple, l'invariance réelle d'invariants supposés en se servant directement du code source du programme. Le model-checking, lui, demande à ce que l'on transforme le programme ou le code source en modèle de comportement, généralement simplifié. Les propriétés que l'on souhaite vérifier sont alors énoncées dans un certain langage (logique temporelle) et un programme « vérificateur » répond si oui ou non la propriété est vérifiée par le modèle.

Dans le cadre de ce stage de DEA, nous avons souhaité prolonger les travaux innovants de William Chan [Cha00] sur les **requêtes de logique temporelle**. C'est une extension du model checking qui, au lieu de demander « est-ce que le système satisfait ϕ ? », nous permet de demander « pour quelle valeur de X le système vérifie-t-il $\phi(X)$? ». Ici, X n'est pas un paramètre du système, mais un paramètre de la propriété, que l'on cherche à « remplir ». Ces requêtes permettent ainsi, non pas de vérifier une propriété spécifique du modèle mais de prendre connaissance du modèle en le questionnant. Nous pensons, par exemple, que c'est un bon moyen de comprendre un modèle que l'on est en train de « créer ». De plus, les résultats de ces requêtes permettent directement de résoudre un grand nombre d'instances du problème du model-checking, puisque c'est une généralisation directe de ce dernier.

À partir de [Cha00], nous avons pris deux directions relativement indépendantes. D'une part étendre ces travaux à d'autres logiques temporelle. D'autre part, examiner, indépendement des logiques utilisées, quels peuvent être les moyens de résoudre le problème du query-checking (que nous définissons dans la section 3). Quelles questions peut-on se poser sur l'ensemble des solutions à une instance de problèmes ?

Dans la deuxième section, nous commençons par expliquer ce qu'est le model-checking, montrons quelle peut être une utilisation du model-checking (la compréhension d'un modèle algorithmique) et montrons une faiblesse du model-checking pour cette utilisation. Cela nous mène ensuite naturellement, dans la troisième section, à la définition d'une requête de logique temporelle. Nous y présentons et étendons les travaux de Chan, et (pour le treillis des formules propositionnelles essentiellement) ceux de Bruns et Godefroid. Dans la quatrième section, nous nous demandons s'il est possible de décider certaines propriétés importantes des solutions, et si l'on peut trouver une méthode générique de résolution du problème du query-checking. Nous répondons à ces questions en donnant des algorithmes polynomiaux avec oracle. [BG01].

Notre contribution comprend les sections 3 et 4. La section 3 reprend et étend les travaux de Chan aux logiques CTL*, LTL et FCTL. La section 4 est

« nouvelle ».

2 Rappels et définitions

2.1 Vérification et model-checking

Le model checking est une des techniques de vérification de logiciel. Le problème du model checking est la donnée d'un modèle de comportement du système qu'on souhaite vérifier et d'une formule exprimant une propriété que l'on souhaite voir **satisfaite** par le modèle. Ce dernier est généralement donné sous la forme d'une structure de Kripke :

Définition 2.1 *Étant donné un ensemble fini $\mathcal{P}$ de* **propositions atomiques** *(booléennes), une* **structure de Kripke** *est un quintuplet $\mathcal{S} = \langle \mathcal{Q}, Q_0, \mathcal{P}, \rightarrow, l \rangle$ où $\mathcal{Q}$ est un ensemble fini d'***états de contrôle***, $Q_0 \in \mathcal{Q}$ est l'état initial, $\rightarrow$ est une relation binaire sur $\mathcal{Q}$, et $l : \mathcal{Q} \mapsto 2^{\mathcal{P}}$ est un étiquetage de $\mathcal{Q}$ par des propositions de $\mathcal{P}$.*

Les structures de Kripke sont une représentation plus ou moins abstraite du comportement d'un algorithme (plus rarement, d'un programme complet) dont nous voulons tester certaines propriétés. Les états du système à vérifier que nous voulons considérer pour la vérification sont représentés par les états de contrôle de la structure de Kripke. La relation binaire $\rightarrow$ représente les possibilités d'évolution du modèle à partir de chaque état.

Les logiques temporelles sont une famille de languages utilisés pour exprimer des propriétés d'un système évoluant dans le temps. Ces logiques sont bien adaptées à certains types de systèmes (dont les structures de Kripke) [CES86, Pnu77, Pnu81, Lam80]. Nous pouvons y exprimer, par exemple, qu'un évènement a va se produire à la prochaine étape de l'évolution du système ($\mathbf{X}a$), ou que sous certaines conditions b, un évènement c se produira certainement, sans que sa « date » soit connue pour autant ($b \Rightarrow \mathbf{F}c$).

Définition 2.2 Le problème du model checking
Entrée : *Une structure de Kripke $\mathcal{S}$ et une formule de logique temporelle ϕ.*
Sortie : *« Oui » si et seulement si $\mathcal{S}$ satisfait ϕ (ce que nous notons $\mathcal{S} \models \phi$)*

Si par exemple nous voulons vérifier qu'un modèle de comportement d'un feu tricolore satisfasse bien la propriété que les trois feux alternent sans blocage dans un ordre convenu, nous pouvons modéliser le feu avec une structure de Kripke (voir figure 1), et écrire le fait que les feux alternent sans blocage et dans le bon ordre à l'aide d'une formule de logique temporelle (CTL par exemple). Le programme qui implémente le model-checking nous répond alors si le feu tricolore marche bien ou non.

Définition 2.3 *Un* **chemin** *$\pi = q_0 q_1 q_2 ...$ dans $\mathcal{S}$ est un mot infini sur $\mathcal{Q}$ tel que*

$$\forall i \geqslant 0, q_i \rightarrow q_{i+1}$$

On définit également, pour tout $i \geqslant 0$, le chemin $\pi_i = q_i q_{i+1} q_{i+2} ...$ suffixe du chemin π (nous avons $\pi_0 = \pi$). Remarque *: q_0 peut être différent de Q_0.*

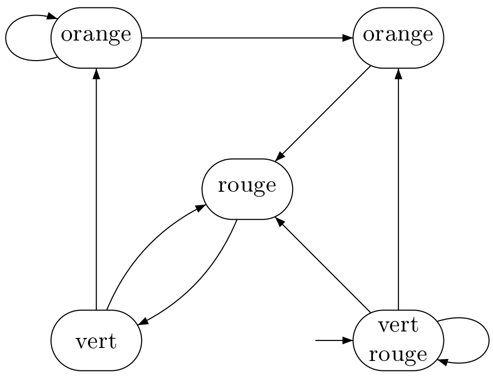

FIG. 1 – Un exemple de structure de Kripke : un feu tricolore($\mathcal{F}_3$)

Une propriété sur le modèle s'énonce par une formule de logique temporelle dont la sémantique porte sur les états de contrôle de la structure de Kripke et les chemins dans cette structure. Plusieurs logiques existent et la complexité du problème du model checking est très variable selon le type de logique utilisée pour **spécifier** les propriétés du modèle.

2.2 Quelques logiques temporelles

Nous présentons ici quelques logiques temporelles, parmi celles qui sont les plus utilisées.

La logique temporelle CTL* permet de mélanger des propriétés portant sur une exécution particulière d'un modèle (un chemin dans la structure de Kripke, *linear time logic*) et sur les possibilités d'évolution d'une exécution (*braching time logic*). On peut exprimer des formules écrites dans d'autres logiques temporelles (LTL, CTL, FCTL,...) dans la logique CTL*. Dans les cas LTL et CTL, aucune réécriture n'est d'ailleurs nécéssaire puisque ces logiques sont des fragments de CTL*.

Soient $q \in \mathcal{Q}$ et ϕ une formule logique. On dira que
- $\mathcal{S} \models \phi$ *ssi* $Q_0 \models \phi$
- $q \models \phi$ *ssi* pour tout chemin π partant de q dans $\mathcal{S}$, on a $\pi \models \phi$ (dont la sémantique est définie plus bas)

Définition 2.4 *La syntaxe de* **CTL*** *est définie par*

$$\phi_S ::= p \mid \neg\phi_S \mid \phi_S \vee \phi_S \mid \mathbf{A}\phi_P \mid \mathbf{E}\phi_P$$

$$\phi_P ::= \phi_S \mid X\phi_P \mid \phi_P \;\mathbf{U}\; \phi_P$$

Les formules de type ϕ_S sont appelées **formules d'état** car leur sémantique porte sur un état de la structure de Kripke. Celles de type ϕ_P sont appellées **formules de chemin**.

On définit de manière inductive la sémantique de CTL* sur un chemin $\pi = q_0q_1q_2...$ dans une structure de Kripke :
- $\pi \models p$ *ssi* $p \in l(q_0)$
- $\pi \models \neg\phi$ *ssi* $\pi \not\models \phi$
- $\pi \models \phi_1 \vee \phi_2$ *ssi* $\pi \models \phi_1$ ou $\pi \models \phi_2$
- $\pi \models \mathbf{X}\phi$ *ssi* $\pi_1 \models \phi$

- $\pi \models \phi_1 \mathbf{U} \phi_2$ *ssi* $\exists pos \geqslant 0, \forall i \in [\![0; pos-1]\!], \pi_i \models \phi_1$ et $\pi_{pos} \models \phi_2$
- $\pi \models \mathbf{A}\phi$ *ssi* pour tout chemin π' partant de q_0 dans $\mathcal{S}$, on a $\pi' \models \phi$
- $\pi \models \mathbf{E}\phi$ *ssi* il existe un chemin π' partant de q_0 dans $\mathcal{S}$ tel que $\pi' \models \phi$

À l'aide des opérateurs définis ci-dessus, nous pouvons en définir d'autres, également utiles ; voir la figure 2.

$\mathbf{F}\phi$	$\Leftrightarrow$	$\top \mathbf{U} \phi$
$\mathbf{G}\phi$	$\Leftrightarrow$	$\neg\mathbf{F}\neg\phi$
$\phi \mathbf{W} \psi$	$\Leftrightarrow$	$(\phi \mathbf{U} \psi) \vee \mathbf{G}\phi$
$\phi \overset{\circ}{\mathbf{U}} \psi$	$\Leftrightarrow$	$\phi \mathbf{U} (\phi \wedge \psi)$
$\phi \overset{\circ}{\mathbf{W}} \psi$	$\Leftrightarrow$	$\phi \mathbf{W} (\phi \wedge \psi)$
$\phi \overline{\mathbf{U}} \psi$	$\Leftrightarrow$	$\phi \mathbf{U} (\neg\phi \wedge \psi)$
$\phi \overline{\mathbf{W}} \psi$	$\Leftrightarrow$	$\phi \mathbf{W} (\neg\phi \wedge \psi)$

FIG. 2 – Divers opérateurs de logique temporelle

$\mathbf{F}\phi$ signifie que ϕ sera vraie un jour ($\exists i \geqslant 0, \pi_i \models \phi$) et se lit « futur ϕ ». $\mathbf{G}$ « toujours » est l'opérateur dual de $\mathbf{F}$. $\mathbf{G}\phi$ signifie que la propriété ϕ est toujours vraie sur le chemin π : $\forall i \geqslant 0, \pi_i \models \phi$. $\phi \mathbf{W} \psi$ « weak until » est une version plus faible de $\phi \mathbf{U} \psi$ qui autorise ϕ à rester toujours vraie et ψ toujours fausse (ou plutôt, non satisfaite). $\overset{\circ}{\mathbf{U}}$, $\overset{\circ}{\mathbf{W}}$, $\overline{\mathbf{U}}$, $\overline{\mathbf{W}}$ sont des variantes de $\mathbf{U}$ et $\mathbf{W}$, qui modifient les conditions de passage de $\models \phi$ à $\models \psi$ le long du chemin π.

Exemple 2.1 *Voici quelques formules CTL* vérifiées ou non par le feu tricolore de la figure 1 :*

$\mathcal{F}_3 \models vert \wedge rouge$	$\mathcal{F}_3 \models \mathbf{EFG}(\neg vert)$
$\mathcal{F}_3 \models orange \Rightarrow \mathbf{EFG}(vert \Rightarrow \mathbf{X}orange)$	$\mathcal{F}_3 \not\models \mathbf{EG}(rouge)$
$\mathcal{F}_3 \models \mathbf{AG}(\neg rouge \vee \neg orange)$	$\mathcal{F}_3 \models \mathbf{G}((rouge \vee orange) \mathbf{W} vert)$

Définition 2.5 **CTL** *est le fragment de CTL* défini par*

$$\phi ::= p \mid \neg\phi \mid \phi \vee \phi \mid \mathbf{AX}\phi \mid \mathbf{A}(\phi \mathbf{U} \phi) \mid \mathbf{EX}\phi \mid \mathbf{E}(\phi \mathbf{U} \phi)$$

C'est le fragment de CTL* dans lequel les opérateurs $\mathbf{X}$ et $\mathbf{U}$ apparaissent directement sous un opérateur $\mathbf{E}$ ou $\mathbf{A}$. Ainsi, toutes les formules CTL sont des formules d'état. Cette dernière propriété est utilisée pour écrire un algorithme efficace du model-checking de CTL.

Définition 2.6 **LTL** *est le fragment de CTL* défini par*

$$\phi ::= p \mid \neg\phi \mid \phi \vee \phi \mid \mathbf{X}\phi \mid \phi \mathbf{U} \phi$$

Toutes les formules LTL sont des formules de chemin. La logique LTL peut donc être utilisée pour décrire le comportement d'une exécution du système en particulier. Alors que la logique CTL permettra de spécifier des comportement plus « généraux », portant sur les possibilités d'évolution de ce système.

Voici quelques exemples de propriétés utiles en pratique :

– propriétés de type *liveness* : Pour exprimer que le feu tricolore passe infiniment souvent par l'état « vert », nous pouvons écrire

$$\mathbf{GF}vert$$

– propriétés de type *safety* : Lorsqu'on souhaite par exemple que deux processus n'accèdent pas à la même ressource en même temps

$$\mathbf{G}\neg(P1 \wedge P2)$$

ou que le feu tricolore n'ait jamais deux feux allumés en même temps

$$\mathbf{G}((vert \wedge \neg orange \wedge \neg rouge) \vee (\neg vert \wedge orange \wedge \neg rouge) \vee (\neg vert \wedge \neg orange \wedge rouge))$$

Pour finir cette section, nous donnons la complexité du problème du model-checking pour les logiques temporelles évoquées ci dessus.
MC(LTL) est PSPACE-complet [SC85].
MC(CTL) est linéaire en temps et en espace [CES86].
MC(CTL*) est PSPACE-complet.
Leur brève évocation ici sera utile dans la section 4 lorsque nous évoquerons la complexité des algorithmes proposés.

3 Requêtes en logique temporelle

Les travaux de Chan portent uniquement sur la logique CTL. Il définit la notion de solution « exacte » (voir l'annexe) et s'intéresse seulement aux requêtes dont les solutions sont exactes dans tous les modèles. Ces requêtes sont appellées requêtes valides. Il montre que décider si une requête CTL est valide est un problème difficile et propose donc un fragment syntaxique de CTL, CTL_v, dont les requêtes sont valides. Enfin il donne pour CTL_v un algorithme bi-linéaire de résolution du problème du query-checking (défini ci-dessous). Dans cette section, nous reprenons ces travaux en les étendant à la logique CTL* (qui inclut LTL), et parlons brièvement du query-checking de FTCL.

Définition 3.1 *Pour éviter une mauvaise interprétation lors de la lecture, nous définissons l'équivalence de deux formules de logique temporelle $\equiv$. Soient ϕ et ψ deux formules de logique temporelle.*

$$\phi \equiv \psi \textit{ ssi } \models \phi \Leftrightarrow \psi$$

$$\textit{où } \models \phi \textit{ signifie } \forall \mathcal{S}, \mathcal{S} \models \phi$$

Nous noterons, pour ϕ et ψ deux formules de logique temporelle

$$\phi \supset \psi \textit{ ssi } \models \phi \Rightarrow \psi$$

3.1 Requêtes et solutions

Le model checking permet de tester si une propriété donnée est ou n'est pas satisfaite par le modèle que l'on « analyse ». Mais, par exemple, remarquant que la propriété $\phi = \mathbf{AG}(erreur \Rightarrow \mathbf{F}alarme)$ n'est pas vérifée, nous aimerions savoir quelle formule, à la place de *erreur*, rendrait ϕ vraie pour le modèle que l'on étudie. Autrement dit, nous voudrions pouvoir répondre à la question $\gamma = \mathbf{AG}(? \Rightarrow \mathbf{F}alarme)$ (qu'est-ce qui déclenche une alarme ?). Nous dirons plutôt que nous voulons trouver la ou les solutions de la **requête** γ, où les « solutions » sont des formules propositionnelles. C'est l'objet du **query checking**, introduit par Chan [Cha00].

Définition 3.2 *Une* **formule propositionnelle** f *est une formule booléenne sur* $\mathcal{P}$

$$f ::= p \mid f \vee f \mid \neg f$$

Définition 3.3 *Soit une logique temporelle TL contenant au moins la logique booléenne sur un ensemble de propositions atomiques* $\mathcal{P}$*. Une* **requête** *TL est une formule TL dans laquelle on a remplacé une proposition atomique par le symbole ? (joker).*

$$\gamma = \mathbf{AG}(? \Rightarrow \mathbf{AF}alarme) \text{ est un exemple de requête CTL}$$

Soit f une formule propositionnelle sur $\mathcal{P}$. Nous notons $\gamma(f)$ la formule de logique temporelle $\gamma[f/?]$ obtenue en remplaçant ? par f dans l'expression de la requête γ.

Définition 3.4 *Une* **solution** f *d'une requête* γ *dans une structure de Kripke* $\mathcal{S}$ *donnée est une formule propositionnelle sur* $\mathcal{P}$ *telle que*

$$\mathcal{S} \models \gamma(f)$$

Nous pouvons dès lors introduire une première définition du problème du query-checking.

Définition 3.5 Le problème du query-checking, pré-version
Entrée : *Une structure de Kripke* $\mathcal{S}$ *et une requête de logique temporelle* γ*.*
Sortie : *L'ensemble des solutions de* γ *dans le modèle* $\mathcal{S}$*, i.e.* $\{f, \mathcal{S} \models \gamma(f)\}$*.*

Voici deux lemmes essentiels sur les solutions d'une requête. Leur version CTL est présentée avec preuve dans [Cha00].

Définition 3.6 *Nous dirons qu'une requête est* **posivite** *(resp.* **négative***) si le joker apparait sous un nombre pair (resp. impair) de négations* ($\neg$)*.*

Lemme 3.1 *Soient* γ *une requête CTL* et* f *et* g *deux formules propositionnelles.*

- $[f \Rightarrow g] \Rightarrow [\gamma(f) \supset \gamma(g)]$ *si* γ *est positive.*
- $[f \Rightarrow g] \Rightarrow [\gamma(g) \supset \gamma(f)]$ *si* γ *est négative.*

Nous savons que nous pouvons réécrire toute formule de logique temporelle de telle sorte que le symbole de négation $\neg$ ne soit porté que par les propositions atomiques ou par le joker (on utilise la dualité des opérateurs temporels). Considérons qu'une requête négative soit sous cette forme. Alors, pour calculer ses solutions dans $\mathcal{S}$, nous pouvons construire une nouvelle requête en supprimant le $\neg$ devant le joker, calculer les solutions de cette nouvelle requête (qui est positive), et finalement considérer les négations des ces solutions, qui sont les solutions de la requête négative initiale.

Dans un souci de simplification, **nous ne considérons désormais que des requêtes positives.** Nous venons de voir que cela n'altère en rien la généralité des résultats présentés ci-après. Par exemple, l'ensemble des solutions d'une requête négative sera clos par le bas et non clos par le haut...

Lemme 3.2 *Soient $\mathcal{S}$ un modèle et γ une requête CTL* positive.*

- $[\mathcal{S} \models \gamma(\bot)] \Leftrightarrow [\forall f, \mathcal{S} \models \gamma(f)]$
- $[\mathcal{S} \models \gamma(\top)] \Leftrightarrow [\exists f, \mathcal{S} \models \gamma(f)]$

Les preuves de ces deux lemmes sont très semblables à celles de Chan pour la logique CTL. Pour le lemme 3.1, on raisonne par induction sur la structure de γ. Le lemme 3.2 se déduit du lemme 3.1.

Le lemme 3.1 met en évidence une certaine structure de l'ensemble des solutions. En effet nous pouvons remarquer que lorsque f est solution de γ dans $\mathcal{S}$, alors toutes les formules logiquement impliquées par f sont également solutions de γ dans $\mathcal{S}$.

Observation 3.3 *L'ensemble des solutions d'une requête γ dans une structure de Kripke $\mathcal{S}$ est clos par le haut pour la relation d'implication.*

Le lemme 3.2 nous permet de savoir rapidement si une requête admet une solution dans un modèle donné ; il suffit pour cela de tester si $\mathcal{S} \models \gamma(\top)$, en utilisant un algorithme de model-checking approprié à la logique utilisée pour écrire la requête.

3.2 Requêtes valides et solutions minimales

Définition 3.7 *Une solution f de γ dans $\mathcal{S}$ est dite* **minimale** *si c'est une solution de γ dans $\mathcal{S}$ et si toute formule propositionnelle impliquant strictement f n'est pas solution de γ dans $\mathcal{S}$.*

$$f \textit{ est solution minimale de } \gamma \textit{ dans } \mathcal{S} \textit{ ssi } \forall g\,((g \Rightarrow f \wedge \mathcal{S} \models \gamma(g)) \Rightarrow f \Leftrightarrow g)$$

Nous pouvons maintenant redéfinir le problème du query-checking

Définition 3.8 Le problème du query-checking
Entrée : *Une structure de Kripke $\mathcal{S}$ et une requête de logique temporelle γ.*
Sortie : *L'ensemble des solutions minimales de γ dans le modèle $\mathcal{S}$, i.e.*

$$\{f, (\mathcal{S} \models \gamma(f)) \textit{ et } f \textit{ minimale}\}$$

Dans la suite de cette section, nous nous intéressons aux requêtes ayant une unique solution minimale. Ces requêtes sont intéressantes car leur utilisation est plus facile, d'une part d'un point de vue de complexité et d'autre part pour l'interprétation des résultats. Alors qu'un algorithme naïf pour le problème du query-checking est de complexité doublement exponentielle, résoudre ce problème pour le fragment CTL_v [Cha00] se fait en temps bi-linéaire $(|\mathcal{S}| \cdot |\gamma|)$. De plus l'unicité de la solution permet, quand on interprète les résultats, de parler de *la bonne* solution. Ce type de requête (à solution unique) correspond peut-être à un ensemble de requêtes qui « ont un sens » lorsqu'on exprime de vive voix ce qu'elles demandent à propos du modèle. De manière plus pragmatique, nous pouvons émettre l'idée qu'il est plus facile de considérer l'unique solution minimale d'une requête que de rester indécis devant un grand nombre de solutions minimales dont on ne saura que faire.

Définition 3.9 *Une requête γ est dite* **valide** *si elle admet une unique solution minimale dans tous les modèles.*

La restriction du problème du query-checking aux requêtes valides semblent en simplifier sa mise en oeuvre algorithmique. C'est du moins ce que nous suggère l'algorithme – linéaire en temps – de Chan pour une certaine classe de requêtes CTL valides. Nous donnons la preuve complète du théorème 1 de Chan ainsi que ses deux variantes pour les logiques CTL* et LTL. Nous en déduisons que décider la validité d'une requête est un problème difficile et justifie la définition *syntaxique* d'une classe de requête CTL* valides (par construction).

Théorème 3.1 *(preuve du théorème 1 de W. Chan). La validité d'une formule CTL se réduit à la validité d'une requête CTL.*

Preuve : *Soit ϕ une formule CTL. Montrons que ϕ est valide ssi pour toute proposition atomique x n'apparaissant pas dans ϕ, la requête CTL $\gamma = \mathbf{A}((x \vee \phi)\ \mathbf{W}\ ?)$ a toujours une solution et est distributive sur la conjonction (c'est à dire que γ est une requête valide).*

$\boxed{\Rightarrow}$ *γ a une solution car $\gamma(\top)$ est une formule valide, et γ est bien distributive sur la conjonction.*

$\boxed{\Leftarrow}$ *Raisonnons par contraposée. Supposons que ϕ n'est pas une formule valide. Soit une structure de Kripke $\mathcal{S} = \langle \mathcal{Q}, Q_0, \mathcal{P}, \rightarrow, l\rangle$ telle que $\mathcal{S} \not\models \phi$. On peut supposer que $x \notin \mathcal{P}$. Soit $\sigma \notin \mathcal{Q}$; considérons la structure $\mathcal{S}' = \langle \mathcal{Q} \cup \{\sigma\}, \sigma, \mathcal{P} \cup \{x\}, \rightarrow', l'\rangle$. Avec*

$$\rightarrow' = \rightarrow \cup \{(\sigma, Q_0)\}$$

$$\begin{cases} l'(q) = l(q) & si\ q \in \mathcal{Q}\backslash\{\sigma, Q_0\} \\ l'(\sigma) = \{x\} \\ l'(Q_0) = l(Q_0) \cup \{\neg x\} \end{cases}$$

Posons $p_1 = x$ et $p_2 = \neg x$. Nous avons $\mathcal{S}' \models \gamma(p_1)$ et $\mathcal{S}' \models \gamma(p_2)$ donc $\mathcal{S}' \models \gamma(p_1) \wedge \gamma(p_2)$. Mais $\mathcal{S}' \not\models \gamma(p_1 \wedge^\gamma p_2)$ ∎

Ce théorème montre que décider la validité d'une requête CTL est un problème EXPTIME-complet, ce qui est dû à la complexité identique du problème de décision de la validité d'une **formule** CTL [Eme90].

On donne deux versions CTL* et LTL du théorème 1 de Chan :

Proposition 3.1 *Décider la validité d'une requête CTL* est un problème 2EXPTIME-complet.*

Preuve : *Commençons par montrer que c'est un problème 2EXPTIME-facile. Soit γ une requête CTL*. Nous savons que γ est valide si et seulement si $\models \gamma(\top)$ et $\gamma(p_1)\wedge\gamma(p_2) \equiv \gamma(p_1 \wedge^\gamma p_2)$, où p_1 et p_2 n'apparaissent pas dans γ, sont des formules valides. Décider la validité d'une formule CTL* est un problème 2EXPTIME-complet donc notre problème initial est bien 2EXPTIME-facile.*

Montrons que c'est un problème 2EXPTIME-dur. Soit ϕ une formule CTL. Soit la requête CTL* $\gamma = \mathbf{A}((x \vee \phi)\ \mathbf{W}\ ?)$. On utilise la même preuve qu'au théorème 1 pour montrer que ϕ valide $\Leftrightarrow \gamma$ valide.*

On peut donc conclure que décider la validité d'une requête CTL est un problème 2EXPTIME-complet* ∎

Proposition 3.2 *Décider la validité d'une requête LTL est un problème PSPACE-complet.*

Preuve : *identique à la précédente, en utilisant $\gamma = (x \vee \phi)\ \mathbf{W}\ ?$* ∎

3.3 CTL_v^*, un fragment dont les requêtes sont valides

Définition 3.10 *Nous dirons qu'une requête γ est* **distributive sur la conjonction** *si, pour toutes propositions p_1 et p_2, nous avons*

$$\gamma(p_1 \wedge p_2) \equiv \gamma(p_1) \wedge \gamma(p_2)$$

La définiton d'une requête valide est claire mais difficilement utilisable lorsqu'on cherche à savoir concrètement si une requête donnée est valide. Voici un lemme [Cha00] qui donne une caractérisation plus précise d'une requête valide.

Lemme 3.4 *Une requête est valide si et seulement si elle a une solution dans tout modèle et elle est distributive sur la conjonction.*

Savoir si une requête admet une solution dans tout modèle revient (grâce au lemme 3.2) à savoir si $\gamma(\top) \Leftrightarrow \top$. Intuitivement, la distributivité sur la conjonction permet de « regrouper » toutes les solutions en une seule solution minimales.

Nous présentons maintenant un fragment de CTL* dont les requêtes sont valides. Dans la suite, ϕ est une **formule** CTL* et β est une **requête** CTL* valide.

1. $\boxed{\gamma = ?}$ est une **requête** valide car $\gamma(\top) \equiv \top$ est une **formule** valide et la distributivité sur la conjonction est évidente.
2. $\boxed{\gamma = \neg ?}$ est une requête *négative* valide.
3. $\boxed{\gamma = \phi \vee \beta \text{ ou } \gamma = \beta \vee \phi}$ est une **requête** valide. En effet :
   - $\gamma(\top)$ est valide (car $\beta(\top)$ l'est)
   - $\gamma(p_1) \wedge \gamma(p_2) \equiv \phi \vee (\beta(p_1) \wedge \beta(p_2)) \equiv \phi \vee \beta(p_1 \wedge p_2) \equiv \phi \vee \beta(p_1 \wedge p_2) \equiv \gamma(p_1 \wedge p_2)$
4. $\boxed{\gamma = \mathbf{X}\beta}$ est une **requête** valide. En effet :
   - $\gamma(\top) \equiv \mathbf{X}\beta(\top) \equiv \mathbf{X}\top \Leftrightarrow \top$
   -
   $$\begin{array}{lll} & \gamma(p_1) \wedge \gamma(p_2) & \\ \equiv & \mathbf{X}\beta(p_1) \wedge \mathbf{X}\beta(p_2) & \\ \equiv & \mathbf{X}(\beta(p_1) \wedge \beta(p_2)) & \\ \equiv & \mathbf{X}\beta(p_1 \wedge p_2) & \text{car } \beta \text{ est une requête valide} \\ \equiv & \gamma(p_1 \wedge p_2) & \end{array}$$
5. $\boxed{\gamma = \beta \ \mathbf{U} \ \phi}$ et $\boxed{\gamma = \beta \ \mathbf{W} \ \phi}$ sont des **requêtes** valides. En effet :
   - $\gamma(\top) = \beta(\top) \ \mathbf{U} \ \phi$. $\beta(\top)$ étant valide, la sémantique de $\mathbf{U}$ nous montre que $\gamma(\top)$ est aussi valide.
   - $\gamma(p_1) \wedge \gamma(p_2) \equiv (\beta(p_1) \ \mathbf{U} \ \phi) \wedge (\beta(p_1) \ \mathbf{U} \ \phi) \equiv (\beta(p_1) \wedge \beta(p_2)) \ \mathbf{U} \ \phi \equiv \gamma(p_1 \wedge p_2)$
6.

$\phi \ \mathbf{U} \ \beta$	○	$\phi \ \overset{\circ}{\mathbf{U}} \ \beta$	○	$\phi \ \overline{\mathbf{U}} \ \beta$	•
$\phi \ \mathbf{W} \ \beta$	○	$\phi \ \overset{\circ}{\mathbf{W}} \ \beta$	○	$\phi \ \overline{\mathbf{W}} \ \beta$	•
$\beta \ \mathbf{U} \ \phi$	•	$\beta \ \overset{\circ}{\mathbf{U}} \ \phi$	•	$\beta \ \overline{\mathbf{U}} \ \phi$	○
$\beta \ \mathbf{W} \ \phi$	•	$\beta \ \overset{\circ}{\mathbf{W}} \ \phi$	•	$\beta \ \overline{\mathbf{W}} \ \phi$	○

• requête valide ○ requête non valide.

Expliquons « graphiquement » pourquoi, par exemple, $\phi\ \overset{\circ}{\mathbf{U}}\ \beta$ n'est pas une requête valide. Dans le tableau ci-dessous, les rangées représentent une même exécution d'un modèle, et chaque colonne est une étape de cette exécution (une étape de ce chemin dans la structure de Kripke). Une formule de logique temporelle dans une case indique que cette formule est vraie à cette étape. Un vide indique que l'on ne s'intéresse pas à la vérité de la formule temporelle.

ϕ	ϕ	ϕ	ϕ	...
$\neg\beta(a)$	$\beta(a)$	$\neg\beta(a)$	$\neg\beta(a)$	...
	$\neg\beta(b)$		$\beta(b)$	...

Nous avons bien que ce chemin vérifie $\phi\ \overset{\circ}{\mathbf{U}}\ \beta(a)$ et $\phi\ \overset{\circ}{\mathbf{U}}\ \beta(b)$ mais il ne vérifie pas $\phi\ \overset{\circ}{\mathbf{U}}\ (\beta(a) \wedge \beta(b)) \equiv \phi\ \overset{\circ}{\mathbf{U}}\ \beta(a \wedge b)$. La formule $\phi\ \overset{\circ}{\mathbf{U}}\ \beta$ n'est donc pas une requête valide. Avec ce même style de « preuve », nous pouvons montrer la validité ou la non-validité des formules de logique temporelle des tableaux ci-dessus et ci-dessous.

Remarquons que $\beta\ \overset{\circ}{\mathbf{W}}\ \phi \equiv \phi\ \mathbf{R}\ \beta$. Ainsi $\boxed{\phi\ \mathbf{R}\ \beta}$ est une requête valide. **R** est l'opérateur *release*. Soit π un chemin dans $\mathcal{S}$. $\pi \models \phi\ \mathbf{R}\ \beta$ signifie que $\exists i \in [0..\infty], \forall j \leqslant i, \pi_j \models \beta$ et,si $i \neq \infty, \pi_i \models \phi$.

7. $\boxed{\gamma = \mathbf{A}\beta}$ est une **requête** valide.

$\boxed{\boxed{\gamma = \mathbf{E}\beta}}$ n'est **pas** une requête valide.

Nous définissons une classe de requêtes valides dites **persistantes**, qui nous permet d'agrandir notre collection de requêtes valides.

Définition 3.11 *Une formule/requête ρ est dite* **persistante** *si pour tout chemin π*

$$\pi \models \rho \Leftrightarrow \forall i \geqslant 0, \pi_i \models \rho$$

Notons que toute requête persistante est aussi une requête valide. Dans la suite, ρ désigne une requête persistante, β une requête valide, et ϕ une formule CTL*.

1. $\boxed{\gamma = \mathbf{G}\beta}$ et $\boxed{\gamma = \mathbf{A}\rho}$ sont des requêtes persistantes.
2. $\boxed{\gamma = \mathbf{X}\rho}$ et $\boxed{\gamma = \mathbf{F}\rho}$ sont des requêtes persistantes.
3. $\boxed{\gamma = \phi \vee \rho}$ est une requête persistante si ϕ est persistante.
4.

$\phi\ \mathbf{U}\ \rho$	●●	$\phi\ \overset{\circ}{\mathbf{U}}\ \rho$	●	$\phi\ \overline{\mathbf{U}}\ \rho$	●
$\phi\ \mathbf{W}\ \rho$	●●	$\phi\ \overset{\circ}{\mathbf{W}}\ \rho$	●	$\phi\ \overline{\mathbf{W}}\ \rho$	●●
$\rho\ \mathbf{U}\ \phi$	●	$\rho\ \overset{\circ}{\mathbf{U}}\ \phi$	●	$\rho\ \overline{\mathbf{U}}\ \phi$	○
$\rho\ \mathbf{W}\ \phi$	●	$\rho\ \overset{\circ}{\mathbf{W}}\ \phi$	●	$\rho\ \overline{\mathbf{W}}\ \phi$	●

●● requête persistante ● requête valide.

De même, $\boxed{\phi\ \mathbf{R}\ \rho}$ et $\boxed{\rho\ \mathbf{R}\ \phi}$ sont des requêtes valides.

3.4 Query-checking de $FCTL$

Dans le model-checking de $FCTL$, nous utilisons simplement le model-checking de CTL sur un ensemble restreint Q' d'états de la structure de Kripke $\mathcal{S}$.

Nous signalons simplement que le query-checking de $FCTL$ se fait de même, en utilisant le query-checking de CTL sur le même ensemble restreint Q'. Nous pouvons par exemple réutiliser l'algorithme [Cha00] pour le fragment CTL_v de CTL directement.

4 Calcul des solutions

Nous avons présenté ce que sont les requêtes en logique temporelle, et aperçu les difficultés qui apparaissent lorsqu'on veut résoudre le problème du query-checking : la taille doublement exponentielle de l'espace des solutions et la complexité du problème de validité d'une requête notemment. La notion de requête valide semble être intéressante dans la mesure où, dans le cadre CTL, elle mène à un algorithme linéaire (voir l'annexe A). Nous mettons en évidence un fragment syntaxique de CTL*, CTL_v^* (et implicitement, un fragment syntaxique de LTL) dont les requêtes sont valides, c'est à dire qu'elles n'ont qu'une seule solution minimale quelque soit le modèle dans lequel on cherche la solution. Mais nous ne donnons pas d'algorithme pour d'autre fragment que CTL_v, pour lequel Chan donne un algorithme de complexité linéaire.

Nous nous intéressons maintenant à des questions d'ordre plus général. Comment calculer toutes les solutions minimales d'un problème de query-checking ? Une solution donnée est elle minimale ? Combien de solutions minimales admet telle requête dans tel modèle ? Nous présentons également un algorithme intéressant permettant de savoir si une requête admet une unique solution minimale ou bien plusieurs dans un modèle donné. Dans le cas où la solution minimale est unique, cette solution est, de plus, explicitement donnée.

Voici une proposition qui montre que le nombre de solutions minimales peut être exponentiel en la taille du problème. Il nous conforte dans l'idée que l'algorithme exhaustif décrit dans la section 4.2 est de complexité minimale pour la recherche de toutes les solutions minimales d'un problème de query-checking, sans du moins en donner de preuve.

Proposition 4.1 *La requête $\gamma = \mathbf{EG}?$ admet 2^n solutions minimales dans la structure de Kripke de la figure 3, qui contient $2n+1$ propositions atomiques et $2n+2$ états. Ces solutions sont de la forme*

$$\left(p_0 \vee \bigvee_{i=1}^{n} \nu_i\right) \quad \textit{où } \nu_i = p_{2i} \textit{ ou } p_{2i-1}$$

Ces formules sont toutes sur l'étage $n+1$ du treillis, et toute solution pour le modèle de la figure 3 contient au moins $n+1$ valuations, donc elles sont toutes minimales.

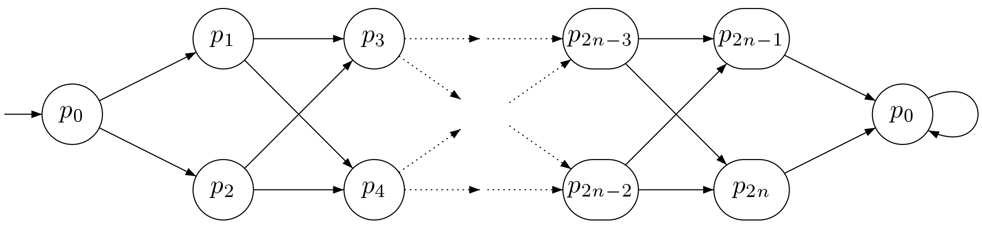

FIG. 3 – Un nombre exponentiel de solutions minimales

4.1 Le treillis des formules propositionnelles

Dans cette section nous nous intéressons à la structure de l'espace des solutions : les formules propositionnelles sur $\mathcal{P}$, où $\mathcal{P}$ un ensemble fini de propositions atomiques.

Rappelons qu'une **valuation** est une application $\mathcal{P} \mapsto \{\bot, \top\}$ qui donne une valeur de vérité à chaque proposition atomique. C'est par la fonction l de la structure de Kripke $\mathcal{S} = \langle \mathcal{Q}, Q_0, \mathcal{P}, \rightarrow, l\rangle$ qu'à chaque état de $\mathcal{Q}$ est associée une valuation v_q

$$\forall p \in \mathcal{P}, v_q(p) = \begin{cases} 0 & \text{si } p \in l(q) \\ 1 & \text{sinon} \end{cases}$$

Appellons $\mathcal{V}$ l'ensemble des valuations sur $\mathcal{P}$.

Nous pouvons aussi voir une formule propositionnelle f comme l'ensemble des valuations qui satisfont cette formule. Nous verrons donc désormais les formules propositionnelles à la fois comme formule logique et comme ensemble de valuation(s).

Définition 4.1 *Le cardinal d'une formule f est le nombre de valuations qui la satisfont.*

$$|f|_{sat} = |[\![f]\!]|$$

$$\textit{où } [\![f]\!] = \{v \in \mathcal{V}, v \models f\}$$

Pour toute valuation v, on a $v \models f$ *ssi* $v \in [\![f]\!]$

Définition 4.2 *Le* **treillis des formules propositionnelles** *est le treillis $(\mathcal{P}(\mathcal{V}), \hookrightarrow)$ où $\mathcal{P}(\mathcal{V})$ est l'ensemble des parties de $\mathcal{V}$ et*

$$X \hookrightarrow Y \ \textit{ssi} \begin{cases} X \subset Y \\ \textit{et} \\ |X| + 1 = |Y| \end{cases}$$

La figure 4 montre ce treillis lorsque $\mathcal{P}$ contient deux propositions atomiques.

Le treillis contient $2^{2^{|\mathcal{P}|}}$ noeuds, qui sont les solutions potentielles d'une requête γ dans une structure de Kripke $\mathcal{S}$.

Nous nous représentons le treillis *par étages* car cela nous servira par la suite. L'*étage n* du treillis est l'ensemble des formules contenant n valuations. Nous nous représentons graphiquement un étage en alignant ses formules sur une même ligne (cf figure 4).

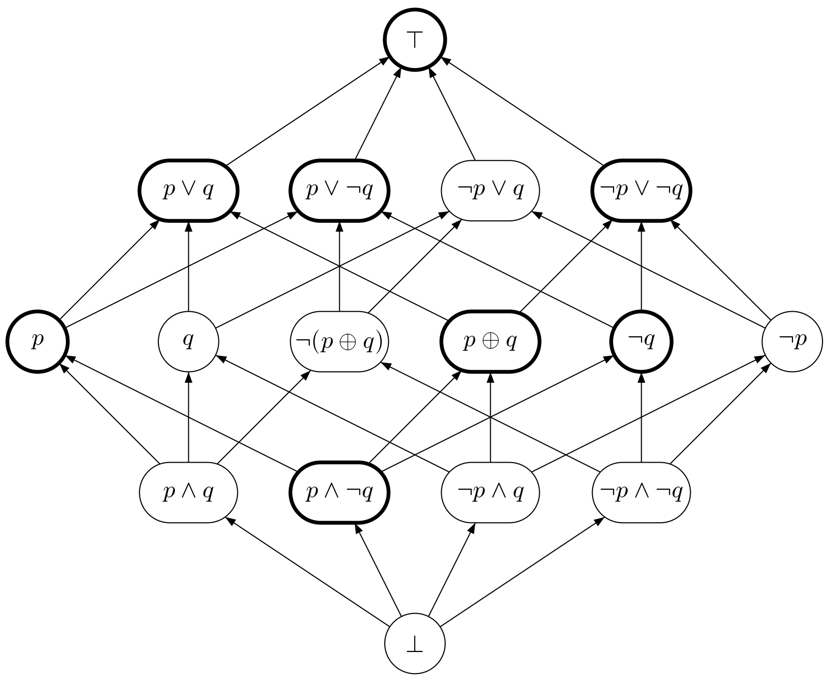

FIG. 4 – Le treillis des formules pour $\mathcal{P} = \{p, q\}$. $\oplus$ est l'opérateur ou-exclusif. $p \oplus q \Leftrightarrow (\neg p \wedge q) \vee (p \wedge \neg q)$

Il y a donc $2^{|\mathcal{P}|} + 1$ étages dans le treillis, et l'étage n contient $C^{n}_{2^{|\mathcal{P}|}}$ formules propositionnelles (modulo $\Leftrightarrow$) ou ensembles de valuations.

Nous avons vu que l'ensemble des solutions de γ dans $\mathcal{S}$ est clos par le haut pour la relation d'implication (et donc également pour la relation $\hookrightarrow$, car $\Rightarrow$ est la clôture transitive et réflexive de $\hookrightarrow$). Nous avons mis en évidence (en gras dans la figure 4) un ensemble clos par le haut d'élément [1] minimal $p \wedge \neg q$.

Dans la suite de cette section, nous cherchons à trouver les solutions minimales d'une requête en parcourant ce treillis.

4.2 Un algorithme générique

Nous présentons cet algorithme relativement naïf de recherche de toutes les solutions minimales d'une requête dans le but d'aider à la compréhension de l'algorithme « partiel » présenté plus loin. Nous ne savons pas s'il existe un meilleur algorithme pour rechercher exhaustivement les solutions minimales d'une requête γ dans un modèle $\mathcal{S}$ donné.

Dans la suite, les complexités des problèmes sont exprimées en utilisant implicitement l'oracle du problème du model-checking associé à la logique contenant la requête, et dont les complexités ont été présentées à la fin de la section 2.

Soit f une formule propositionnelle. Nous noterons $f^{\uparrow}$ l'ensemble des surensembles (la clôture par le haut) de f

$$f^{\uparrow} = \{g, f \Rightarrow g\}$$

[1]Notons qu'un ensemble clos par le haut peut avoir plusieurs éléments minimaux.

et $f^{\downarrow}$ l'ensemble des sous-ensembles (la clôture par le bas) de f

$$f^{\downarrow} = \{g, g \Rightarrow f\}$$

Soit f une formule propositionnelle. Nous pouvons grâce à l'oracle savoir si $\mathcal{S} \models \gamma(f)$ ou pas. Il est important de remarquer que si c'est le cas, nous savons d'emblée que toutes les formules de $f^{\uparrow}$ sont également solutions de γ dans $\mathcal{S}$. De manière duale, si f n'est pas solution, nous pouvons éliminer d'un coup l'ensemble $f^{\downarrow}$ de l'ensemble des solutions potentielles.

Lemme 4.1 *Soit $\mathcal{S} = \langle \mathcal{Q}, Q_0, \mathcal{P}, \rightarrow, l \rangle$ une structure de Kripke et γ une requête. Soit f une formule propositionelle et v une valuation n'apparaissant pas dans les étiquettes des états de $\mathcal{Q}$ ($v \notin \{l(q), q \in \mathcal{Q}\}$). Alors*

$$\mathcal{S} \models \gamma(f) \Leftrightarrow \gamma(f \cup \{v\})$$

Le lemme 4.1 nous montre que les solutions minimales d'une requête sont à rechercher parmi les sous ensembles de $l(\mathcal{Q})$. Ce résultat est très intéressant car il nous permet de restreindre fortement l'espace des solutions potentielles à parcourir : nous savons que les solutions minimales sont à rechercher parmi les sous-ensemble de la formule $f_{\mathcal{S}} = \{v_q, q \in \mathcal{Q}\}$. Nous pouvons donc nous restreindre à parcourir le sous-treillis $f_{\mathcal{S}}^{\downarrow}$ du treillis des formules propositionnelles.

Nous pouvons considérer que dans le pire des cas, $|\mathcal{P}| = |\mathcal{Q}|$ (car nous attribuons, dans la structure de Kripke $\mathcal{S}$) une valuation par état à l'aide de la fonction l). Le treillis $f_{\mathcal{S}}^{\downarrow}$ que nous devons parcourir contient $2^{|\mathcal{Q}|}$ noeuds et est très petit devant la taille du treillis initial ($2^{2^{\mathcal{P}}}$ noeuds). Nous gagnons un facteur exponentiel dans le nombre de noeuds(formules propositionnelles) à explorer.

Venons en maintenant à l'algorithme naïf proprement dit. Il est décrit sur la figure 5.

Cet algorithme parcours le treillis $f_{\mathcal{S}}^{\downarrow}$ en partant du haut (de la formule $f_{\mathcal{S}}$), et en maintenant deux ensembles. Un ensemble **SOL** de solutions pseudo-minimales qui sera à la fin de l'exécution l'ensemble complet des solutions minimales de γ dans $\mathcal{S}$. Et un ensemble **UKN** des formules du treillis non encore examinées.

Nous pouvons représenter les ensembles **SOL** et **UKN** à l'aide de BDD, où par leurs éléments extrémaux puisque **SOL** est clos par le haut et **UKN** est l'intersection d'un clos par le haut et d'un clos par le bas. Mais dans les deux cas, il nous a semblé qu'une implémentation était difficilement envisageable.

Les opérations et outils de base dont nous avons besoins pour mettre en oeuvre cet algorithme sont

- le calcul de $min(E)$ où E est un ensemble de formules propositionnelles. La complexité du calcul de min, dans le cas où les formules propositionnelles sont représentées sous forme de BDD, est polynomiale : nous comparons les formules de E deux à deux en éliminant f dès que $\exists g, g \Rightarrow f$.
- Créer à partir d'une formule propositionnelle f, les formules propositionnelles g telles que $[\![g]\!] \hookrightarrow [\![f]\!]$
- Une représentation compacte des sous ensembles clos par le haut, clos par le bas, et de leur intersections (pour **UKN**)
- pouvoir convertir entre elles plusieurs représentations d'une même formule propositionnelles pour pouvoir faire l'interfaçage avec les algorithme de model-checking

Définition 4.3 *Soit A un ensemble de formules propositionnelles sur $\mathcal{P}$. Nous définissons l'ensemble des éléments minimaux de A*

$$min(A) = \{f \in A | \forall f' \in A, f' \subset f \Rightarrow f' = f\}$$

$f_{\mathcal{S}}^{\downarrow}$: ensemble des formules propositionnelles à examiner
SOL : $min\{\phi, \mathcal{S} \models \gamma(\phi)\}$ (à la terminaison de l'algorithme)
UKN : ensemble des formules non examinées

Initialisation :
UKN $:= f_{\mathcal{S}}^{\downarrow} \backslash \{f_{\mathcal{S}}, \perp\}$ On suppose que $\mathcal{S} \models \gamma(f_{\mathcal{S}})$ et $\mathcal{S} \not\models \gamma(\perp)$
SOL $:= \{f_{\mathcal{S}}\}$

On appelle `Algo_exhaustif`$(f_{\mathcal{S}})$:

```
Algo_exhaustif(g)
{
    Pour toute formule f telle que f ↪ g faire
    Si f ∈ UKN alors
    {
        Si S ⊨ γ(f) alors
        {
            SOL := min(SOL ∪ {f})
            UKN := UKN\f↑
            Algo_exhaustif(f)
        }
        Sinon
        {
            UKN := UKN\f↓
        }
    }
}
```

On récupère **SOL** , l'ensemble des solutions minimales.

FIG. 5 – L'algorithme générique

4.3 Un algorithme générique partiel

Nous répondons dans la fin de cette section à deux questions :
Une solution donnée est-elle minimale ?
γ et $\mathcal{S}$ connues, peut-on dire, de manière efficace, si γ admet une unique solution minimale dans $\mathcal{S}$?

Nous donnons pour chacune de ces questions, un algorithme polynômial (avec l'oracle du model-checking) qui résout le problème.

Proposition 4.2 *Décider la minimalité d'une solution est un problème polynômial.*

Preuve : *Soit $\mathcal{S} = \langle \mathcal{Q}, Q_0, \mathcal{P}, \rightarrow, l \rangle$ une structure de Kripke, γ une requête et f une solution de γ dans $\mathcal{S}$. Nous voulons savoir si f est une solution minimale ou pas.*

Soit $Q_f = \{q \in \mathcal{Q}, q \models f\} = \{q \in \mathcal{Q}, l(q) \in [\![f]\!]\}$ On a : $\{l(q), q \in Q_f\} \subset [\![f]\!]$
Deux cas se présentent :

1. *Si $|Q_f| < |f|_{sat}$ alors f n'est pas solution minimale. C'est une conséquence du lemme 4.1.*

2. *Si $|Q_f| \geqslant |f|_{sat}$. Pour chaque valuation v de $[\![f]\!]$ nous testons si $\mathcal{S} \models \gamma(f_v)$. Si le test échoue pour toutes les valuations de f alors f est bien une solution minimale. Si l'une des valuations peut être supprimée, alors f n'est pas solution minimale.* ■

Notons que cet algorithme (que nous appellerons `Algo_minimale()` par la suite) permet également de construire l'ensemble des valuations qui peuvent être enlevées (au moins une par une) de l'ensemble $[\![f]\!]$ pour obtenir une solution plus petite. Cette particularité nous sera utile dans l'algorithme présenté ci-après, qui permet de savoir en temps polynomial si une requête admet une unique solution dans une structure de Kripke donnée.

Le calcul effectif de $|f|_{sat}$ peut se faire en temps $O((f))$ linéaire en fonction de la taille qu'occupe f en mémoire si f est représentée sous forme de BDD.

Notons également que la complexité de ce dernier problème (présenté ci-dessous) n'est pas en contradiction avec le théorème 3.1 et les propositions 3.1 et 3.2 qui montre que décider la validité d'une requête est plus que polynômial. En effet, nous nous posons ici la question dans le cadre d'un seul modèle. L'unicité de la solution minimale est éventuellement vraie seulement pour un modèle donné et non pour tous les modèles.

Soient une structure de Kripke $\mathcal{S}$ et une requête γ. Nous supposons que $\mathcal{S} \models \gamma(f_{\mathcal{S}})$ (dans le cas contraire, γ n'admettrait aucune solution dans $\mathcal{S}$). Nous voulons savoir si γ admet une unique solution minimale ou plus. Dans le cas d'unicité, l'algorithme présenté ici nous donne concrètement cette solution minimale unique. Il est décrit sur la figure 6.

Initialisation :
On suppose que $\mathcal{S} \models \gamma(f_{\mathcal{S}})$ et $\mathcal{S} \not\models \gamma(\bot)$
L : ensemble de valuations

`Algo_unique()`
{
1 **L** := `Algo_minimale(`$f_{\mathcal{S}}$`)`
Si **L** $= \emptyset$ alors
2 return(La solution $f_{\mathcal{S}}$ est minimale et unique)
3 $g := [\![f_{\mathcal{S}}]\!] \backslash \mathbf{L}$
Si $\mathcal{S} \models \gamma(g)$ alors
4 return(La solution g est minimale et unique)
5 return(Il y a au moins deux solutions minimales)
}

FIG. 6 – L'algorithme générique ; partiel, mais polynomial

Proposition 4.3 *Décider si $(\mathcal{S}, \gamma)$ admet une unique solution est un problème polynômial.*

Preuve : *Examinons l'algorithme présenté sur la figure 6. L'énumération ci-dessous reprend les numéros de ligne de la figure.*

1. *Nous commençons par calculer l'ensemble* **L** *des valuations de $f_{\mathcal{S}}$ qui, otées de l'ensemble $f_{\mathcal{S}}$ (une à la fois), donnent des solutions à γ dans $\mathcal{S}$.*
2. *Si* **L** *est vide, nous sommes assurés que $f_{\mathcal{S}}$ est l'unique solution minimale. En effet, supposons qu'il existe $f \in f_{\mathcal{S}}^{\downarrow}, f \neq f_{\mathcal{S}}$, telle que $\mathcal{S} \models \gamma(f)$. Nous avons $[\![f]\!] \subset [\![f_{\mathcal{S}}]\!]$ donc il existe un ensemble de valuations X tel que*

$[\![f]\!] \subset X \hookrightarrow [\![f_\mathcal{S}]\!]$. Or $\mathbf{L}$ est vide donc X n'est pas solution de γ dans $\mathcal{S}$. Et d'après le lemme 3.1, f n'est pas solution de γ dans $\mathcal{S}$; nous obtenons une contradiction.

3. *une opération booléenne sur des ensembles finis de valuations.*
4. *Montrons que si $\mathcal{S} \models \gamma(g)$ alors g est bien l'unique solution minimale de γ dans $\mathcal{S}$.*
   Supposons que g ne soit pas une solution minimale. Il existe une valuation $v \models g$ telle que $\mathcal{S} \models \gamma([\![g]\!]\backslash\{v\})$. De plus $v \notin \mathbf{L}$ car $g = [\![f_\mathcal{S}]\!]\backslash\mathbf{L}$.
   $$[\![g]\!]\backslash\{v\} = [\![f_\mathcal{S}]\!]\backslash\mathbf{L}\backslash\{v\} \subset [\![f_\mathcal{S}]\!]\backslash\{v\}$$
   Donc $\mathcal{S} \models \gamma([\![f_\mathcal{S}]\!]\backslash\{v\})$ et $v \in \mathbf{L}$ par construction de $\mathbf{L}$.
   Nous obtenons $v \in \mathbf{L}$ et $v \notin \mathbf{L}$, ce qui est bien contradictoire.
5. *$\mathcal{S} \not\models \gamma(g)$ signifie que nous ne pouvons pas enlever de $[\![f_\mathcal{S}]\!]$ toutes les valuations de $\mathbf{L}$ d'un coup. Or toutes les valuations de $\mathbf{L}$ peuvent être, au moins une à une, enlevée de $[\![f_\mathcal{S}]\!]$ pour obtenir une solution de γ dans $\mathcal{S}$. Nous en déduisons qu'il existe au moins deux solutions minimales distinctes.* ■

5 Conclusion

Après avoir brièvement présenté les logiques temporelles et le model-checking, nous avons montré l'importance que pouvait avoir la notion de query-checking. Nous avons exploré ce problème par deux voies différentes.

La section 3 examine le cas des requêtes valides, qui n'ont qu'une seule solution minimale quel que soit le modèle que l'on analyse. Nous avons vu que nous pouvions, dans une certaines mesure, étendre les travaux de [Cha00] sur les requêtes CTL, aux logiques LTL, CTL* et FCTL. De même que pour CTL_v, nous y donnons un fragment de CTL*, CTL_v^* dont les requêtes sont valides. Par contre, nous ne donnons pas d'algorithme, par exemple pour résoudre les requêtes LTL, plus efficace que celui donné figure 5. Chan s'inspire du model-checking de CTL pour écrire l'algorithme de query-checking de CTL_v et il nous semble probable que cette méthode peut être également appliquée à d'autres logiques temporelle.

Dans la section 4, nous commençons par présenter certains aspects intéressants de la structure de l'espace des solutions d'une requête dans un modèle donné. Nous utilisons ensuite cette structure pour produire un algorithme de calcul exhaustif des solutions minimales d'une instance du problème du query-checking. Nous pensons que l'implémentation de cet algorithme est de peu d'intérêt : d'une part car elle parait techniquement compliquée, et d'une complexité trop grande (nous avons vu que le nombre de solutions minimales pouvait être exponentiel) ; d'autre part, comme nous l'indiquons en page 8, d'un point de vue pratique, il semble plus difficile de tirer parti des informations que nous apporterait la résolution d'une instance du problème du query-checking, lorsque celle-ci nous fournit un trop grand nombre de solutions minimales. Nous présentons alors deux algorithmes dont la complexité est de manière surprenante polynômiale. Le premier permet de s'assurer de la minimalité d'une solution f donnée. Sa complexité polynômiale est conséquence du lemme 4.1 qui restreint le nombre de sous formules g ($g \Rightarrow f$) à tester. Le second algorithme utilise le précédent pour determiner si une instance du problème du query-checking admet

« une seule » solution minimale et, le cas échéant, donne cette solution. Nous pensons que c'est un algorithme très intéréssant dans la mesure où il supprime la nécéssité de définir, comme dans la section 3, des fragments syntaxiques de logiques temporelles dont les requêtes sont valides. Il généralise en fait la notion de requête valide, en la portant localement à tous les modèles individuellement. Cet algorithme permet donc de résoudre en temps polynômial (n'oublions pas l'oracle, cependant !) toute instance du problème où la requête admet *dans le modèle considéré* une unique solution minimale, indépendament de la logique utilisée.

A Annexe, le query-checking de CTL_v

Nous présentons dans cette annexe l'algorithme de query-checking de CTL_v et donnons la preuve de sa correction (car elle n'a pas été publiée dans [Cha00]). Cet algorithme est conçu pour le fragment de requêtes CTL valides défini syntaxiquement dans [Cha00]. Nous ne donnons explicitement ce fragment. Il pourra être vu dans la description de l'algorithme, ci-dessous.

Définition A.1 *Soit $\mathcal{E}$ un ensemble d'états de contrôle d'une structure de Kripke $\mathcal{S}$. Nous notons $\chi(\mathcal{E})$ la formule propositionnelle*

$$\chi(\mathcal{E}) = \bigvee_{q \in \mathcal{E}} v_q$$

(voir au début de la section 4.1 pour la définition de v_q)

Définition A.2 *Soit $\mathcal{E}$ un ensemble d'états de contrôle d'une structure de Kripke $\mathcal{S}$. Nous notons $post(\mathcal{E})$ l'ensemble des successeurs des états de $\mathcal{E}$ par la relation $\rightarrow$ de $\mathcal{S}$*

$$post(\mathcal{E}) = \{q' \in \mathcal{Q} | \exists q \in \mathcal{E}, q \rightarrow q'\}$$

Définition A.3 *Nous notons respectivement $\mu Z.f(Z)$ et $\nu Z.f(Z)$ les opérateurs de plus petit et plus grand point fixe.*

Définition A.4 *Soit ϕ une formule CTL. Nous définissons l'ensemble*

$$R_\phi = \mu Z.((S \cup post(Z)) \cap Q_\phi)$$

(pour la définition de Q_ϕ voir la proposition) 4.2.

Définition A.5 *Nous qualifierons d'***exacte** *une solution minimale et unique.*

Remarque : dans cette annexe, les structures de Kripke peuvent avoir plusieurs états initiaux : Q_0 devient donc un ensemble d'états de contrôle.

La procédure *Solve* suivante calcule la solution $Solve(\gamma, Q_0)$ de la requête CTL_v γ dans la structure $\mathcal{S}$, par induction sur la structure de γ, lorsqu'on l'appelle avec les arguments γ et Q_0. ϕ représente une formule CTL.

$$\begin{array}{rcl}
Solve(?,\mathcal{E}) & = & \chi(\mathcal{E}) \\
Solve(\neg ?,\mathcal{E}) & = & \neg\chi(\mathcal{E}) \\
Solve(\phi \vee \gamma,\mathcal{E}) & = & Solve(\gamma,\mathcal{E}\backslash Q_\phi) \\
Solve(\mathbf{AX}\gamma,\mathcal{E}) & = & Solve(\gamma,post(\mathcal{E})) \\
Solve(\mathbf{A}(\gamma \overset{\circ}{\mathbf{W}} \phi),\mathcal{E}) & = & Solve(\gamma,\mathcal{E}\cup R_{\neg\phi}\cup post(R_{\neg\phi})) \\
Solve(\mathbf{A}(\phi \overline{\mathbf{W}} \gamma),\mathcal{E}) & = & Solve(\gamma,(\mathcal{E}\cup post(R_{\neg\phi}))\backslash Q_\phi) \\
Solve(\mathbf{A}(\phi \mathbf{W} \gamma),\mathcal{E}) & = & Solve(\gamma,B) \text{ où } R = R_{\phi\wedge\neg\gamma(\bot)} \\
 & & \qquad \text{et } B = (\mathcal{E}\cup post(R))\backslash(Q_\phi \cup Q_{\gamma(\bot)}) \\
Solve(\mathbf{A}(\phi \mathbf{U} \gamma),\mathcal{E}) & = & Solve(\gamma,B\cup C) \text{ où } C = \nu Z.(R\cap post(Z))
\end{array}$$

Preuve : Montrons par induction sur la structure de γ que $Solve(\gamma, Q_0)$ est la solution exacte de γ dans $\mathcal{S}$.

Notons que les règles d'induction ci-dessus (sauf les 2 premières) sont de la forme $Solve(F(\gamma),\mathcal{E}) = Solve(\gamma, G(\mathcal{E}, F))$. Pour chacune des règles 3 à 8 dans cette preuve, nous définissons la strutucre de Kripke $\mathcal{S}'$ comme étant la structure $\mathcal{S}$ dans laquelle nous remplaçons l'ensemble d'états de contrôle $\mathcal{E}$ par $G(\mathcal{E}, F)$. Pour chacune des ces règles, la structure de la preuve est la suivante : nous montrons l'équivalence entre « $Solve(\gamma, Q_0)$ est solution exacte de γ dans $\mathcal{S}$ » et « $Solve(\gamma, Q_0)$ est solution exacte de γ' dans $\mathcal{S}'$ »

Notons également que nous ne nous préoccupons pas de montrer que la solution est unique. Ceci nous est en effet assuré par la *validité* des requêtes sur lesquelles nous travaillons, bien que la preuve de la validité des requêtes CTL_v ne soit pas présentée ici (elle se fait facilement par récurrence sur la structure de γ).

1. $\boxed{\gamma = ?}$: $Solve(?, Q_0) = \chi(Q_0)$, pas de problème.
2. $\boxed{\gamma = \neg ?}$: $Solve(\neg ?, Q_0) = \neg\chi(Q_0)$, pas de problème.
3. $\boxed{\gamma = \phi \vee \gamma' \text{ ou } \gamma = \gamma' \vee \phi.\ Solve(\gamma, Q_0) = Solve(\gamma', Q_0\backslash Q_\phi)}$

   Montrons que $Solve(\gamma, Q_0)$ est une solution. Nous savons que $Solve(\gamma, Q_0)$ est solution exacte dans $\mathcal{S}'$, la structure de Kripke S dans laquelle nous avons remplacé Q_0 par $Q_0\backslash Q_\phi$ (c'est l'hypothèse d'induction). Soit $q \in Q_0$. Si $q \notin Q_\phi$ alors $q \models_{\mathcal{S}'} \gamma'(Solve(\gamma, Q_0))$ donc $q \models_{\mathcal{S}} \gamma(Solve(\gamma, Q_0))$. Si $q \in Q_0$ alors $q \models \mathcal{S}\phi$ donc $q \models_{\mathcal{S}} \gamma(Solve(\gamma, Q_0))$. $Solve(\gamma, Q_0)$ est donc bien solution.

   Montrons que $Solve(\gamma, Q_0)$ est une solution minimale. Soit p une solution.

   $$\forall q \in Q_0, q \models_{\mathcal{S}} \gamma(Solve(\gamma, Q_0)) \text{ et } q \models_{\mathcal{S}} \gamma(p)$$

   $$\forall q \in Q_0\backslash\phi, q \models_{\mathcal{S}'} \gamma'(Solve(\gamma, Q_0)) \text{ et } q \models_{\mathcal{S}'} \gamma'(p)$$

   Mais $Solve(\gamma, Q_0)$ est exacte dans $\mathcal{S}'$ donc $Solve(\gamma, Q_0) \Rightarrow p$. $Solve(\gamma, Q_0)$ est bien minimale.

   (Remarque : pour simplifier l'écriture, on suppose là encore que γ est positive)
4. $\boxed{\gamma = \mathbf{AX}\gamma'.\ Solve(\gamma, Q_0) = Solve(\gamma', post(Q_0))}$

   Montrons que $Solve(\gamma, Q_0)$ est une solution. Soit $\mathcal{S}'$ la structure de Kripke $\mathcal{S}$ dans laquelle on a remplacé l'ensemble Q_0 des états initiaux par $post_{\mathcal{S}}(Q_0)$. On a $\mathcal{S}' \models \gamma'(Solve(\gamma, Q_0))$. Soit $q \in Q_0$ et $q' \in post(\{q\})$. On a $q' \models$

$\gamma'(Solve(\gamma, Q_0))$ donc $q \models \mathbf{AX}\gamma'(Solve(\gamma, Q_0))$: $q \models \gamma(Solve(\gamma, Q_0))$. Donc $Solve(\gamma, Q_0)$ est bien solution.

Montrons que $Solve(\gamma, Q_0)$ est une solution minimale. Soit p une solution.

$$\forall q \in Q_0, q \models \gamma(p)$$

$$\forall q' \in post(Q_0), q' \models \gamma'(p)$$

Or $Solve(\gamma, Q_0)$ est exacte dans $\mathcal{S}'$ donc $Solve(\gamma, Q_0) \Rightarrow p$. $Solve(\gamma, Q_0)$ est bien minimale.

5. $\boxed{\gamma = \mathbf{A}(\gamma' \ \mathring{\mathbf{W}} \ \phi).\ Solve(\gamma, Q_0) = Solve(\gamma', Q_0 \cup R_{\neg\phi} \cup post(R_{\neg\phi}))}$

   Montrons que $Solve(\gamma, Q_0)$ est une solution. Soit $q_0 \in Q_0$ et $q_0 \to q_1 \to q_2 \to ...$ une exécution de $\mathcal{S}$.
   - si $q_0 \models \phi, q_0 \models \gamma(Solve(\gamma, Q_0))$. OK.
   - si $\forall i, q_i \not\models \phi$ alors $\forall i, q_i \in R_{\neg\phi}$ donc $\forall i, q_i \models \gamma'(Solve(\gamma, Q_0))$ donc $q_0 \models \gamma(Solve(\gamma, Q_0))$.
   - sinon, $\exists i > 0, q_i \models \phi$ (le premier i) . $q_i \in post(R_{\neg\phi})$ donc $q_i \models \phi \wedge \gamma'(Solve(\gamma, Q_0))$ et $\forall j < i, q_j \models \gamma'(Solve(\gamma, Q_0))$ (car $q_j \in R_{\neg\phi}$). Donc $q_0 \models \gamma(Solve(\gamma, Q_0))$.

   $Solve(\gamma, Q_0)$ est bien solution.

   Montrons que $Solve(\gamma, Q_0)$ est une solution minimale. Soit p une solution. Soit $q' \in Q_0 \cup R_{\neg\phi} \cup post(R_{\neg\phi})$.
   - si $q' \in Q_0$ alors $q' \models \gamma(p)$ et $q' \models \gamma'(p)$ (grâce au $\mathring{\mathbf{W}}$).
   - si $q' \in R_{\neg\phi}$, $\exists q \in Q_0$ tel que $q \to^* q'$ (en restant dans $R_{\neg\phi}$). $q \models \neg\phi$ et $q \models \gamma(p)$ donc $q' \models \gamma(p)$ aussi. Or, comme $q' \not\models \phi$, on a $q' \models \gamma'(p)$.
   - si $q' \in post(R_{\neg\phi})$ de même : $q' \models \gamma'(p)$

     Ainsi, p est solution de γ' dans $\mathcal{S}'$: $Solve(\gamma, Q_0) \Rightarrow p$. $Solve(\gamma, Q_0)$ est bien minimale.

6. $\boxed{\gamma = \mathbf{A}(\phi \ \overline{\mathbf{W}} \ \gamma').\ Solve(\gamma, Q_0) = Solve(\gamma', (Q_0 \cup post(R_\phi)) \backslash Q_\phi)}$

   On sait que $Solve(\gamma, Q_0)$ est solution exacte de γ' dans $\mathcal{S}'$. Montrons que $Solve(\gamma, Q_0)$ est solution exacte dans $\mathcal{S}$. Posons $Q_0' = (Q_0 \cup post(R_\phi)) \backslash Q_\phi$

   Soit $q \in Q_0$.
   - Si $q \not\models \phi$ alors $q \in Q_0'$: $q \models \gamma'(Solve(\gamma, Q_0)) \wedge \neg\phi$ donc $q \models \gamma(Solve(\gamma, Q_0))$.
   - Si $q \models \phi$. Soit $q = q_0 \to q_1 \to q_2 \to ...$ un chemin dans $\mathcal{S}$.
     - Soit $\forall i, q_i \models \phi$ alors $q \models \gamma(Solve(\gamma, Q_0))$.
     - Soit $\exists i > 0$ tel que $q_i \not\models \phi$:

$$\left.\begin{array}{r}\forall j = 0..i-1, q_j \models \phi \\ q_i \models \neg\phi \wedge \gamma'(Solve(\gamma, Q_0)) \text{ (car } q_i \in Q_0')\end{array}\right\} \text{donc } q \models \gamma(Solve(\gamma, Q_0))$$

   Ainsi $Solve(\gamma, Q_0)$ est bien solution.

   Montrons que $Solve(\gamma, Q_0)$ est une solution minimale. Soit p une solution. Montrons que $\forall q' \in Q_0', q' \models \gamma'(p)$. Soit $q' \in Q_0'$. $q' \models \neg\phi$.
   - Si $q' \in Q_0$ alors $q' \models \gamma(p)$ donc $q' \models \gamma(p)$ (car $q' \models \neg\phi$)
   - Si $q' \notin Q_0$ alors $q' \in post(R_\phi)$ donc $q' \models \gamma(p)$ donc $q' \models \gamma'(p)$.

   p est donc une solution de γ' dans $\mathcal{S}'$: $Solve(\gamma, Q_0) \Rightarrow p$. $Solve(\gamma, Q_0)$ est bien solution exacte.

7. $\boxed{\gamma = \mathbf{A}(\phi \ \mathbf{W} \ \gamma').\ Solve(\gamma, Q_0) = Solve(\gamma', (Q_0 \cup post(R_{\phi \wedge \neg\gamma'(\perp)})) \backslash (Q_\phi \cup Q_{\gamma'(\perp)}))}$

   On sait que $Solve(\gamma, Q_0)$ est solution exacte de γ' dans $\mathcal{S}'$. Montrons que $Solve(\gamma, Q_0)$ est solution exacte de γ dans $\mathcal{S}$. Posons $Q_0' = (Q_0 \cup$

$post(R_{\phi \wedge \neg\gamma'(\bot)}))\backslash(Q_\phi \cup Q_{\gamma'(\bot)})$
Soit $q \in Q_0$.
- Si $q \models \gamma'(\bot)$ alors $q \models \gamma'(Solve(\gamma, Q_0))$ et donc $q \models \gamma(Solve(\gamma, Q_0))$.
- Si $q \not\models \gamma'(\bot)$,
  - Si $q \not\models \phi$ alors $q \in Q'_0$ donc $q \models \gamma'(Solve(\gamma, Q_0))$ donc $q \models \gamma(Solve(\gamma, Q_0))$.
  - Si $q \models \phi$ alors $q \in R$. Soit $q = q_0 \to q_1 \to q_2 \to ...$ un chemin dans $\mathcal{S}$.
    - Si $\forall i > 0, q_i \in R$ alors $\forall i \geqslant 0, q_i \models \phi$ et donc $q \models \phi\ \mathbf{W}\ \gamma'(Solve(\gamma, Q_0))$.
    - Sinon, $\exists i > 0, q_i \in post(R)\backslash R$.
      - Si $q_i \models \phi$ alors $q_i \models \gamma'(\bot)$ donc $q_i \models \gamma'(Solve(\gamma, Q_0))$ et $q \models \phi\ \mathbf{W}\ \gamma'(Solve(\gamma, Q_0))$.
      - Si $q_i \not\models \phi$ alors, si $q_i \models \gamma'(\bot)$, OK (cf ci-dessus), sinon $q_i \in B$ donc $q_i \models \gamma'(Solve(\gamma, Q_0))$ donc $q \models \phi\ \mathbf{W}\ \gamma'(Solve(\gamma, Q_0))$.

Nous avons donc $q \models \gamma(Solve(\gamma, Q_0))$.

Montrons que $Solve(\gamma, Q_0)$ est une solution minimale. Soit p une solution. Soit $q' \in Q'_0$.
- Si $q' \in Q_0$ alors $q' \models \mathbf{A}(\phi\ \mathbf{W}\ \gamma'(p))$ et $q' \not\models \phi$ donc $q' \models \gamma'(p)$.
- Sinon, $q' \in post(R_{\phi \wedge \neg\gamma'(\bot)})\backslash(Q_\phi \cup Q_{\gamma'(\bot)})$. Il existe $= q_0 \to q_1 \to q_2 \to ... \to q_n \to q'$ tel que
  - $q_0 \in Q_0$, $q_0 \models \mathbf{A}\phi\ \mathbf{W}\ \gamma'(Solve(\gamma, Q_0))$
  - $\forall i = 0..n, q_i \in R_{\phi \wedge \gamma'(\bot)}$ donc $q_i \models \phi \wedge \cancel{\gamma}'(\bot)$
  - $(q' \models \neg\phi \wedge \neg\gamma'(\bot)$

  $q' \models \neg\phi$ or $q_0 \models \mathbf{A}(\phi\ \mathbf{W}\ \gamma'(p))$ donc $q' \models \gamma'(p)$

p est donc solution de γ' dans $\mathcal{S}'$. $Solve(\gamma, Q_0)$ est solution exacte de γ' dans $\mathcal{S}'$ donc $Solve(\gamma, Q_0) \Rightarrow p$. $Solve(\gamma, Q_0)$ est donc solution exacte de γ dans $\mathcal{S}$.

8. $\boxed{\gamma = \mathbf{A}(\phi\ \mathbf{U}\ \gamma').\ Solve(\gamma, Q_0) = Solve(\gamma', B \cup C) \text{ où } C = \nu Z.(R \cap post(Z))}$

À B qui assure $\mathbf{A}(\phi\ \mathbf{W}\ \gamma')$, on ajoute l'ensemble (la contrainte) C des états pouvant faire partie d'un chemin infini inclu dans $R_{\phi \wedge \neg\gamma(\bot)}$: ceux-ci doivent « vérifier » γ' pour que $\mathbf{A}(\phi\ \mathbf{U}\ \gamma')$ tienne. ■

Références

[BBL+99] Béatrice Bérard, Michel Bidoit, François Laroussinie, Antoine Petit, and Philippe Schnoebelen. *Vérification de logiciels, techniques et outils du model-checking*. Vuibert, 1999.

[BG01] Glenn Bruns and Patrice Godefroid. Temporal logic query checking. In *Proc. 16th IEEE Symp. Logic if Computer Science (LICS'01)*, pages 409–417, 2001.

[CES86] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2) :244–263, April 1986.

[CGH94] Edmund M. Clarke, Orna Grumberg, and K. Hamaguchi. Another look at LTL model checking. *Proceedings of Computer Aided Verification (CAV'94), LNCS 818*, pages 415–427, 1994.

[Cha00] William Chan. Temporal-logic queries. In *Proceedings of Computer Aided Verification (CAV'00), LNCS 1855*, pages 450–463, 2000.

[EL87] E. Allen Emerson and Chin-Laung Lei. Modalities for model checking : Branching time logic strikes back. *Science of Computer Programming*, 8(3) :275–306, 1987.

[Eme90] E. Allen Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, vol. B*, chapter 16, pages 995–1072. Elsevier Science, 1990.

[Lam80] Leslie Lamport. "Sometimes" is sometimes "Not Never". In *Proceedings of the Seventh ACM Symposium on Principles of Programming Languages (POPL'80)*, pages 174–185, January 1980.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. Foundation of Computer Science (FOCS'77)*, pages 46–57, Oct.-Nov. 1977.

[Pnu81] Amir Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13(1) :45–60, 1981.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. Int. Symp. on Programming, LNCS 137*, pages 337–351, April 1982.

[SC85] A.Prasad Sistla and Edmund M. Clarke. The complexity of linear temporal logic. *Journal ACM*, 32(3) :733–749, July 1985.