Automates temporisés et calcul de réécriture

Emmanuel Beffara

Stage MIM1

Résumé

On s'intéresse ici à l'automatisation par la logique de réécriture de preuves de propriétés d'atteignabilité dans une forme d'automates temporisés. On commence pour cela par introduire un formalisme pour représenter des ensembles d'états du système. On construit ensuite à partir de ce formalisme un système de réécriture que l'on montre équivalent à l'automate considéré, puis on définit des stratégies destinées à rendre compte du fonctionnement de l'automate. On implémente enfin les méthodes de preuve établies à l'aide du logiciel ELAN pour montrer comment les preuves sont effectivement menées.

1 Introduction

Les systèmes hybrides sont des systèmes qui peuvent être représentés par des automates finis agissant dans un environnement analogique, c'est-à-dire qu'ils sont munis de variables réelles qui évoluent selon des lois particulières en fonction de l'état et des transitions de l'automate et que leurs états et transitions sont soumis à des contraintes portant sur ces variables. De tels systèmes, en particulier étudiés dans [ACH+95], permettent de modéliser des processus agissant en environnement réel de façon formelle.

1.1 Automates temporisés

Le modèle des automates temporisés, introduit par Rajeev Alur et David L. Dill dans [AD94], est un cas particulier de système hybride dans lequel on impose à toutes les variables de croître à la même vitesse, et où l'on n'autorise que certaines formes de conditions. Leur intérêt est d'une part d'aboutir à un formalisme plus simple que celui des systèmes hybrides tout en conservant un pouvoir expressif suffisant pour modéliser un grand nombre de systèmes dépendants du temps. D'autre part, les restrictions imposées par ce modèle rendent décidables des problèmes indécidables dans le cas général, en particulier celui de l'atteignabilité auquel nous nous intéressons ici.

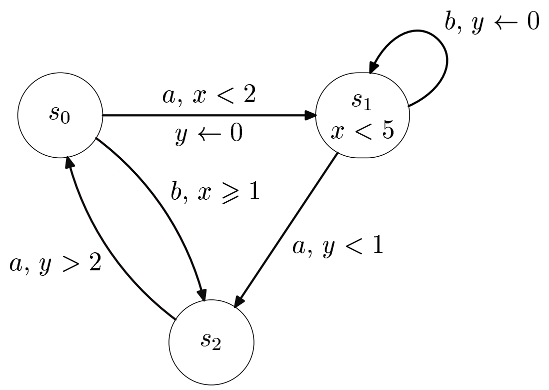


Fig. 1 – un exemple d'automate temporisé

Considérons l'exemple de la figure 1. On a un automate à deux horloges x et y. L'état s_1 est soumis à la contrainte $x < 5$, et les deux autres états ne sont soumis à aucune contrainte. La transition de s_2 à s_0, étiquetée par a, est soumise à la contrainte $y > 2$, et la transition de s_1 vers lui-même n'est soumise à aucune condition mais a pour effet de remettre à zéro l'horloge y.

Une marche d'un tel automate consiste donc en une suite de transitions d'un état à un autre, où deux transitions consécutives sont espacées d'un intervalle de temps durant lequel les horloges croissent toutes à la même vitesse. On fournira une définition formelle de ces automates dans la partie 2.1 avant de se pencher sur l'automatisation de preuves.

Produits synchronisés

Il est possible grâce à ce formalisme de modéliser facilement des systèmes de processus concurrents. On considère pour cela un automate par composant du système et on synchronise les composants en utilisant l'étiquetage des transitions. On aboutit ainsi à un modèle intéressant pour des systèmes composites où des entités indépendantes interagissent par échanges de messages. On trouvera des considérations plus poussées sur les systèmes concurrents de cette forme dans [Dil89] en particulier.

Un exemple classique est celui du passage d'un train sur un passage à niveau, modélisé par les automates de la figure 2. Il est composé de trois sous-automates, respectivement pour le train, la barrière et le système de contrôle. Les étiquettes des transitions sont donc vues comme des messages circulant entre les différents composants du système.

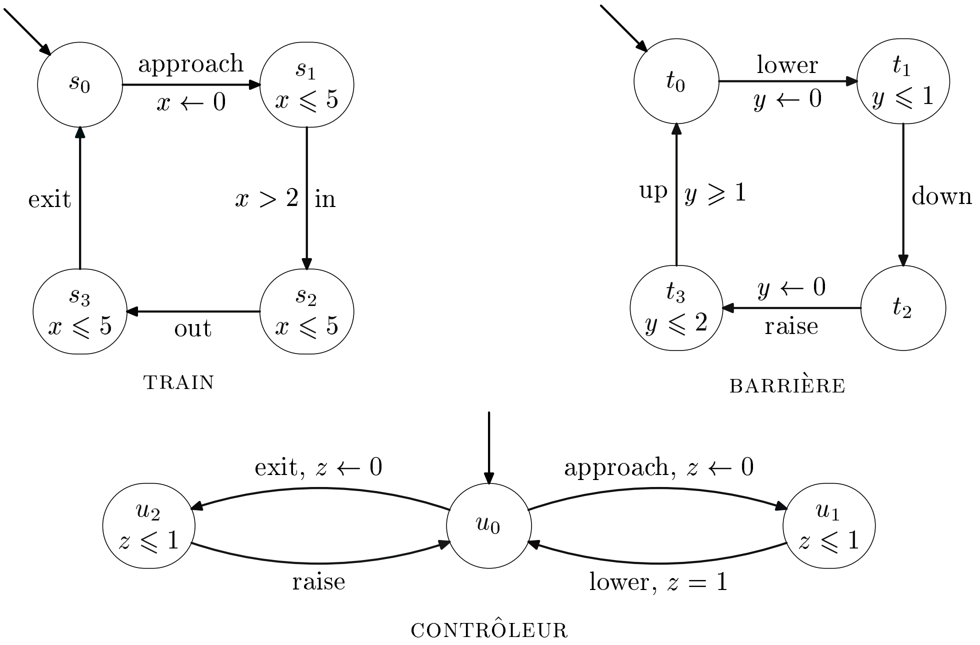


FIG. 2 – le système train / barrière / contrôleur

Un exemple de marche de ce système serait par exemple le suivant, en commençant avec les états s_0, t_0 et u_0 :

1. le train envoie le message « approach » signalant sont arrivée, et il change d'état en même temps que le contrôleur ;
2. moins d'une minute plus tard, le contrôleur revient dans l'état u_0 en émettant le signal « lower » reçu par la barrière qui commence à s'abaisser (état t_1) ;
3. moins d'une minute plus tard, la barrière est baissée et émet le message « down » ;
4. plus de deux minutes après le premier signal, le train arrive sur le passage à niveaux en émettant le message « in » ;

5. et ainsi de suite...

Pour formaliser cette idée, on introduira dans la partie 5.1 la notion de produit synchronisé d'automates temporisés.

Atteignabilité

On se penche ici plus particulièrement sur la résolution de problèmes d'atteignabilité, c'est-à-dire que l'on cherche à savoir s'il est possible, à partir d'un état initial, d'atteindre certains états. On résout souvent de tels problèmes pour garantir qu'un ensemble d'états considérés comme indésirables n'est pas atteignable, afin de prouver la sécurité d'un système. Dans le système de la figure 2, on veut par exemple prouver le fait que le train ne puisse jamais se trouver sur le passage à niveau alors que les barrières sont ouvertes.

1.2 Calcul de réécriture

On va donc modéliser de tels objets par le calcul de réécriture. La notion de réécriture est une notion très générale et donc fondamentale. On peut définir un système de réécriture comme la donnée d'un vocabulaire et d'un ensemble de réductions, c'est-à-dire d'applications de l'ensemble des termes formés sur le vocabulaire dans lui-même.

L'exemple canonique de système de réécriture est le λ-calcul, dans lequel le vocabulaire est composé des symboles de variables $x, y, \ldots$ d'arité nulle, des symboles d'abstraction $\lambda x\, \_, \lambda y\, \_, \ldots$ d'arité 1 et du symbole d'application $\_\,\_$ d'arité 2. Le système est alors muni de la règle de réécriture

$$(\lambda x\, P)\, Q \longrightarrow Q[x := P]$$

où $Q[x := P]$ représente le terme Q dans lequel on substitue le terme P à toutes les occurrences libres de x. On peut ensuite poursuivre le raisonnement et définir la substitution $Q[x := P]$ comme un symbole supplémentaire de substitution $\_[\_ := \_]$ d'arité 3 dont le comportement est défini par les règles

$$\begin{aligned}
x[x := P] &\longrightarrow P \\
y[x := P] &\longrightarrow y \qquad \text{si } x \not\equiv y \\
(\lambda x\, Q)[x := P] &\longrightarrow \lambda x\, Q \\
(\lambda y\, Q)[x := P] &\longrightarrow \lambda y\, (Q[x := P]) \qquad \text{si } x \not\equiv y \\
(Q\, R)[x := P] &\longrightarrow (Q[x := P])\, (R[x := P])
\end{aligned}$$

On peut ensuite définir des stratégies de réduction particulières au lieu de se limiter à la normalisation de termes.

1.3 Travaux présentés

On va ici faire le lien entre le modèle des automates temporisés et le calcul de réécriture. On va pour cela introduire un formalisme pour représenter les états d'un automate et établir une méthode pour transcrire le fonctionnement de l'automate à l'aide de règles de réécriture sur des termes représentant les états. On verra également comment combiner des systèmes de réécriture obtenus par cette méthode afin de manipuler des produits synchronisés d'automates.

Après avoir établi la décidabilité des problèmes d'atteignabilité sur les automates temporisés, on montrera ensuite comment ils peuvent être effectivement résolus à l'aide de ces règles de réécriture. On terminera enfin par une implémentation des règles à l'aide du logiciel ELAN afin de montrer la résolution effective de problèmes d'atteignabilité.

2 Automates temporisés

Un automate temporisé est donc intuitivement un automate fini muni d'un nombre fini d'horloges à valeurs réelles qui croissent toutes à vitesse 1 dans lequel les transitions et les états peuvent être soumis à des contraintes formulées en fonction de ces horloges.

2.1 Définition

Un des éléments qui caractérisent les automates temporisés est le type de contraintes que l'on s'autorise à y utiliser. On appellera *états temporels* les valuations des horloges, c'est-à-dire les applications de l'ensemble fini K des horloges dans $\mathbf{R}$, que l'on peut aussi voir comme des points de l'espace $\mathbf{R}^{|K|}$. On pose alors la définition suivante :

définition 1 (*contrainte temporelle*)

On définit l'ensemble des formules de contrainte sur K comme le plus petit langage contenant la constante $\top$ et les formules $x \sharp c$ et $x - y \sharp c$ avec $x, y \in K$, $c \in \mathbf{Q}$ et $\sharp \in \{<, \leqslant, >, \geqslant\}$ et clos par l'opérateur binaire $\wedge$.

Étant donnée une formule de contrainte φ, on définit son interprétation $\hat{\varphi}$ comme la partie de l'ensemble des états temporels définie par induction par $\widehat{x < c} = \{v \mid v(x) < c\}$ et de même pour les autres formules atomiques, et en interprétant l'opération $\wedge$ comme l'intersection. La constante $\top$ est interprétée comme l'espace entier.

On notera $\mathrm{CC}(K)$ l'ensemble des contraintes temporelles sur K, c'est-à-dire l'ensemble des interprétations de formules de contraintes.

La raison pour laquelle on ne s'autorise à utiliser que ce type de contraintes apparaîtra au cours des preuves. Le principal intérêt est qu'elles permettent de considérer un ensemble fini de contraintes possibles, ce qui rend les problèmes décidables.

On va donc voir un automate temporisé comme un automate fini muni d'un ensemble fini d'horloges réelles, dont les états et transitions sont soumis à des contraintes de cette forme sur les horloges, et qui pourra lors des transitions remettre à zéro certaines de ces horloges. On formalise cette définition de la façon suivante, en reprenant l'idée de la définition d'origine que l'on trouvera dans [AD94, Alu98].

définition 2 (*automate temporisé*)

On définit un automate temporisé comme un quintuplet

$$\mathcal{A} = (\Sigma, Q, K, I, \Delta)$$

- *Σ est l'alphabet (fini) ;*
- *Q est l'ensemble (fini) des états discrets ;*
- *K est l'ensemble (fini) des horloges ;*
- *I est l'invariant, application de Q dans $\mathrm{CC}(K)$;*
- *Δ, partie de $\Sigma \times Q \times \mathrm{CC}(K) \times \mathcal{P}(K) \times Q$, est la table de transition de $\mathcal{A}$.*

On comprend l'invariant comme une application qui à chaque état discret associe l'ensemble des états temporels valides pour cet état. Dans l'exemple du train, l'invariant de l'état s_1 est l'ensemble des valuations des horloges qui satisfont la contrainte $x \leqslant 5$, donc l'ensemble $]-\infty, 5]$.

L'ensemble Σ est ici nommé alphabet par analogie avec les automates finis classiques. On ne s'intéressera cependant pas ici à des problèmes de reconnaissance de langages, même s'il serait possible de la faire. L'alphabet sera plutôt vu comme un ensemble d'étiquettes pour les transitions, ces étiquettes étant interprétées comme des événements ou comme des messages[1] circulant

[1] Le système UPPAAL, destiné précisément à automatiser la résolution de divers problèmes sur les automates temporisés, rend cette idée explicite en étiquetant « msg ! » la transition qui émet le message « msg » et « msg ? » la transition qui le reçoit.

entre plusieurs composants d'un système concurrent, comme c'est le cas dans notre exemple : « approach » est interprété comme un message émis par le train et reçu par le contrôleur.

2.2 Fonctionnement

Une fois cette définition formelle posée, il reste encore à formaliser le fonctionnement d'un tel objet. On définit celui-ci dans le même esprit que la définition originale, au formalisme près.

Étant donné un automate temporisé $\mathcal{A} = (\Sigma, Q, K, I, \Delta)$, on définit tout d'abord un état temporisé, ou simplement état, de $\mathcal{A}$ comme étant la donnée d'un état discret et d'un état temporel, c'est-à-dire un couple (q, v) où q est un élément de l'ensemble Q et où v est une valuation des horloges. On nommera *états discrets* les éléments de Q, ou simplement états s'il n'y a pas d'ambiguïté. Un état (q, v) sera qualifié de valide s'il respecte l'invariant de l'automate, c'est-à-dire si l'état temporel v est élément de la contrainte $I(q)$.

On distingue ensuite deux types de transitions à partir d'un état (q, v) :

les transitions continues qui ont lieu au sein d'un même état discret et qui rendent compte du passage du temps. Une telle transition est caractérisée par sa durée τ, et elle fait donc succéder à (q, v) tout état $(q, v + \tau)$ avec τ positif (où l'on note naturellement $(v + \tau)(k) = v(k) + \tau$) avec la condition que tous les états traversés soient valides, c'est-à-dire

$$\forall t \in [0, \tau], v + t \in I(q)$$

les transitions discrètes qui correspondent aux changements d'état discret. Ces transitions sont définies par la table de transition Δ. Pour un élément $\delta = (a, q, \chi, z, q')$ de Δ, on dit que la transition δ peut être empruntée à partir de (q, v) si v satisfait la contrainte χ. Le successeur de (q, v) par δ est alors l'état $(q', \pi_z(v))$ où π_z est l'opération consistant à mettre à zéro les horloges de z :

$$\pi_z(v)(k) = \begin{cases} v(k) & \text{si } k \notin z \\ 0 & \text{si } k \in z \end{cases}$$

Cette transition est dite *étiquetée* par a, ou associée à l'événement a.

On utilisera la notation

$$(q, v) \longrightarrow (q', v')$$

pour signifier qu'il existe une transition (de type indifférent) par laquelle l'état (q', v') succède à l'état (q, v). Une trajectoire de l'automate $\mathcal{A}$ est donc une suite de transitions de l'un de ces deux types appliquées à un état initial donné.

2.3 Atteignabilité

De la notation précédente, on déduit la notation

$$(q, v) \longrightarrow^* (q', v')$$

pour signifier soit que (q, v) et (q', v') sont identiques, soit qu'il existe une suite $(q_i, v_i)_{1 \leqslant i \leqslant n}$ d'états (éventuellement vide) telle que l'on ait

$$(q, v) \longrightarrow (q_1, v_1) \longrightarrow \cdots \longrightarrow (q_n, v_n) \longrightarrow (q', v')$$

c'est-à-dire que $\rightarrow^*$ est la clôture réflexive et transitive de $\rightarrow$. Ceci permet de définir les problèmes d'atteignabilité, auxquels on s'intéresse principalement ici :

définition 3 (*problème d'atteignabilité*)

Soit $\mathcal{A}$ un automate temporisé, et soient S_0 et S_f deux ensembles d'états temporisés de *l'automate $\mathcal{A}$. On appelle problème d'atteignabilité de S_f à partir de S_0 le problème consistant à décider de la validité de la formule*

$$\exists \theta_0 \in S_0,\ \exists \theta_f \in S_f,\ \theta_0 \longrightarrow^* \theta_f$$

c'est-à-dire que l'on cherche à savoir s'il existe une trajectoire de $\mathcal{A}$ allant de S_0 à S_f.

3 Définitions et résultats préliminaires

La première transformation du problème consiste à restreindre la nature des ensembles d'états considérés. On utilisera pour cela des ensembles d'états temporels de la même forme que les contraintes de l'automate, en utilisant les résultats qui suivent.

3.1 Contraintes linéaires

On commence par établir des résultats sur l'élimination de quantificateurs dans les contraintes linéaires, afin de montrer ensuite des propriétés de stabilité de l'ensemble des contraintes linéaires.

3.1.1 Résultats généraux

Nous allons ici établir quelques résultats intermédiaires sur les contraintes linéaires. Le plus important est la propriété de stabilité suivante, formulée de façon plus générale que ce que notre propos requiert.

théorème 1 (*élimination du quantificateur existentiel*)

Soit K un sous-corps de $\mathbf{R}$. Soient $(v_1, \ldots, v_m)$ une famille de vecteurs de K^{n+1} et $(c_1, \ldots, c_m)$ une famille de constantes de K. Alors il existe une famille $(w_1, \ldots, w_p)$ de vecteurs de K^n et une famille $(d_1, \ldots, d_p)$ de constantes de K telles que la formule

$$\varphi(x_1, \ldots, x_n) \equiv \exists y \bigwedge_{1 \leqslant i \leqslant m} \big(v_i \cdot (x_1, \ldots, x_n, y) \leqslant c_i\big)$$

soit équivalente à la formule

$$\varphi'(x_1, \ldots, x_n) \equiv \bigwedge_{1 \leqslant j \leqslant p} \big(w_j \cdot (x_1, \ldots, x_n) \leqslant d_j\big)$$

c'est-à-dire que tout vecteur $(x_1, \ldots, x_n)$ valide la formule φ si et seulement s'il valide φ'.

Nous allons énoncer une preuve constructive de ce théorème. Cette preuve est en fait une reformulation de la méthode de Fourier-Motzkin.

preuve : L'idée sera d'inférer, par combinaisons linéaires à partir des contraintes initiales $v_i \cdot (\vec{x}, y) \leqslant c_i$, toutes les contraintes possibles ne faisant pas intervenir y, puis de montrer que l'ensemble obtenu sera bien suffisant.

construction de la formule

On commence par partitionner l'ensemble $\{1, \ldots, n\}$ en trois parties I_0, I_+ et I_- (dont certaines sont éventuellement vides) de telle sorte que $\varphi(\vec{x})$ s'écrive

$$\exists y \big(\varphi_0(\vec{x}) \wedge \varphi_+(\vec{x}, y) \wedge \varphi_-(\vec{x}, y)\big)$$

avec

$$\begin{array}{rcl}\varphi_0(\vec{x}) & \equiv & \bigwedge_{i\in I_0}(v_i^0\cdot\vec{x}\leqslant c_i^0)\\ \varphi_+(\vec{x},y) & \equiv & \bigwedge_{i\in I_+}(v_i^+\cdot\vec{x}\leqslant c_i^+ + y)\\ \varphi_-(\vec{x},y) & \equiv & \bigwedge_{i\in I_-}(v_i^-\cdot\vec{x}\leqslant c_i^- - y)\end{array}$$

où les v_i^* sont des vecteurs de K^n, les c_i^* des constantes de K, et où $\vec{x}$ représente le vecteur $(x_1,\dots,x_n)$. Cette réécriture de φ revient simplement à appliquer des coefficients positifs sur les inégalités pour les normaliser selon y puis à réordonner les contraintes obtenues.

Comme $\varphi_0(\vec{x})$ ne dépend pas de y, il est alors évident que $\varphi(\vec{x})$ est équivalente à

$$\varphi_0(\vec{x})\wedge\exists y\big(\varphi_+(\vec{x},y)\wedge\varphi_-(\vec{x},y)\big)$$

On définit alors la formule $\tilde{\varphi}(\vec{x})$ suivante :

$$\tilde{\varphi}(\vec{x})\equiv\bigwedge_{(i,j)\in I_+\times I_-}\big((v_i^+ + v_j^-)\cdot\vec{x}\leqslant c_i^+ + c_j^-\big)$$

On notera bien que si l'un des ensembles d'indices I_+ et I_- est vide, alors cette formule est vide également, et on la définira naturellement équivalente à la valeur de vérité *vrai*. En dehors de ce cas, les clauses de $\tilde{\varphi}$ sont des sommes d'inégalités prises dans φ_+ et φ_-, donc il est clair que $\varphi_+(\vec{x},y)\wedge\varphi_-(\vec{x},y)$ implique $\tilde{\varphi}(\vec{x})$ pour tout $\vec{x}$ et tout y. Par conséquent, si pour un $\vec{x}$ donné $\varphi(\vec{x})$ est vérifiée, il existe un y qui vérifie $\varphi_+(\vec{x},y)\wedge\varphi_-(\vec{x},y)$, et donc $\varphi_0(\vec{x})\wedge\tilde{\varphi}(\vec{x})$ est vérifiée. On a donc

$$\varphi(\vec{x})\Longrightarrow\varphi_0(\vec{x})\wedge\tilde{\varphi}(\vec{x})$$

De plus, $\varphi_0(\vec{x})\wedge\tilde{\varphi}(\vec{x})$ est effectivement une conjonction d'inégalités de la forme $v\cdot\vec{x}\leqslant c$.

vérification de validité

Montrons à présent que la formule obtenue implique bien la formule de départ. On considère donc un vecteur $\vec{x}$ tel que $\varphi_0(\vec{x})\wedge\tilde{\varphi}(\vec{x})$ soit vérifiée, et on veut montrer l'existence d'un élément y de K qui vérifie $\varphi_+(\vec{x},y)$ et $\varphi_-(\vec{x},y)$.

La formule φ_+ signifie que, étant donné $\vec{x}$, un élément y de K vérifie $\varphi_+(\vec{x},y)$ si et seulement si

$$\forall i\in I_+,\ y\geqslant c_i^+ + v_i^+\cdot\vec{x}$$

est vérifié, donc si et seulement si on a

$$y\geqslant\max_{i\in I_+}c_i^+ + v_i^+\cdot\vec{x}$$

De même $\varphi_-(\vec{x},y)$ est vérifiée étant donné $\vec{x}$ si et seulement si on a

$$y\leqslant\min_{j\in I_-}c_j^- - v_j^-\cdot\vec{x}$$

Par conséquent, pour $\vec{x}$ donné, $\varphi_+(\vec{x},y)\wedge\varphi_-(\vec{x},y)$ est satisfait si et seulement si y vérifie

$$\max_{i\in I_+}c_i^+ + v_i^+\cdot\vec{x}\leqslant y\leqslant\min_{j\in I_-}c_j^- - v_j^-\cdot\vec{x}$$

et donc la même formule quantifiée par $\exists y$ est vérifiée si et seulement si $\vec{x}$ satisfait

$$\max_{i\in I_+}c_i^+ + v_i^+\cdot\vec{x}\leqslant\min_{j\in I_-}c_j^- - v_j^-\cdot\vec{x}$$

Considérons alors un vecteur $\vec{x}$ satisfaisant $\tilde{\varphi}(\vec{x})$, et notons i_+ l'élément de I_+ qui maximise $c_i^+ + v_i^+\cdot\vec{x}$ et j_- l'élément de I_- qui minimise $c_j^- - v_j^-\cdot\vec{x}$. La validité de $\tilde{\varphi}(\vec{x})$ impose en particulier celle de

$$c_{i_+}^+ + v_{i_+}^+\cdot\vec{x}\leqslant c_{j_-}^- - v_{j_-}^-\cdot\vec{x}$$

et donc celle de l'inégalité précédente.

Par conséquent $\tilde{\varphi}(\vec{x})$ implique $\exists y\big(\varphi_+(\vec{x},y) \wedge \varphi_-(\vec{x},y)\big)$, et donc comme $\varphi_0(\vec{x})$ est indépendant de y, on a bien l'implication

$$\varphi_0(\vec{x}) \wedge \tilde{\varphi}(\vec{x}) \Longrightarrow \varphi(\vec{x})$$

Cette implication subsiste toujours dans le cas où l'un des ensembles d'indices est vide. On a donc bien l'équivalence dans tous les cas, et le théorème est démontré. ■

Ce résultat est formulé dans un corps, mais il s'étend immédiatement au cas où les coefficients sont entiers relatifs à partir du cas rationnel. Il existe des résultats plus généraux que celui-ci, on pourra en particulier se référer à [Sch86] pour des travaux plus poussés sur le sujet, mais le cas qui nous concerne ici ne nécessite pas de résultat plus puissant. Nous verrons même en 3.2.2 que nous n'avons besoin que d'un résultat plus restreint.

On a donc établi une méthode d'élimination des quantificateurs existentiels dans les conjonctions de contraintes linéaires larges. On admettra que ce résultat subsiste si l'on s'autorise à utiliser des inégalités strictes en plus des inégalités larges, la preuve étant fondamentalement similaire.

On peut ensuite généraliser facilement le résultat à toutes les quantifications de formules de contraintes, même si seule l'élimination du quantificateur existentiel nous servira par la suite :

corollaire 2 (*élimination des quantificateurs*)

Toute formule construite par conjonction et quantification à partir d'inégalités de la forme $\vec{v} \cdot \vec{x} \leqslant c$ est logiquement équivalente à une conjonction non quantifiée de telles inégalités.

preuve : Si l'on appelle CL le langage des contraintes linéaires, c'est-à-dire l'ensemble des conjonctions non quantifiées d'inégalités de le forme voulue, le théorème précédent prouve que le langage $\exists CL$ est équivalent à CL, et donc que $\exists^* CL$ est lui aussi équivalent à CL.

Le cas des quantifications universelles se règle plus facilement. En effet, pour toute inégalité de la forme $\vec{v} \cdot \vec{x} \leqslant c$ où le coefficient de la variable x_i est non nul et pour toute valeur des x_j avec $j \neq i$, il existe une valeur de x_i pour laquelle l'inégalité n'est pas satisfaite. Par conséquent, pour toute formule φ de CL où la variable x_i apparaît au moins une fois avec un coefficient non nul, la formule $\forall x_i \varphi$ est fausse. De plus, si x_i n'apparaît pas, $\forall x_i \varphi$ est équivalente à φ. Le langage $\forall CL$ est donc équivalent lui aussi au langage CL, et par conséquent $\forall^* CL$ l'est également.

Par récurrence, on montre alors trivialement que toute formule de CL précédée d'une alternance quelconque de quantificateurs peut être réduite à une formule de CL équivalente. ■

3.1.2 Résultats spécifiques

Le résultat précédent sur l'élimination des quantificateurs n'est pas suffisant dans le cas des contraintes utilisées dans les automates temporisés. En effet, on ne s'autorise que des inégalités dont le membre de gauche fait apparaître soit une unique variable avec un coefficient 1 ou -1 soit la différence de deux variables.

La stabilité est tout de même conservée lorsque les inégalités sont de cette forme, c'est-à-dire que l'on a la propriété suivante :

proposition 3

Si φ est une formule de contrainte temporelle au sens de la définition 1, alors toute quantification de φ est équivalente à une formule de contrainte temporelle de cette forme.

preuve : Il suffit donc de montrer que si $\varphi(x_1, \ldots, x_n)$ est une formule de contrainte temporelle, alors pour tout i la formule quantifiée $\exists x_i\, \varphi(x_1, \ldots, x_n)$ est équivalente à une formule de contrainte.

Reprenons alors les notations du théorème. Il faut donc montrer que la formule $\tilde{\varphi}$ est bien une formule de contrainte temporelle. On avait posé

$$\tilde{\varphi}(\vec{x}) \equiv \bigwedge_{(i,j)\in I_+\times I_-} \left((v_i^+ + v_j^-)\cdot\vec{x} \leqslant c_i^+ + c_j^-\right)$$

où les v_i^+ et v_i^- sont définis par les formules

$$\begin{array}{rcl} \varphi_+(\vec{x},y) & \equiv & \bigwedge_{i\in I_+}(v_i^+\cdot\vec{x} \leqslant c_i^+ + y) \\ \varphi_-(\vec{x},y) & \equiv & \bigwedge_{i\in I_-}(v_i^-\cdot\vec{x} \leqslant c_i^- - y) \end{array}$$

La forme des contraintes temporelles impose d'une part que pour chaque i de I_+, le vecteur v_i^+ ait au plus une coordonnée non nulle, celle-ci étant alors égale à 1, et d'autre part que pour chaque i de I_-, le vecteur v_i^- ait au plus une coordonnée non nulle, qui sera cette fois égale à -1. Par conséquent, les sommes $(v_i^+ + v_j^-)$ peuvent avoir au plus une coordonnée positive (alors égale à 1) et une coordonnée négative (alors égale à -1).

Les clauses de $\tilde{\varphi}$ sont donc soit triviales (de la forme $0 \leqslant c$) soit de la forme requise, et comme on dispose de la constante $\top$ tous les cas correspondent à une formule de la forme voulue. ∎

3.2 Zones d'états

On introduit maintenant la notion de zone d'états. Cette notion revient à considérer un ensemble d'états temporisés dont l'état discret est commun et dont l'ensemble des états temporels est une contrainte temporelle. L'élimination des quantificateurs dans les formules de contraintes nous permettra alors de montrer qu'il est possible de ne plus considérer que des zones d'états et non plus des états isolés.

Si l'on note $\mathcal{A}(q,v)$ l'ensemble des successeurs de l'état temporisé (q,v) par une transition unique et $\mathcal{A}(E)$ la réunion des $\mathcal{A}(q,v)$ pour (q,v) parcourant E, l'ensemble des états atteignables à partir de S_0 sera alors le plus petit point fixe contenant S_0 de l'application $E \mapsto E \cup \mathcal{A}(E)$ sur les ensembles d'états temporisés. De plus, selon notre définition, un état est toujours successeur de lui-même par la transition continue de durée nulle, donc on a $E \subseteq \mathcal{A}(E)$ et l'ensemble des états atteignables à partir de S_0 est donc le plus petit point fixe de $\mathcal{A}$ contenant S_0.

3.2.1 Définition

Pour un automate $\mathcal{A} = (\Sigma, Q, K, I, \Delta)$ et un état (q,v) donné valide, c'est-à-dire tel que v soit élément de $I(q)$, l'ensemble des successeurs de (q,v) s'exprime de la façon suivante, en appliquant directement notre définition :

$$\begin{array}{rcl} \mathcal{A}(q,v) & = & \big\{(q, v+\tau) \,\big|\, (\tau \geqslant 0) \wedge \big(v+\tau \in I(q)\big)\big\} \\ & \cup & \big\{(q', \pi_z(v)) \,\big|\, \big((a,q,\chi,z,q') \in \Delta\big) \wedge (v \in \chi) \wedge \big(v \in I(q')\big)\big\} \end{array}$$

en effet si (q,v) est valide, $v+\tau \in I(q)$ est équivalent à $\forall t \in [0,\tau], v+t \in I(q)$ puisque $I(q)$ est convexe, car il est défini par une conjonction de contraintes linéaires.

Or le but de notre démarche est d'étudier l'ensemble des états atteignables à partir d'un ensemble donné d'états initiaux. On a donc tout intérêt à considérer des ensembles d'états plutôt que des états isolés. On définira donc la notion suivante :

définition 4 (*zone d'états*)

On appelle zone d'états d'un automate $\mathcal{A} = (\Sigma, Q, K, I, \Delta)$ tout couple (q, C) où q est un élément de Q et où C est une contrainte temporelle sur K, donc un élément de $\mathrm{CC}(K)$.

Cette définition permet de définir de façon plus légère les ensembles de successeurs. On introduit pour cela les notations intermédiaires suivantes :

– notons *time* l'opération de passage du temps sur les états temporels, c'est-à-dire

$$time(E) = \{v + t \mid v \in E, t \geqslant 0\}$$

– étendons ensuite l'opérateur *time* aux zones d'états en posant la définition suivante qui tient compte de l'invariant :

$$time(q, C) = \big(q, time(C) \cap I(q)\big)$$

On s'autorise également à appliquer cet opérateur à des ensembles de zones d'états de façon naturelle ;
– on note toujours $\pi_z(E)$ la projection de E selon les variables de z, et on peut alors définir pour chaque transition discrète δ l'opérateur $trans_\delta$ par

$$trans_{(a,q,\chi,z,q')}(q, C) = \big(q', \pi_z(C \cap \chi) \cap I(q')\big)$$

On en déduit l'opération *trans* sur les ensembles de zones, qui applique toutes les transitions possibles :

$$trans(E) = \{trans_\delta(q, C) \mid (q, C) \in E, \delta = (a, q, \chi, z, q') \in \Delta\}$$

À supposer que l'image par ces opérateurs d'une zone d'états soit elle-même une zone d'états, $time(q, C)$ sera donc la zone des successeurs valides d'états de (q, C) par transition continue. De même $trans_\delta(q, C)$ sera la zone des successeurs d'états de (q, C) par la transition discrète δ. On peut donc définir l'action de l'automate $\mathcal{A}$ sur les ensembles de zones d'états par la formule

$$\mathcal{S}_{\mathcal{A}}(E) = time(E) \cup trans(E)$$

dont nous allons montrer à présent qu'elle est valide, c'est-à-dire qu'elle fournit bien des ensembles de zones d'états.

3.2.2 Propriétés de stabilité

Nous montrons ici que l'ensemble des contraintes temporelles est stable par chacune des opérations que nous utilisons dans l'expression des ensembles de successeurs.

En dehors du cas trivial de l'intersection de contraintes qui correspond à la conjonction de formules, ceci vient du fait que les opérateurs utilisés s'expriment tous par quantification existentielle :
– Le cas de la projection est simple. En effet, par définition, la projection d'un ensemble E selon les variables de z est l'ensemble des projections des éléments de E, donc un état temporel v est élément de $\pi_z(E)$ si et seulement si E contient un élément dont v est la projection, ce qui s'exprime en termes de formules de contrainte par

$$\pi_{\{z_1,\dots,z_k\}}(\varphi) \Longleftrightarrow (\exists z_1 \cdots \exists z_k \varphi) \wedge (z_1 = 0) \wedge \cdots \wedge (z_k = 0)$$

si φ représente E et si $\pi_z(\varphi)$ est défini comme étant une formule qui représente $\pi_z(\hat{\varphi})$;
– Le cas du passage du temps est similaire, en effet en se basant sur la définition on peut écrire

$$time(E) = \{v + \tau \mid v \in E, \tau \geqslant 0\} = \{v \mid \exists \tau \geqslant 0, v - \tau \in E\}$$

et donc si E est l'interprétation d'une formule $\varphi(x_1, \dots, x_n)$, l'ensemble $time(E)$ est l'interprétation de $\exists\tau\big(\varphi(x_1 - \tau, \dots, x_n - \tau) \wedge (\tau \geqslant 0)\big)$, et il est facile de voir que le fait de substituer à chaque variable x l'expression $x - t$ aboutit bien (après simplification) à des contraintes de la forme voulue.

Par conséquent, comme les contraintes temporelles sont par définition des interprétations de formules de contraintes, on en déduit la stabilité de l'ensemble des contraintes temporelles par les opérateurs voulus :

proposition 4 (*stabilité des contraintes*)

> *Soit $\mathcal{A} = (\Sigma, Q, K, I, \Delta)$ un automate temporisé. Soient C et C' des contraintes temporelles sur K, q un élément de Q et z une partie de K, alors*
> - *$C \cap C'$ est une contrainte temporelle ;*
> - *$\pi_z(C)$ est une contrainte temporelle ;*
> - *$time_q(C)$ est une contrainte temporelle.*

En conséquence de ce résultat, l'ensemble des successeurs d'une zone d'états est bien un ensemble de zones d'états et notre opérateur $\mathcal{S}_\mathcal{A}$ est valide.

Afin d'expliciter la correspondance de cet opérateur avec le précédent, notons $st(E)$ l'ensemble des états appartenant à une zone de E, c'est-à-dire

$$st(E) = \{(q, v) \mid \exists (q, C) \in E, v \in C\}$$

On vérifie alors aisément la validité de la correspondance suivante qui montre que l'opérateur $\mathcal{S}_\mathcal{A}$ agit intuitivement de la même façon que $\mathcal{A}$:

$$\mathcal{A}\big(st(E)\big) = st\big(\mathcal{S}_\mathcal{A}(E)\big)$$

3.2.3 Ensembles de zones

Par conséquent, si un ensemble de zones d'états est un point fixe de $\mathcal{S}_\mathcal{A}$, alors son image par st est un point fixe de $\mathcal{A}$. La relation précédente permet même de montrer que si E est le plus petit point fixe de $\mathcal{S}_\mathcal{A}$ contenant E_0, alors $st(E)$ est le plus petit point fixe de $\mathcal{A}$ contenant $st(E_0)$. On notera bien cependant que $st(E)$ peut *a priori* être un point fixe de $\mathcal{A}$ sans que E soit un point fixe de $\mathcal{S}_\mathcal{A}$.

Le problème d'atteignabilité initial peut donc se ramener à décider si, pour un ensemble E_0 de zones initiales et un ensemble E_f de zones finales, le plus petit point fixe E de $\mathcal{S}_\mathcal{A}$ contenant E_0 est tel que $st(E)$ intersecte $st(E_f)$.

On voudrait cependant pouvoir ne considérer que des zones d'états et non plus les ensembles d'états sous-jacents, afin de ne manipuler effectivement que des objets dont le nombre est fini. Ceci est possible en remarquant que pour deux ensembles E et F de zones d'états, $st(E)$ et $st(F)$ s'intersectent si et seulement s'il existe un élément q de Q et deux contraintes C_E et C_F telles que
- (q, C_E) soit élément de E ;
- (q, C_F) soit élément de F ;
- $C_E \cap C_F$ soit non-vide.

Il reste simplement à savoir décider si une contrainte donnée est vide ou pas, mais ce problème se résout par élimination de quantificateurs. En effet, si l'on considère une contrainte définie par la formule $\varphi(x_1, \dots, x_n)$, savoir si cette contrainte est vide revient à décider la validité de la formule

$$\exists x_1 \cdots \exists x_n \varphi(x_1, \dots, x_n)$$

On peut donc maintenant considérer des ensembles de zones d'états au lieu d'ensembles d'états. L'intérêt est que, pour un automate temporisé donné, l'ensemble des contraintes susceptibles d'intervenir dans l'une des zones considérées en cours de calcul peut être limité à un nombre fini. En effet, toute contrainte faisant intervenir une constante supérieure (en valeur absolue) à toutes celles apparaissant dans les contraintes de l'automate et dans la définition des ensembles E_0 et E_f peut être réduite en une contrainte $\top$ ou $\perp$ selon sa forme. Une justification précise de ce résultat se trouve dans [Alu98] mais elle utilise le formalisme des *clock regions*, un peu différent de celui utilisé ici (il s'agit en quelque sorte de définir des zones élémentaires disjointes et de considérer des ensembles de telles zones ; nos zones d'états sont alors les réunions convexes de ces zones élémentaires).

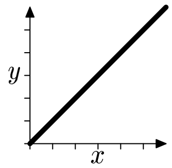
Cet argument de finitude ne rend pas calculable, en règle générale, le plus petit point fixe de $\mathcal{S}_\mathcal{A}$ contenant un ensemble de contraintes donné, car celui-ci peut faire intervenir une ensemble infini de zones d'états, mais il permet de décider l'atteignabilité d'un ensemble de zones donnés.

3.2.4 Exemple

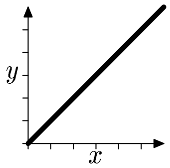
Construisons par étapes l'ensemble des états atteignables à partir de l'état $(s_0, (0,0))$ dans l'automate de la figure 1. Notons E l'ensemble constitué de la zone réduite à cet état. On représentera ici les contraintes temporelles par des formules dont elles sont l'interprétation.

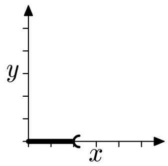
Une application de *time* sur E donne une zone d'états représentée par une demi-droite du plan des états temporels :

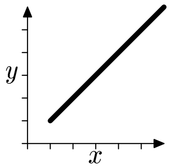
$$\mathit{time}(E) = \big\{ \big(s_0, (0 \leqslant x - y \leqslant 0) \wedge (x \geqslant 0)\big) \big\}$$



L'application de *trans* sur le résultat donne deux zones, l'une avec l'état s_1, correspondant à la projection selon y de cette zone intersectée avec $x < 2$, et l'autre dans l'état s_2 correspondant à son intersection avec $x \geqslant 1$. L'application de *time* sur $\mathit{time}(E)$ laisse celui-ci invariant, donc $\mathcal{S}_\mathcal{A}(\mathit{time}(E))$ se représente visuellement par

s_0 : 

s_1 : 

s_2 : 

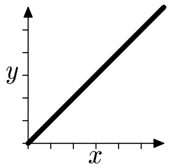
soit formellement

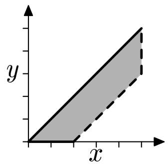
$$\mathcal{S}_\mathcal{A}(\mathit{time}(E)) = \left\{ \begin{array}{c} \big(s_0, (0 \leqslant x - y \leqslant 0) \wedge (x \geqslant 0)\big) \\ \big(s_1, (0 \leqslant x < 2) \wedge (0 \leqslant y \leqslant 0)\big) \\ \big(s_2, (0 \leqslant x - y \leqslant 0) \wedge (x \geqslant 1)\big) \end{array} \right\}$$

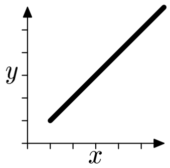
Une application de *time* sur le résultat laisse inchangées les zones concernant s_0 et s_2 mais remplacent la zone concernant s_1 par une nouvelle. On obtient formellement

$$\mathit{time}(\mathcal{S}_\mathcal{A}(\mathit{time}(E))) = \left\{ \begin{array}{c} \big(s_0, (0 \leqslant x - y \leqslant 0) \wedge (x \geqslant 0)\big) \\ \big(s_1, (x < 5) \wedge (y \geqslant 0) \wedge (0 \leqslant x - y < 2)\big) \\ \big(s_2, (0 \leqslant x - y \leqslant 0) \wedge (x \geqslant 1)\big) \end{array} \right\}$$

soit en termes de parties du plan

s_0 : 

s_1 : 

s_2 : 

Puis on continuerait de cette façon jusqu'à obtenir un ensemble de zones dont la réunion serait stable.

4 Système de réécriture

Nous avons donc réduit notre problème sur des ensembles d'états d'un automate temporisé en un problème portant sur des familles de zones d'états. En remplaçant les contraintes temporelles par des formules qui les représentent, on obtient alors des familles de couples dont un membre est pris dans un ensemble fini et dont l'autre membre est une formule d'un langage simple. On a donc ainsi des termes manipulables par réécriture.

Deux types principaux de règles seront nécessaires : d'une part, il faut des règles pour effectuer des calculs sur les zones d'états, et d'autre part il faut des règles pour rendre compte du fonctionnement de l'automate.

Sauf mention contraire, afin d'alléger les énoncés qui suivent, on fera à présent implicitement référence à un automate temporisé $\mathcal{A} = (\Sigma, Q, K, I, \Delta)$ donné.

4.1 Le vocabulaire

Les termes manipulés représenteront donc les ensembles de zones d'états considérés dans la formulation du problème à laquelle nous sommes parvenus.

Les formules de contrainte temporelle sont des conjonctions de contraintes linéaires, mais leur forme est imposée. On n'utilisera donc pas des termes permettant de représenter des inégalités de la forme $\vec{v} \cdot \vec{x} \leqslant c$ mais on introduira un constructeur pour chaque forme autorisée. On inclut donc dans le vocabulaire utilisé

- un symbole de constante par élément de K ;
- les entiers relatifs ;
- les constantes $\top$ et $\bot$;
- les constructeurs binaires

$$\_ \leqslant \_ \qquad \_ < \_ \qquad \_ \geqslant \_ \qquad \_ > \_$$

  où le membre de gauche est un élément de K et le membre de droite un entier ;
- les constructeurs ternaires

$$\_ - \_ \leqslant \_ \qquad \_ - \_ < \_$$

  où les deux premiers membres sont des éléments de K (que l'on supposera toujours distincts) et le troisième membre un entier.

De cette façon tous les types de contraintes voulus sont autorisés. Les constantes $\top$ et $\bot$, interprétées respectivement comme les valeurs booléennes vraie et fausse, permettront de considérer des cas dégénérés sans problème. On définit donc le langage des clauses de contraintes linéaires comme étant composé des deux constantes booléennes et des termes formés par ces constructeurs.

On introduit ensuite le constructeur binaire $\_ \wedge \_$ et on définit donc le langage des contraintes linéaires comme la clôture du langage des clauses par ce constructeur. On considérera *a priori* la conjonction comme un opérateur commutatif et associatif, ce qui ne peut que simplifier les règles. De plus, on suppose toujours que nos conjonctions de clauses contiennent une occurrence du symbole $\top$.

On ajoute enfin les constructeurs $\mathit{proj}(\_, \_)$ et $\mathit{time}(\_)$ afin de représenter les fonctions que nous utiliserons.

4.2 Règles de réécriture pour les zones

En dehors de l'intersection de zones, qui se traduit trivialement sur les termes par une conjonction, toutes les opérations dont nous avons besoin sont des applications de l'élimination des quantificateurs. Cependant, plutôt que de transcrire directement la méthode constructive introduite dans la preuve, nous allons plutôt décrire chaque opération par des règles spécifiques, afin de limiter le nombre de règles à appliquer pour réduire un terme.

4.2.1 Projections

Commençons par définir les règles pour la projection selon une variable x. La première chose à faire est donc de partager l'ensemble des clauses selon le statut de x. On définit donc un constructeur ternaire $[\_|\_|\_]$ contenant trois listes de clauses que l'on interprétera respectivement comme celles de φ_0, φ_+ et φ_- selon les notations précédentes. Pour le calcul, on ajoute également un opérateur $\_ \gg [\_|\_|\_]$, en comprenant le terme $C \gg [u|v|w]$ comme une étape du calcul lors de laquelle u, v et w sont les trois parties du terme initial déjà établies et où C reste à traiter. De cette façon, on peut définir les règles suivantes :

$$[\mathit{div}'(x)]\ \frac{t \in \{x < c,\ x \leqslant c,\ x - y < c,\ x - y \leqslant c\}}{t \wedge C \gg [u|v|w] \longrightarrow C \gg [u|v|t \wedge w]}$$

$$[div'(x)]\,\frac{t \in \{x > c,\, x \geqslant c,\, y - x < c,\, y - x \leqslant c\}}{t \wedge C \gg [u|v|w] \longrightarrow C \gg [u|t \wedge v|w]}$$

$$[div'(x)]\,\frac{x \text{ n'apparaît pas dans } t}{t \wedge C \gg [u|v|w] \longrightarrow C \gg [t \wedge u|v|w]}$$

Et on définit une règle supplémentaire pour terminer le processus en utilisant la convention selon laquelle toute conjonction de clauses contient une occurrence de $\top$:

$$[div'(x)]\ \ \top \gg [u|v|w] \longrightarrow [u|v|w]$$

On définit donc la stratégie $div(x)$ comme consistant à appliquer la règle $div'(x)$ tant que possible sur un terme, ce qui fait bien passer d'un terme $C \gg [u|v|w]$ au triplet de conjonctions de clauses voulu.

La projection du terme t selon x consiste donc à traiter les trois parties obtenues en réduisant le terme $t \gg [\top|\top|\top]$ par la stratégie $div(x)$. Il reste donc à construire les combinaisons linéaires, selon la méthode précédemment exposée. On commence donc par définir un constructeur $\_ + \_$ et les règles associées pour combiner les clauses. Ces règles seront regroupées sous le nom $comb(x)$, et on suppose que le premier membre fait partie du φ_+ associé à la formule initiale et que le second fait partie de φ_- :

$$[comb(x)] \begin{cases} (x \geqslant c) \ + (x - y \leqslant d) \longrightarrow \ (y \geqslant c - d) \\ (y - x \leqslant c) + \ (x \leqslant d) \ \longrightarrow \ (y \leqslant c + d) \\ (y - x \leqslant c) + (x - z \leqslant d) \longrightarrow (y - z \leqslant c + d) \end{cases}$$

On ajoute également deux règles conditionnelles pour les cas restants :

$$[comb(x)]\,\frac{c \leqslant d}{(x \leqslant c) + (x \geqslant d) \longrightarrow \top} \qquad [comb(x)]\,\frac{c > d}{(x \leqslant c) + (x \geqslant d) \longrightarrow \bot}$$

On ajoute également les règles correspondantes dans les cas où certaines des inégalités sont strictes.

Il reste alors à appliquer ces règles de combinaison de clauses à tous les couples possibles de termes. On va pour cela définir de façon générique un double parcours de liste à partir d'une stratégie S chargée de réduire un terme $t + t'$ à la combinaison de t et t'.

On définira ce parcours par deux règles et deux stratégies. Le principe sera de réduire un terme de la forme $[u|v|w]$ en la conjonction de u et de toutes les clauses obtenues en combinant v et w. Dans les règles suivantes, t, t' et t'' représentent des clauses et u, u', C et C' des conjonctions de clauses. La première règle suppose que le terme v est réduit à une clause et parcourt simplement le terme w :

$$[appOne(S)] \begin{cases} \dfrac{S : t + t' \longrightarrow t''}{[u|t|t' \wedge C] \longrightarrow [u \wedge t''|t|C]} \\ \\ [u|t|\top] \longrightarrow u \end{cases}$$

On définit alors la stratégie $appOne'(S)$ comme consistant à répéter tant que possible $appOne(S)$. Cette stratégie fournit donc une conjonction à partir d'un terme $[u|t|w]$. On définit alors la seconde règle dont le rôle est d'appliquer cette stratégie avec chaque clause de la première liste :

$$[appList(S)] \begin{cases} \dfrac{appOne'(S) : [u|t|C'] \longrightarrow u'}{[u|t \wedge C|C'] \longrightarrow [u'|C|C']} \\ \\ [u|\top|C'] \longrightarrow u \end{cases}$$

De même que précédemment, on pose enfin que la stratégie $app(S)$ consiste à répéter $appList(S)$ tant que possible, ce qui réduit un terme $[u|v|w]$ en la conjonction voulue.

On obtiendra donc la projection du terme t selon x en appliquant la stratégie $app\big(comb(x)\big)$ au terme obtenu en réduisant t par la stratégie $div(x)$. Il ne reste alors qu'à ajouter au terme obtenu un équivalent de $x = 0$, donc la conjonction $(x \leqslant 0) \wedge (x \geqslant 0)$ afin de rendre compte sur x de la projection, d'où la règle de réduction principale pour la projection :

$$\frac{div(x) : t \gg [\top|\top|\top] \longrightarrow [u|v|w] \qquad app\big(comb(x)\big) : [u|v|w] \longrightarrow u'}{proj(t,x) \longrightarrow u' \wedge (x \leqslant 0) \wedge (x \geqslant 0)}$$

4.2.2 Opérateur *time*

Les règles définissant l'application de l'opérateur *time* sur un formule de contrainte seront similaires à celles pour les projections. On commencera à nouveau par diviser les clauses en trois groupes. Cependant, l'application directe de la méthode ferait apparaître une variable t supplémentaire. On contourne en fait le problème en utilisant la propriété suivante de l'opérateur *time* :

lemme 5

Soit C une contrainte temporelle représentée par la formule φ. On suppose pour l'énoncé que toutes les inégalités sont larges. écrivons φ sous la forme

$$\varphi \equiv \bigwedge_{i\in I}(x_i - y_i \leqslant c_i) \wedge \bigwedge_{j\in J}(x_j \geqslant d_j) \wedge \bigwedge_{k\in K}(x_k \leqslant e_k)$$

Alors la contrainte $time(C)$ est représentée par la formule

$$\bigwedge_{i\in I}(x_i - y_i \leqslant c_i) \wedge \bigwedge_{j\in J}(x_j \geqslant d_j) \wedge \bigwedge_{(j,k)\in J\times K}(x_k - x_j \leqslant e_k - d_j)$$

preuve : Reprenons à nouveau les notations du théorème 1. L'opérateur *time* sur les formules de contrainte se définit par la formule

$$time\big(\varphi(\vec{x})\big) \equiv \exists t\, \varphi(\vec{x} - t) \wedge (t \geqslant 0)$$

Il est alors clair que les clauses se partagent en

$$\begin{aligned}\varphi_0 &\equiv \textstyle\bigwedge_{i\in I}(x_i - y_i \leqslant c_i)\\ \varphi_+ &\equiv \textstyle\bigwedge_{k\in K}(x_k \leqslant e_k + t) \wedge (0 \leqslant t)\\ \varphi_- &\equiv \textstyle\bigwedge_{j\in J}(-x_j \leqslant d_j - t)\end{aligned}$$

en substituant à chaque x_i l'expression $x_i - t$. Par conséquent, le premier membre est bien celui attendu. D'autre part, le terme $\tilde{\varphi}$ sera composé d'une part de la combinaison de $0 \leqslant t$ avec chaque clause de φ_- et d'autre part de la combinaison de chacune des autres clauses de φ_+ avec celles de φ_-. Or la combinaison de $0 \leqslant t$ avec un $-x_j \leqslant d_j - t$ donne $-x_j \leqslant d_j$ donc le deuxième membre du résultat est bien la conjonction des $x_j \geqslant d_j$.

Enfin, le troisième membre est la conjonction des combinaisons de clauses de φ_+, à l'exception de $0 \leqslant t$, et de clauses de φ_-, ce qui implique que le troisième membre du résultat s'écrit

$$\bigwedge_{(j,k)\in J\times K}(x_k - x_j \leqslant e_k - d_j)$$

Et on obtient bien le terme attendu. ∎

On admettra une fois de plus que le résultat s'étend naturellement au cas où certaines clauses contiennent des inégalités strictes.

On utilisera à nouveau la notation $[\_|\_|\_]$ mais le premier membre contiendra donc maintenant les clauses de la forme $x - y \leqslant c$ ou $x - y < c$, le deuxième contiendra les clauses de la forme

$x \geqslant c$ ou $x > c$ et le troisième contiendra les termes de la forme $x \leqslant c$ ou $x < c$. On procède donc comme pour la définition de la stratégie $div(x)$ en créant ici une stratégie *choose* définie comme l'application tant que possible de l'une des règles *choose'* suivantes sur un terme :

$$[choose']\,\frac{t \in \{x - y \leqslant c,\; x - y < c\}}{t \wedge C \gg [u|v|w] \longrightarrow C \gg [t \wedge u|v|w]}$$

$$[choose']\,\frac{t \in \{x \geqslant c,\; x > c\}}{t \wedge C \gg [u|v|w] \longrightarrow C \gg [u|t \wedge v|w]}$$

$$[choose']\,\frac{t \in \{x \leqslant c,\; x < c\}}{t \wedge C \gg [u|v|w] \longrightarrow C \gg [u|v|t \wedge w]}$$

$$[choose']\;\top \gg [u|v|w] \longrightarrow [u|v|w]$$

L'étape suivante est à nouveau la combinaison des clauses du deuxième et du troisième membre du terme $[u|v|w]$ obtenu. On n'aura ici qu'à combiner des termes simples, on peut donc poser les règles simples suivantes :

$$[combine]\;(x \geqslant c) + (y \leqslant d) \longrightarrow (y - x \leqslant d - c)$$

$$[combine]\,\frac{d \geqslant c}{(x \geqslant c) + (x \leqslant d) \longrightarrow \top} \qquad [combine]\,\frac{d < c}{(x \geqslant c) + (x \leqslant d) \longrightarrow \bot}$$

et on ajoute les règles correspondantes pour les cas où il y a des inégalités strictes dans les clauses.

Le terme final s'obtiendra donc en réduisant le terme intermédiaire $[u|v|w]$ par la stratégie *app*(*combine*) et en construisant la conjonction du terme obtenu avec le terme v, puisque comme nous l'avons montré les clauses de la forme $x \geqslant c$ ou $x > c$ doivent être conservées :

$$\frac{choose : C \gg [\top|\top|\top] \longrightarrow [u|v|w] \qquad app(combine) : [u|v|w] \longrightarrow C'}{time(C) \longrightarrow C' \wedge v}$$

4.3 Transcription de l'automate

Une fois ces outils de calcul sur les contraintes temporelles établis, il est maintenant possible de transcrire le fonctionnement d'un automate temporisé par des règles de réécriture sur des termes adaptés.

Une approche possible est de considérer des termes représentant des zones d'états, c'est-à-dire des termes contenant un état discret et une formule de contrainte temporelle. Dans cette vision des choses, les transitions deviennent les règles de réécriture sur ces termes. La résolution d'un problème d'atteignabilité consiste alors en un parcours de l'ensemble des termes obtenus en réduisant un terme représentant un état initial.

On commence donc par ajouter à notre vocabulaire un symbole de constante par état, donc par élément de Q. On définit de plus le constructeur $\_/\_$ dont le premier membre est un état et dont le second membre est une formule de contrainte. On appellera *terme d'état* tout terme de la forme q/φ.

4.3.1 Transcription des transitions

Un état d'un automate temporisé peut subir deux types de transitions : les transitions continues et les transitions discrètes. On définira donc une famille de règles de réécriture pour chaque type de transition.

On va donc définir une règle de réduction *time* chargée de représenter l'application d'une transition continue sur une zone d'états. Il s'agit simplement de représenter le calcul de la fonction *time* sur les zones d'états, donc on peut écrire

$$[\mathit{time}]\ \ q/\varphi \longrightarrow q/(\mathit{time}(\varphi) \wedge \mathit{inv}(q))$$

où *inv* est un symbole fonctionnel dont la réduction associe son invariant à chaque état. Cette règle est dépendante de l'automate et doit donc être définie spécifiquement.

Dans le cas de l'exemple de la figure 1, les règles de réduction pour le symbole *inv* sont simplement

$$\begin{array}{l} \mathit{inv}(s_0) \longrightarrow \top \\ \mathit{inv}(s_1) \longrightarrow (x < 5) \wedge \top \\ \mathit{inv}(s_2) \longrightarrow \top \end{array}$$

On définit parallèlement pour chaque transition (a, q, χ, z, q') une variante de la règle *trans* pour rendre compte du passage de la transition discrète. On va en fait distinguer les règles selon leur étiquetage, on verra dans la section 5 que cet étiquetage sert également à effectuer des produits d'automates. On note toujours $z = \{z_1, \dots, z_k\}$.

$$[\mathit{trans}(a)]\ \ q/\varphi \longrightarrow q'/\mathit{proj}(\cdots \mathit{proj}(\chi \wedge \varphi, z_1) \cdots z_k) \wedge \mathit{inv}(q')$$

Quitte à supposer que les contraintes des transitions sont compatibles avec les invariants des états d'arrivée, c'est-à-dire que pour toute transition (a, q, χ, z, q'), si un état temporisé satisfait $I(q)$ et χ alors son successeur par cette transition satisfait toujours $I(q')$, on peut simplifier la règle précédente en

$$[\mathit{trans}(a)]\ \ q/\varphi \longrightarrow q'/\mathit{proj}(\cdots \mathit{proj}(\chi \wedge \varphi, z_1) \cdots z_k)$$

qui a l'avantage de ne plus faire intervenir l'invariant.

Ces règles sont également dépendantes de l'automate et doivent être codées spécifiquement à l'automate considéré. Dans le cas de l'exemple de la figure 1, on a donc les règles suivantes :

$$[\mathit{trans}(a)] \left\{ \begin{array}{l} s_0/\varphi \longrightarrow s_1/\mathit{proj}((x < 2) \wedge \varphi, x) \\ s_1/\varphi \longrightarrow s_2/(y < 1) \wedge \varphi \\ s_2/\varphi \longrightarrow s_0/(y > 2) \wedge \varphi \end{array} \right.$$

$$[\mathit{trans}(b)] \left\{ \begin{array}{l} s_0/\varphi \longrightarrow s_2/(x \geqslant 1) \wedge \varphi \\ s_1/\varphi \longrightarrow s_1/\mathit{proj}(\varphi, y) \end{array} \right.$$

On définit ensuite la stratégie *trans* comme consistant à appliquer toutes les transitions *trans*(a) pour a parcourant Σ et à renvoyer l'ensemble des réductions obtenues.

Une trajectoire de l'automate sera donc une série de réductions d'un terme représentant une zone d'états initiaux par les règles *time* et *trans*, celles-ci représentant respectivement le passage du temps dans un état donné et le passage d'une transition discrète.

4.3.2 Validité des règles

Montrons plus précisément quelle est la correspondance entre ces deux règles de réécriture et le fonctionnement de l'automate temporisé considéré.

proposition 6

Si la règle time réduit le terme q/φ en un terme q/ψ et si $(q, \hat{\varphi})$ est une zone d'états valide, alors $(q, \hat{\psi})$ est la zone des états atteignables à partir de ceux de $(q, \hat{\varphi})$ par transition continue.

preuve : Par définition de la règle *time*, le terme ψ est une conjonction de contraintes temporelles équivalente à $\varphi' \wedge I(q)$ où φ' est la réduction de $time(\varphi)$. De plus, d'après le lemme 5, le terme $time(\varphi)$ se réduit en une formule de contrainte temporelle dont l'interprétation est $time(\hat{\varphi})$. Par conséquent, on a

$$\hat{\psi} = time(\hat{\varphi}) \cap I(q)$$

Considérons alors un état (q, v) de la zone $(q, \hat{\psi})$. L'état temporel v est élément de $time(\hat{\varphi})$, donc il existe un état v' dans la zone $\hat{\varphi}$ et un réel positif t tels que $v = v' + t$. De plus, v satisfait l'invariant de q puisque $\hat{\psi} \subseteq I(q)$ et v' satisfait également cet invariant puisque $(q, \hat{\varphi})$ est supposé valide. (q, v) succède donc à (q, v') par passage du temps dans le respect de l'invariant de q.

Réciproquement, considérons un état temporel v dans $\hat{\varphi}$ et un réel positif t tels que $v + t$ satisfasse l'invariant de q. Alors par définition de l'opérateur *time* sur les ensembles d'états temporels, $v + t$ est élément de $time(\hat{\varphi})$. De plus, comme $v + t$ satisfait l'invariant de q, $v + t$ est élément de $I(q)$. $(q, v + t)$ est donc un état de la zone $(q, \hat{\psi})$.

On a donc bien l'égalité attendue, ce qui prouve la validité de la règle de réduction *time*. ∎

L'autre propriété à prouver est la validité de la règle *trans*. L'énoncé formel de cette propriété est plus délicat que pour la règle *time* car on ici affaire à une règle non déterministe :

proposition 7

Soit q/φ un terme d'état. L'ensemble des réductions possibles de q/φ par la règle trans définit un ensemble de zones d'états égal à l'ensemble des zones successeurs de $(q, \hat{\varphi})$ par transition discrète.

preuve : Considérons une application possible de la règle *trans* au terme q/φ réduisant celui-ci en le terme q'/ψ. Cette règle est associée à une transition $(a, q, \hat{\chi}, z, q')$ pour une formule de contrainte χ donnée. Par définition de la règle, la formule ψ est alors équivalente à la réduction du terme

$$proj(\cdots proj(\varphi \wedge \chi, z_1) \cdots, z_k)$$

où l'on note $z = \{z_1, \ldots, z_k\}$. Comme les règles de réduction de l'opérateur *proj* sont une transcription directe de la méthode d'élimination des quantificateurs exposée dans la preuve du théorème 1, la réduction de ce terme est équivalente à

$$\exists z_k \cdots \exists z_1 (\varphi \wedge \chi) \wedge (z_1 = 0) \wedge \cdots \wedge (z_k = 0)$$

Par conséquent un état temporel v vérifie ψ si et seulement s'il existe un état v' satisfaisant χ et φ dont il soit la projection selon $z_1, \ldots, z_k$, c'est-à-dire si et seulement si v est l'état temporel succédant après transition instantanée à un état v' de $\hat{\varphi}$ vérifiant la contrainte χ de la transition.

Ceci signifie que l'ensemble des états temporels satisfaisant ψ est l'ensemble des successeurs des états satisfaisant φ et la contrainte de la transition considérée, donc $(q', \hat{\psi})$ est la zone des états successeurs des états de la zone $(q, \hat{\varphi})$ par la transition $(q, q', \hat{\chi}, z)$

L'ensemble des réductions possibles de q/φ est donc un ensemble de termes dont les interprétations sont les zones successeurs de $(q, \hat{\varphi})$ par chacune des transitions possibles partant de l'état q (certaines de ces zones peuvent être vides), et on retrouve donc bien le résultat voulu. ∎

Les règles de réécriture définies ici définissent donc bien un système fonctionnant intuitivement comme l'automate temporisé initial.

En particulier, il découle immédiatement la correspondance suivante avec la notion de successeur définie précédemment. Il s'agit surtout d'une formalisation de la correspondance que l'on attend entre les deux formes des opérateurs.

proposition 8

Soit un ensemble $E = \{q_i/\varphi_i \,|\, i \in I\}$ de termes d'états. Soit

$$F = \{(q_i, \hat{\varphi}_i) \,|\, i \in I\}$$

l'ensemble des zones d'états correspondantes. Soit alors E' l'ensemble des termes obtenus à partir de ceux de E par une réduction unique selon l'une des règles time et trans. Alors on a l'égalité

$$\{(q, \hat{\varphi}) \,|\, q/\varphi \in E'\} = \mathcal{S}_{\mathcal{A}}$$

preuve : Cette correspondance découle immédiatement des résultats précédents, car la réduction par *time* d'un terme q/φ correspond à l'application de l'opérateur $time_q$ sur la zone correspondante et car la réduction par *trans* correspond à la modification de l'état et à la projection selon les variables associées à la transition. ■

Il est alors clair qu'un état est atteignable à partir d'une zone d'états initiale si et seulement si un terme représentant cette zone d'états peut se réduire en un certain nombre d'applications de ces deux familles de règles en un terme représentant une zone dont cet état fait partie.

Montrons plus précisément comment formuler un problème d'atteignabilité dans un automate fini à l'aide de ce système de réécriture :

proposition 9

Soit S_0 une zone d'états représentée par le terme q_0/φ_0 et soit S_f une zone d'états représentée par le terme q_f/φ_f. Le problème de l'atteignabilité de S_f à partir de S_0 est équivalent au problème consistant à décider s'il existe un terme q_f/ψ obtenu à partir de q_0/φ_0 par un certain nombre d'applications des règles time et trans pour lequel la formule $\psi \wedge \varphi_f$ soit satisfiable.

preuve : La zone d'états S_f est atteignable à partir de S_0 si et seulement s'il existe un entier n tel que S_f intersecte l'une des zones de ${\mathcal{S}_{\mathcal{A}}}^n(\{S_0\})$. D'après la proposition précédente, ceci signifie qu'il existe parmi l'ensemble des réductions possibles de q_0/φ_0 par n applications successives de l'une des règles *time* et *trans* un terme q/φ tel que la zone $\hat{\varphi} \cap \hat{\varphi}_f$ soit non vide, c'est-à-dire tel que $\varphi \wedge \varphi_f$ soit satisfiable.

Réciproquement, s'il existe un terme q_f/φ obtenu après n applications de l'une des deux règles sur le terme q_0/φ_0 tel que $\varphi \wedge \psi$ soit satisfiable, alors la zone d'états obtenue par les transitions correspondantes à partir de S_0 intersecte S_f, donc l'équivalence est établie. ■

4.4 Résolution effective

On a donc formulé le problème de l'atteignabilité d'une zone d'états dans notre système de réécriture. Il reste à établir un moyen de le résoudre effectivement, c'est-à-dire un moyen de parcourir l'ensemble des réductions possibles du terme initial de telle sorte qu'il soit possible non seulement de décider si les zones obtenues intersectent la zone S_f, mais aussi de déterminer après un parcours suffisant qu'aucune zone d'états atteignable n'intersecte S_f. Il faut de plus formuler la méthode en termes de règles de réécriture et de stratégies de réduction.

La durée du stage n'a malheureusement pas permis d'établir formellement une méthode effective de résolution du problème d'atteignabilité. Une implémentation des règles précédentes à l'aide du logiciel ELAN a cependant été réalisée, permettant de calculer des transitions de l'automate et de résoudre le problème pour un nombre de transitions borné préalablement. Cette implémentation est présentée en annexe.

5 Produits d'automates

Maintenant que les problèmes d'atteignabilité ont été transcrits sous forme de problème de réécriture pour un automate simple, on peut se demander comment gérer des systèmes complexes comme l'exemple du train.

5.1 Définitions

La première chose à faire est donc de définir précisément ce qu'est un produit synchronisé d'automates temporisés. C'est ici que l'on utilise l'étiquetage des transitions par les éléments de l'ensemble Σ.

5.1.1 Notion informelle

Pour rendre compte de façon satisfaisante de ce que l'on attend d'un système synchronisé d'automates temporisés, on veut en fait que lorsque les composants sont dans un état discret donné, il ne soit possible pour un composant d'emprunter une transition étiquetée par a que si tous les composants susceptibles d'emprunter une transition étiquetée par a (indépendamment des contraintes temporelles et des états discrets) le font aussi.

Afin de clarifier notre propos, considérons la cas de la figure 3. La transition en trait plein est la seule transition présente dans le produit synchronisé des deux automates, alors que les transitions en pointillés seraient présentes dans le produit non synchronisé.

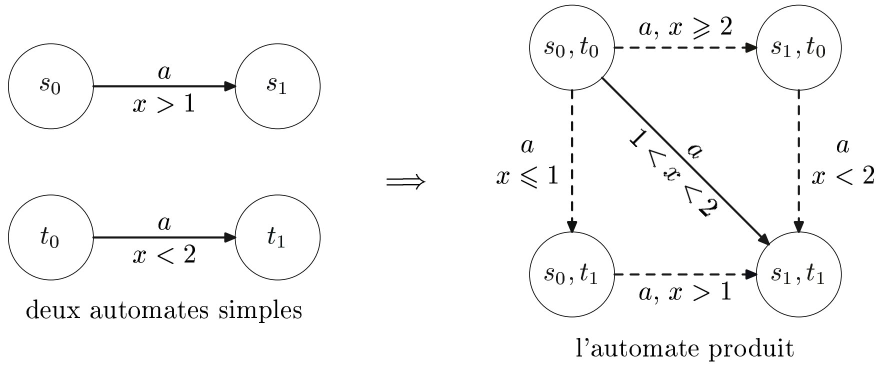


FIG. 3 – produit de deux automates minimaux

Comme autre exemple, considérons le produit synchronisé des automates TRAIN et CONTRÔLE de la figure 2. On veut que le système produit ne puisse emprunter une transition étiquetée par « approach » que si les deux automates sous-jacents empruntent simultanément une telle transition, puisque cette étiquette apparaît dans les deux parties. Par contre, le système peut emprunter une transition étiquetée par « out » même si les deux ne prennent pas une telle transition, puisque cette étiquette n'apparaît que dans l'un seul des deux automates.

5.1.2 Définition formelle

Soient $\mathcal{A} = (\Sigma, Q_A, K_A, I_A, \Delta_A)$ et $\mathcal{B} = (\Sigma, Q_B, K_B, I_B, \Delta_B)$ deux automates temporisés avec K_A et K_B disjoints. Pour C_A et C_B des contraintes temporelles sur K_A et K_B respectivement, on définit $C_A \sqcap C_B$ par

$$v \in C_A \sqcap C_B \iff v_{|K_A} \in C_A \wedge v_{|K_B} \in C_B$$

On appelle alors produit synchronisé de $\mathcal{A}$ et $\mathcal{B}$ l'automate temporisé

$$\mathcal{A} \,||\, \mathcal{B} = (\Sigma, Q_A \times Q_B, K_A \cup K_B, I, \Delta)$$

où l'on définit l'invariant I par $I(q_A, q_B) = I_A(q_A) \sqcap I_B(q_B)$. Pour définir la table de transition Δ, on introduit d'abord les notations

$$\Sigma' = \Sigma_A \cap \Sigma_B \qquad \Sigma'_A = \Sigma_A \setminus \Sigma' \qquad \Sigma'_B = \Sigma_B \setminus \Sigma'$$

On définit ensuite les trois types de transitions de l'automate produit :

$$\begin{aligned}
\Delta_A &= \left\{ \big(a, (q_A, q_B), \bar{\chi}, z, (q'_A, q_B)\big) \,\middle|\, a \in \Sigma'_A,\ (a, q_A, \chi, z, q'_A) \in \Delta_A,\ q_B \in Q_B \right\} \\
\Delta_B &= \left\{ \big(a, (q_A, q_B), \bar{\chi}, z, (q_A, q'_B)\big) \,\middle|\, a \in \Sigma'_B,\ (a, q_B, \chi, z, q'_B) \in \Delta_B,\ q_A \in Q_A \right\} \\
\Delta' &= \left\{ \big(a, (q_A, q_B), \chi_A \sqcap \chi_B, z_A \cup z_B, (q'_A, q'_B)\big) \,\middle|\, \begin{array}{c} a \in \Sigma' \\ (a, q_A, \chi_A, z_A, q'_A) \in \Delta_A \\ (a, q_B, \chi_B, z_B, q'_B) \in \Delta_B \end{array} \right\}
\end{aligned}$$

où $\bar{\chi}$ représente l'extension de la contrainte χ aux horloges de $K_A \cup K_B$ définie trivialement par

$$v \in \bar{\chi} \iff v_{|K_A} \in \chi$$

On en déduit enfin la définition de la table de transition du produit synchronisé :

$$\Delta = \Delta_A \cup \Delta_B \cup \Delta'$$

On suppose en fait K_A et K_B disjoints car la sémantique d'un produit où des variables seraient partagées serait difficile à définir et encore plus à interpréter. Il convient cependant pour avoir une définition cohérente de prouver la propriété suivante :

proposition 10

Si C_A et C_B sont des contraintes temporelles sur K_A et K_B respectivement, alors $C_A \sqcap C_B$ est une contrainte temporelle sur $K_A \cup K_B$.

preuve : Considérons φ_A et φ_B des formules de contrainte, sur K_A et K_B respectivement, dont l'interprétation est respectivement C_A et C_B. On a par définition

$$C_A \sqcap C_B = \{ v \mid v_{|K_A} \in C_A \wedge v_{|K_B} \in C_B \}$$

Pour un état temporel v sur $K_A \cup K_B$, notons

$$v = (x_1, \ldots, x_k, y_1, \ldots, y_l)$$

où les x_i sont les variables de $K_A \setminus K_B$ et les y_i celles de $K_B \setminus K_A$. On a alors

$$v \in C_A \sqcap C_B \iff \varphi_A(x_1, \ldots, x_k) \wedge \varphi_B(y_1, \ldots, y_l)$$

Donc $C_A \sqcap C_B$ est bien représenté par une formule dont les variables sont l'ensemble des x_i et des y_i. De plus $C_A \sqcap C_B$ est représenté par la conjonction des représentations de C_A et C_B. ■

5.2 Exemple

Reprenons l'exemple du train. En appliquant cette définition, le produit synchronisé de l'automate représentant le train et de l'automate représentant le système contrôle donne l'automate de la figure 4 (pour des raisons d'espace, « app » y a été substitué à « approach »).

Les états représentés en pointillés sont ceux dont on sait qu'ils ne sont pas atteignables, indépendamment des contraintes temporelles, à cause de l'orientation des transitions.

Cet exemple souligne la complexité du résultat obtenu. En effet, l'automate produit a pour nombre d'états le produit des nombres d'états des composantes, et le nombre de transitions évolue dans les mêmes proportions. Il serait donc intéressant d'établir des règles génériques pour rendre compte du fonctionnement de l'automate produit à partir des règles définissant les sous-automates.

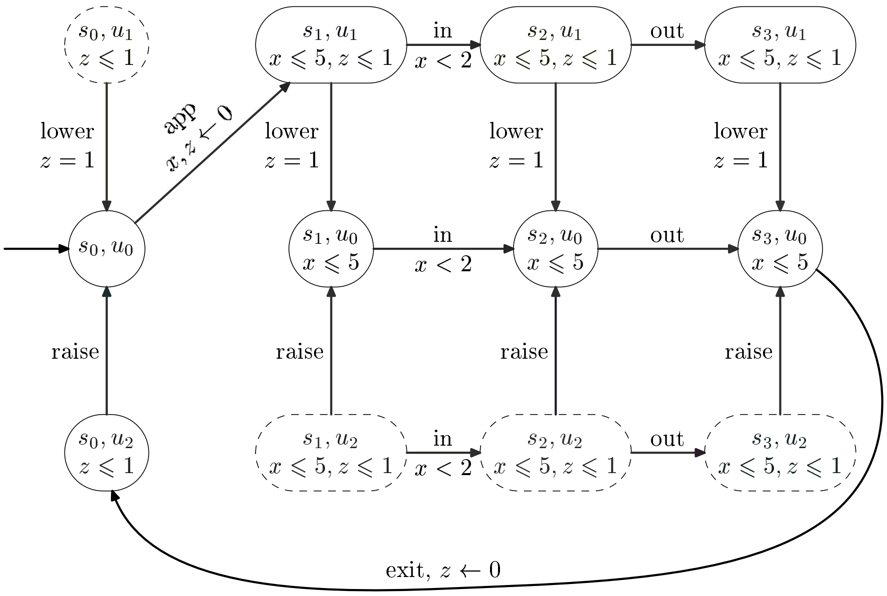


FIG. 4 – produit synchronisé de TRAIN et CONTRÔLEUR

5.3 Règles de réécriture

Considérons deux automates temporisés $\mathcal{A}$ et $\mathcal{B}$ et les systèmes de réécriture associés. Les états discrets du produit $\mathcal{A} \,||\, \mathcal{B}$ sont les couples constitués d'un état discret de $\mathcal{A}$ et d'un état discret de $\mathcal{B}$, et les zones d'états sont des contraintes sur $K_A \cup K_B$ qui se représentent par des formules de la même forme que pour les deux automates.

On introduit donc un constructeur $\_,\_$ dont le membre de gauche est un élément de Q_A et le membre de droite un élément de Q_B, et on définit l'ensemble des états discrets du produit comme l'ensemble des termes construits de cette façon. Les termes d'état du produit auront donc la forme $q_a, q_b/\varphi$.

On veut à présent déduire des règles de réduction propres aux deux sous-automates les règles correspondantes pour le produit.

5.3.1 Invariant et transitions continues

Par définition du produit synchronisé, l'invariant de l'état (q_A, q_B) est l'ensemble $I_A(q_A) \sqcap I_B(q_B)$, dont on a montré qu'il était une contrainte temporelle sur $K_A \cup K_B$, représentée par la conjonction des invariants de $\mathcal{A}$ et $\mathcal{B}$. Par conséquent la règle suivante est valide :

$$inv(q_A, q_B) \longrightarrow inv(q_A) \wedge inv(q_B)$$

Par conséquent, comme l'opérateur *time* est géré de façon indépendante de l'automate en uniquement en fonction de l'invariant, les règles de transition continue sont valides avec cette construction du produit.

5.3.2 Table de transition

Pour la table de transition, on ne pourra pas écrire de règle aussi simple parce que la définition de la table de transition du produit est beaucoup plus délicate que celle de l'invariant. On peut en fait s'en sortir en définissant une stratégie à partir des règles des sous-automates.

C'est ici qu'il est intéressant d'avoir divisé les règles de transition selon leur étiquette. En effet il est nécessaire de connaître l'étiquetage des transitions pour savoir si une transition correspond à une synchronisation ou si elle ne concerne que l'un des automates.

Pour un élément a de Σ, il est tout d'abord possible que a n'apparaisse dans aucune des transitions des deux automates dont on calcule le produit. Il n'y a alors aucune transition dans la table du produit qui ne soit étiquetée par a, et donc aucune règle de réécriture à introduire dans ce cas.

Dans le cas où a apparaît dans un seul des deux automates, la transition de l'automate produit est une simple reformulation de l'automate d'origine. On peut donc définir les deux règles suivantes pour prendre ce cas en compte :

$$[\mathit{trans}(a)]\,\frac{a \in \Sigma'_A \qquad \mathit{trans}(a) : q_A/\varphi \longrightarrow q'_A/\varphi'}{q_A, q_B/\varphi \longrightarrow q'_A, q_B/\varphi'}$$
$$[\mathit{trans}(a)]\,\frac{a \in \Sigma'_B \qquad \mathit{trans}(a) : q_B/\varphi \longrightarrow q'_B/\varphi'}{q_A, q_B/\varphi \longrightarrow q_A, q'_B/\varphi'}$$

Ces deux règles sont valides car les opérations effectuées sur φ dans chaque cas n'affectent que les clauses concernant les variables de l'automate dont on effectue une transition, les autres clauses restant inchangées conformément à nos attentes.

Pour le cas où une étiquette intervient dans les deux automates, on peut ensuite poser la règle suivante en utilisant le fait que K_A et K_B sont disjoints :

$$[\mathit{trans}(a)]\,\frac{a \in \Sigma' \qquad \mathit{trans}(a) : q_A/\varphi \longrightarrow q'_A/\varphi' \qquad \mathit{trans}(a) : q_B/\varphi' \longrightarrow q'_B/\varphi''}{q_A, q_B/\varphi \longrightarrow q'_A, q'_B/\varphi''}$$

car c'est uniquement lorsque K_A et K_B sont disjoints que l'on a la commutativité des opérateurs de projection et intersection. Dans la règle précédente, on peut en effet écrire

$$\varphi' = \mathit{proj}(\cdots \mathit{proj}(\varphi \wedge \chi_A, z_1^A) \cdots, z_k^A)$$
$$\varphi'' = \mathit{proj}(\cdots \mathit{proj}(\varphi' \wedge \chi_B, z_1^B) \cdots, z_l^B)$$

donc on a en composant

$$\varphi'' = \mathit{proj}(\cdots \mathit{proj}(\mathit{proj}(\cdots \mathit{proj}(\varphi \wedge \chi_A, z_1^A) \cdots, z_k^A) \wedge \chi_B, z_1^B) \cdots, z_k^B)$$

Or χ_B ne fait intervenir aucune des variables $z_1^A, \ldots, z_k^A$ puisque celles-ci sont dans K_A, donc comme l'opérateur *proj* ne modifie que les clauses faisant intervenir la variable selon laquelle se fait la projection, la formule χ_B n'est pas affectée par les projections selon les z_i^A. On a donc

$$\varphi'' = \mathit{proj}(\cdots \mathit{proj}(\mathit{proj}(\cdots \mathit{proj}(\varphi \wedge \chi_A \wedge \chi_B, z_1^A) \cdots, z_k^A), z_1^B) \cdots, z_k^B)$$

Par conséquent la règle $\mathit{trans}(a)$ dans ce cas renvoie bien toutes les réductions correspondant au passage d'une transition discrète étiquetée par a dans chaque membre.

5.3.3 Marches de l'automate produit

Étant donnés ces deux règles, construites à partir des règles caractéristiques des deux sous-automates, on a défini toutes les règles nécessaires à la formalisation en réécriture de l'automate produit. On peut donc énoncer le résultat suivant, conséquence directe des définitions que nous venons d'introduire :

proposition 11 (*validité du produit*)

> *Les règles de réécriture obtenues par cette méthode à partir des règles définissant les automates $\mathcal{A}$ et $\mathcal{B}$ sont équivalentes aux règles résultant de la transcription directe de l'automate $\mathcal{A} \,\|\, \mathcal{B}$.*

On notera bien l'importance de conserver les autres opérations indépendantes de l'automate, et en particulier la règle *time*. En effet il est nécessaire d'appliquer l'opérateur *time* sur la conjonction des formules de contrainte des deux sous-automates. Le cas contraire reviendrait en quelque sorte à faire passer le temps indépendemment dans les deux parties, ce qui ferait perdre la synchronisation qui est la raison d'être du produit d'automates.

6 Conclusion et ouvertures

On a donc établi une méthode permettant de créer, à partir d'un automate temporisé, un système de réécriture dont les réductions correspondent aux transitions de l'automate. On a également établi une opération sur les systèmes de réécriture obtenus correspondant à la notion de produit synchronisé sur les automates temporisés. Enfin, un début d'implémentation de ces règles a été réalisé à l'aide d'un outil générique de calcul de réécriture, montrant l'intérêt des résultats obtenus pour la manipulation effective d'automates temporisés.

6.1 Travaux restant à effectuer

La durée limitée du stage n'a pas permis d'atteindre tous les objectifs initiaux. Il reste en particulier à formaliser les quelques idées énoncées ici pour la méthode de résolution effective du problème d'atteignabilité, puis à implémenter cette méthode.

Il reste également à définir et à implémenter une méthode pour réaliser en ELAN la combinaison de systèmes de réécriture, afin de pouvoir réaliser effectivement des produits synchronisés.

6.2 Généralisations possibles

Certains points mériteraient d'être étudiés plus précisément. En particulier, il pourrait être intéressant de se demander quels types de contraintes peuvent être utilisés. Le langage des formules de contrainte utilisé ici est stable par les opérations que l'on veut effectuer sur lui, mais il serait intéressant de savoir s'il existe des langages plus vastes admettant cette propriété tout en conservant la décidabilité des problèmes d'atteignabilité.

D'autre part, il pourrait aussi être intéressant d'étudier d'autres formulations du problème en réécriture, par exemple en faisant apparaître plus explicitement la notion de passage du temps et de synchronisation. Ceci pourrait se faire en faisant apparaître l'état des horloges dans les termes manipulés, et en écrivant différemment les règles de réécriture.

Enfin, il y aurait probablement des résultats intéressants à tirer de la correspondance entre les deux formalismes étudiés. La notion de produit synchronisé et sa formalisation en réécriture laissent en effet penser qu'il s'agit d'un cas particulier d'une notion plus vaste de synchronisation.

A Implémentation en ELAN

Les méthodes établies ici ont donc conduit à une implémentation à l'aide du logiciel ELAN. Celui-ci, développé au Loria par l'équipe Prothéo, équipe d'accueil de ce stage, est un système générique pour le calcul de réécriture. On pourra trouver dans [BCD$^+$00] une description complète du système ELAN.

Le principe de réalisation d'un programme ELAN est de définir tout d'abord un certain nombre de *sortes*, que l'on peut voir comme les types des langages traditionnels, puis de définir des opérateurs servant à construire des termes de chaque sorte. On définit ensuite des règles de réécriture pour les termes de chaque sorte, puis des *stratégies* de réduction. Une stratégie est en quelque sorte une procédure décrivant comment les différentes règles sont appliquées. Il est possible par exemple de définir une stratégie consistant à appliquer une règle (ou une stratégie) le plus de fois possibles, ou au contraire de ne l'appliquer qu'une fois exactement.

Une caractéristique importante d'ELAN est qu'il manipule des systèmes de réécriture *non déterministes*. Il est possible en effet de définir une famille de règles pouvant produire plusieurs termes différents à partir d'un terme donné. On dispose alors de stratégies permettant de manipuler des ensembles de termes de plusieurs façons différentes.

Enfin, une notion fondamentale est celle de *spécification*. On définit généralement une système de réécriture de façon générique, en laissant certaines caractéristiques propres à un cas particulier définissables séparément, et une spécification est la définition de ces caractéristiques. Par exemple, dans le cas du calcul sur les contraintes, la spécification devra contenir la liste des symboles représentant les horloges, et la définition générique fournira la construction des termes à partir de ces symboles, ainsi que les règles de calcul car elles ne dépendent pas des symboles employées.

A.1 Calculs sur les contraintes

On fournit donc ici l'implémentation du module de calcul sur les contraintes temporelles des automates. La sorte *clockzone* est celle qui représente une contrainte. La spécification devra contenir la liste K des horloges.

Par rapport aux système exposé précédemment, on a ici ajouté un opérateur $inter(\_,\_)$ pour le calcul de l'intersection de deux zones, ceci afin d'éliminer lors de l'intersection les clauses superflues, pour alléger les termes sans effectuer une telle simplification après chaque opération.

```
/*
 * Manipulation des contraintes linéaires pour
 * les automates temporisés.
 *
 * (cette version ne suppose pas _ ∧ _ commutatif)
 *
 */
module ClockZone[K]

//---- MODULES IMPORTÉS ----//

import
 global
   K                  // les symboles pour les horloges

   identifier         // pour définir les symboles
   list[identifier]

   list[clock]        // pour la projection multiple

   int ;              // pour les constantes
end

//---- SORTES INTRODUITES ----//

sort
   clock        // une horloge
   clause       // une inégalité
   clockzone ;  // une conjontion d'inégalités
end

//---- CONSTRUCTEURS ----//

operators
 global
   // définition des horloges

   FOR EACH x : identifier SUCH THAT x := (listExtract) elem(K) :
   { x : clock ; }

   // définition des clauses

   true : clause ;
   false : clause ;

   @ < @ : ( clock int ) clause ;
   @ <= @ : ( clock int ) clause ;
   @ > @ : ( clock int ) clause ;
   @ >= @ : ( clock int ) clause ;

   @ − @ <= @ : ( clock clock int ) clause ;
```

```
    @ − @ <  @  :  ( clock clock int ) clause  ;

    // définition des conjonctions de clauses

    @       :  ( clause ) clockzone  ;
    @ ∧ @   :  ( clockzone clockzone ) clockzone  assocRight  ;
    [ @ ∧ @ ] :  ( clockzone clockzone ) clockzone  alias @ ∧ @ : ;

    // fonctions sur les zones

    empty( @ )  :  ( clockzone ) bool  ;

    quantify( @ , @ )  :  ( clockzone clock ) clockzone  ;

    inter( @ , @ )  :  ( clockzone clockzone ) clockzone  ;
    proj( @ , @ )   :  ( clockzone clock ) clockzone  ;
    time( @ )       :  ( clockzone ) clockzone  ;

    lproj( @ , @ )  :  ( clockzone list[clock] ) clockzone  ;
  local
    // termes intermédiaires pour l'insertion d'une clause

    min( @ , @ )  :  ( int int ) int  ;
    max( @ , @ )  :  ( int int ) int  ;

    ( @ and @ and @ )  :  ( clockzone clause clockzone ) clockzone  ;

    @ >> @  :  ( clockzone clockzone ) clockzone  ;

    // termes intermédiaire pour le découpage de conjonctions

    [@|@|@]           :  ( clockzone clockzone clockzone ) clockzone  ;
    . @ >>[@|@|@]     :  ( clockzone clockzone clockzone clockzone )
                                          clockzone  ;
    @ . @ >>[@|@|@]   :  ( clock clockzone clockzone clockzone clockzone )
                                          clockzone  ;

    second( @ )       :  ( clockzone ) clockzone  ;

    // termes pour la combinaison de clauses

    @ + @  :  ( clause clause ) clause  ;

    // pour la satisfiabilité

    emp( @ )  :  ( clockzone ) bool  ;
end

//---- STRATÉGIES ----//

stratop
  local
    ins     :  < clockzone → clockzone >  bs  ;

    div     :  < clockzone → clockzone >  bs  ;
    app     :  < clockzone → clockzone >  bs  ;

    empty   :  < bool → bool >  bs  ;
end

//---- RÈGLES DE RÉDUCTION ----//

//// combinaison de clauses

/*  Attention : il ne s'agit pas de règles génériques, mais de
 *  règles adaptées aux cas qui nous intéressent, c'est pourquoi
 *  x>5 + x<7 se réduit en true puisque cette combinaison doit
 *  être ignorée dans le cas de l'opérateur time, seul cas où
 *  cette combinaison apparait.
```

```
 */
rules for clause
   x, y, z : clock  ;
   c, d, e : int    ;
   t       : clause ;
 global
   [] x> c + x< d ⇒ t choose try where t:=()true if d> c
                                try where t:=()false if d<=c end end
   [] x> c + x<=d ⇒ t choose try where t:=()true if d> c
                                try where t:=()false if d<=c end end
   [] x>=c + x< d ⇒ t choose try where t:=()true if d> c
                                try where t:=()false if d<=c end end
   [] x>=c + x<=d ⇒ t choose try where t:=()true if d>=c
                                try where t:=()false if d< c end end

   [] x > c + y < d ⇒ y − x < e where e:=()d − c end
   [] x > c + y <= d ⇒ y − x < e where e:=()d − c end
   [] x >= c + y < d ⇒ y − x < e where e:=()d − c end
   [] x >= c + y <= d ⇒ y − x <= e where e:=()d − c end

   [] x > c + x − y < d ⇒ y > e where e:=()c − d end
   [] x > c + x − y <= d ⇒ y > e where e:=()c − d end
   [] x >= c + x − y < d ⇒ y > e where e:=()c − d end
   [] x >= c + x − y <= d ⇒ y >= e where e:=()c − d end

   [] x − y < c + y < d ⇒ x < e where e:=()c + d end
   [] x − y < c + y <= d ⇒ x < e where e:=()c + d end
   [] x − y <= c + y < d ⇒ x < e where e:=()c + d end
   [] x − y <= c + y <= d ⇒ x <= e where e:=()c + d end

   [] x − y < c + y − z < d ⇒ x − z < e where e:=()c + d end
   [] x − y < c + y − z <= d ⇒ x − z < e where e:=()c + d end
   [] x − y <= c + y − z < d ⇒ x − z < e where e:=()c + d end
   [] x − y <= c + y − z <= d ⇒ x − z <= e where e:=()c + d end
end

//// réduction des opérateurs

rules for int
   p, q : int ;
 global
   [] min(p,q) ⇒ p if p <= q end
   [] min(p,q) ⇒ q if p > q end

   [] max(p,q) ⇒ p if p >= q end
   [] max(p,q) ⇒ q if p < q end
end

rules for clockzone
   x, y       : clock       ;
   c, d       : int         ;
   t, tt      : clause      ;
   u, v, w, C : clockzone   ;
   lc         : list[clock] ;

 global

   // simplification des termes

   [] true ∧ u ⇒ u end

   [] false ∧ u ⇒ false end
   [] u ∧ false ⇒ false end
```

```
// il doit y avoir moyen de se passer de cette règle, puisque
// l'opérateur ∧ est déclaré associatif...
[assoc] [ [ u ∧ v ] ∧ w ]  ⇒  [ u ∧ [ v ∧ w ] ]  end

[] inter(t ∧ u, v)  ⇒  inter(u, w)
        where C  := (normalize(assoc)) v
        where w  := (ins) (true and t and C)  end
[] inter(t, v)  ⇒  w
        where C  := (normalize(assoc)) v
        where w  := (ins) (true and t and C)  end

[] quantify(C,x)  ⇒  v
        where u  := (div) x . C >> [true|true|true]
        where v  := (app) u
        end

[] proj(C,x)  ⇒  x>=0 ∧ x<=0 ∧ quantify(C,x)  end

[] lproj(C,x.lc)  ⇒  lproj(proj(C,x),lc)  end
[] lproj(C,nil)   ⇒  C                    end

[] time(C)  ⇒  second(u) ∧ v
        where u  := (repeat*(select)) . C >> [true|true|true]
        where v  := (app) u
        end

[] second([u|v|w])  ⇒  v  end

local

// insertion contrôlée d'une clause

[ins0]  (true and true and u)  ⇒  u      end
[ins0]  (u and t and false)    ⇒  false  end

[ins1]  (u and true and t ∧ v)  ⇒  (t ∧ u and true and v)  end

[ins1]  (u and x<=c and x<=d ∧ v)  ⇒
                    (x<=min(c,d) ∧ u and true and v)  end
[ins1]  (u and x< c and x< d ∧ v)  ⇒
                    (x< min(c,d) ∧ u and true and v)  end
[ins1]  (u and x<=c and x< d ∧ v)  ⇒  (t ∧ u and true and v)
                    choose try where t:=()x<=c  if c< d
                           try where t:=()x< d  if d<=c end  end
[ins1]  (u and x< c and x<=d ∧ v)  ⇒  (t ∧ u and true and v)
                    choose try where t:=()x< c  if c<=d
                           try where t:=()x<=d  if d< c end  end

[ins1]  (u and x>=c and x>=d ∧ v)  ⇒
                    (x>=max(c,d) ∧ u and true and v)  end
[ins1]  (u and x> c and x> d ∧ v)  ⇒
                    (x> max(c,d) ∧ u and true and v)  end
[ins1]  (u and x>=c and x> d ∧ v)  ⇒  (t ∧ u and true and v)
                    choose try where t:=()x>=c  if c> d
                           try where t:=()x> d  if d>=c end  end
[ins1]  (u and x> c and x>=d ∧ v)  ⇒  (t ∧ u and true and v)
                    choose try where t:=()x> c  if c>=d
                           try where t:=()x>=d  if d> c end  end

[ins1]  (u and x−y<=c and x−y<=d ∧ v)  ⇒
                    (x−y<=min(c,d) ∧ u and true and v)  end
[ins1]  (u and x−y< c and x−y< d ∧ v)  ⇒
                    (x−y< min(c,d) ∧ u and true and v)  end
[ins1]  (u and x−y<=c and x−y< d ∧ v)  ⇒  (t ∧ u and true and v)
                    choose try where t:=()x−y<=c  if c< d
```

```
                              try where t := ()x−y< d  if d<=c end  end
  [ins1]  (u and x−y< c and x−y<=d ∧ v)  ⇒  (t ∧ u and true and v)
                         choose try where t := ()x−y< c  if c<=d
                              try where t := ()x−y<=d  if d< c end  end

  [ins2]  (u and t and tt ∧ v)  ⇒  (tt ∧ u and t and v)  end

  [ins3]  (u and true and true)  ⇒  u      end
  [ins4]  (u and t    and true)  ⇒  t ∧ u  end

  // division des termes pour la projection

  [div1]  x .   x < c ∧ C >> [u|v|w]  ⇒  x . C >> [u|v| x < c ∧ w]  end
  [div1]  x .   x <=c ∧ C >> [u|v|w]  ⇒  x . C >> [u|v| x <=c ∧ w]  end
  [div1]  x . x−y< c ∧ C >> [u|v|w]  ⇒  x . C >> [u|v|x−y< c ∧ w]  end
  [div1]  x . x−y<=c ∧ C >> [u|v|w]  ⇒  x . C >> [u|v|x−y<=c ∧ w]  end

  [div1]  x .   x > c ∧ C >> [u|v|w]  ⇒  x . C >> [u| x > c ∧ v|w]  end
  [div1]  x .   x >=c ∧ C >> [u|v|w]  ⇒  x . C >> [u| x >=c ∧ v|w]  end
  [div1]  x . y−x< c ∧ C >> [u|v|w]  ⇒  x . C >> [u|y−x< c ∧ v|w]  end
  [div1]  x . y−x<=c ∧ C >> [u|v|w]  ⇒  x . C >> [u|y−x<=c ∧ v|w]  end

  [div2]  x . t ∧ C >> [u|v|w]  ⇒  x . C >> [t ∧ u|v|w]  end

  [div3]  x . true >> [u|v|w]  ⇒  [u|v|w]  end

  // division des termes pour time

  [select]  . x< c ∧ C >> [u|v|w]  ⇒  . C >> [u|v|x< c ∧ w]  end
  [select]  . x<=c ∧ C >> [u|v|w]  ⇒  . C >> [u|v|x<=c ∧ w]  end

  [select]  . x> c ∧ C >> [u|v|w]  ⇒  . C >> [u|x> c ∧ v|w]  end
  [select]  . x>=c ∧ C >> [u|v|w]  ⇒  . C >> [u|x>=c ∧ v|w]  end

  [select]  . x−y< c ∧ C >> [u|v|w]  ⇒  . C >> [x−y< c ∧ u|v|w]  end
  [select]  . x−y<=c ∧ C >> [u|v|w]  ⇒  . C >> [x−y<=c ∧ u|v|w]  end

  [select]  . true >> [u|v|w]  ⇒  [u|v|w]  end

  // combinaison des listes de clauses

  [appOne]  [u|t|tt ∧ v]  ⇒  [inter(t+tt, u)|t|v]  end
  [appOne]  [u|t|true]    ⇒  u                    end

  [appList]  [u|t ∧ v|w]  ⇒  [C|v|w]
             where C := (repeat*(appOne)) [u|t|w]  end
  [appList]  [u|true|v]   ⇒  u                     end

end

//// test de satisfiabilité

rules for bool
  x, y : clock     ;
  c    : int       ;
  u    : clockzone ;
  b    : bool      ;
 global
  [] empty(u)  ⇒  b
           where b := (empty) emp(u)  end

 local
  [empty1]  emp( x  <= c ∧ u)  ⇒  emp(quantify( x  <= c ∧ u, x))  end
  [empty1]  emp( x  <  c ∧ u)  ⇒  emp(quantify( x  <  c ∧ u, x))  end
  [empty1]  emp( x  >= c ∧ u)  ⇒  emp(quantify( x  >= c ∧ u, x))  end
  [empty1]  emp( x  >  c ∧ u)  ⇒  emp(quantify( x  >  c ∧ u, x))  end
  [empty1]  emp(x−y <= c ∧ u)  ⇒  emp(quantify(x−y <= c ∧ u, x))  end
  [empty1]  emp(x−y <  c ∧ u)  ⇒  emp(quantify(x−y <  c ∧ u, x))  end
```

```
  [empty2]  emp(false)  ⇒  true   end
  [empty2]  emp(true)   ⇒  false  end
end

//----  DÉFINITION DES STRATÉGIES  ----//

strategies for clockzone
 implicit
  [ ]  div  ⇒  repeat*(first one(div1, div2))  ;  div3  end
  [ ]  app  ⇒  repeat*(appList)                        end
  [ ]  ins  ⇒  first one(ins0,
                          repeat*(first one(ins1, ins2))  ;
                          first one(ins3, ins4))
                    end
end

strategies for bool
 implicit
  [ ]  empty  ⇒  repeat*(empty1)  ;  empty2  end
end

end
```

A.2 Transcription des automates

Le module qui suit est celui destiné au calcul sur les zones d'états d'un automate temporisé. La spécification doit ici contenir les caractéristiques de l'automate, à savoir l'ensemble K des horloges, l'ensemble Q des états et l'ensemble S des étiquettes (en tant que listes de symboles), ainsi que l'invariant *Inv* et la table de transition *Trans* sous la forme définie dans le module Terms fourni plus loin.

```
/*
 *   Transcription générique d'un automate
 */

module Automate[K,Q,S,Inv,Trans]

//----  MODULES IMPORTÉS  ----//

import
 global
  K Q S Inv Trans
  int
  bool
  identifier
  eq[state]
  Terms[K,Q,S]  ;
end


//----  CONSTRUCTEURS  ----//

operators
 global

  @ / @   :   ( state clockzone ) statezone  ;

  @ -- @   :   ( int statezone ) statezone  ;

  inv( @ )   :   ( state ) clockzone  ;
  trans . @ ( @ )   :   ( label statezone ) statezone  ;
```

```
    trans( @ )  :  ( statezone ) statezone ;
    time( @ )   :  ( statezone ) statezone ;

    reachability of @ from @  :  ( statezone statezone ) bool ;

  local
    // un petit hack pour tirer au sort une étiquette
    '-''-''-'  :  label ;

    st(@)  :  ( statezone ) state ;
    zn(@)  :  ( statezone ) clockzone ;

    tr_before( @ )  :  ( transition ) state ;
    tr_label( @ )   :  ( transition ) label ;
    tr_cond( @ )    :  ( transition ) clockzone ;
    tr_zero( @ )    :  ( transition ) list[clock] ;
    tr_after( @ )   :  ( transition ) state ;

    inv_state( @ )  :  ( invariant ) state ;
    inv_zone( @ )   :  ( invariant ) clockzone ;

end

stratop
  global
    trytrans :  < label → label >  bs ;
    allPaths :  < statezone → statezone >  bs ;
    tranz    :  < statezone → statezone >  bs ;
    dt       :  < statezone → statezone >  bs ;
end


//---- RÈGLES ----//

//// GÉNÉRIQUES

rules for statezone
    lbl        : label ;
    s          : state ;
    z, z1, z2  : clockzone ;
    sz, r      : statezone ;
    n          : int ;
  global

    [] trans.lbl(n -- sz)  ⇒  n−1 -- trans.lbl(sz)  if n > 0  end

    [trans]  sz   ⇒  r
             where lbl := (trytrans) ---
             where r := (dt) trans.lbl(sz)  end
    [time]   s / z  ⇒  s / z1
             where z1 := () inter(inv(s), time(z))  end

    [time]   n -- s / z  ⇒  n -- s / z2
             where z1 := () time(z)
             where z2 := () inter(inv(s), z1)  end

    [] trans(sz)  ⇒  r  where r := (tranz) sz  end
end

rules for clockzone
    s, t  : state ;
    z     : clockzone ;
    n     : int ;
    lbl   : label ;
    lc    : list[clock] ;
  global
```

```
    [] tr_cond(s,lbl:z,lc,t)  ⇒  z  end
    [] inv_zone(s:z)          ⇒  z  end

    [] zn(     s / z)  ⇒  z  end
    [] zn(n -- s / z)  ⇒  z  end
end

rules for state
    s, t  :  state  ;
    z     :  clockzone  ;
    n     :  int  ;
    lbl   :  label  ;
    lc    :  list[clock]  ;
  global
    [] tr_before(s,lbl:z,lc,t)  ⇒  s  end
    [] tr_after (s,lbl:z,lc,t)  ⇒  t  end
    [] inv_state(s:z)           ⇒  s  end

    [] st(     s / z)  ⇒  s  end
    [] st(n -- s / z)  ⇒  s  end
end

rules for label
    s, t  :  state  ;
    z     :  clockzone  ;
    n     :  int  ;
    lbl   :  label  ;
    lc    :  list[clock]  ;
  global
    [] tr_label(s,lbl:z,lc,t)  ⇒  lbl  end

    FOR EACH label  :  identifier SUCH THAT label:=(listExtract)elem(S)  :
    {
       [trylabel] lbl  ⇒  label  end
    }
end

rules for list[clock]
    s, t  :  state  ;
    z     :  clockzone  ;
    n     :  int  ;
    lbl   :  label  ;
    lc    :  list[clock]  ;
  global
    [] tr_zero(s,lbl:z,lc,t)  ⇒  lc  end
end

//// SPÉCIFIQUES À L'AUTOMATE

rules for statezone
    z  :  clockzone  ;
  global
    FOR EACH tr        :  transition  ;
             bef, aft  :  state  ;
             label     :  label  ;
             cond      :  clockzone  ;
             zero      :  list[clock]
    SUCH THAT tr  := (listExtract) elem(Trans)
            AND bef   :=()tr_before(tr)
            AND label:=()tr_label (tr)
            AND cond  :=()tr_cond  (tr)
            AND zero  :=()tr_zero  (tr)
```

```
            AND aft  :=()tr_after (tr) :
  {
    [DT]  trans.label(bef/z)  ⇒  aft / lproj(inter(cond,z),zero)  end
  }
end

rules for clockzone
 global
  FOR EACH invar : invariant ; st : state ; cz : clockzone
  SUCH THAT invar :=(listExtract)elem(Inv)
            AND st :=()inv_state(invar)  AND cz :=()inv_zone(invar) :
  {
    [ ]  inv(st)  ⇒  cz  end
  }
end

rules for bool
  sA, sB, s : state ;
  zA, zB, z : clockzone ;
  sz1, sz2  : statezone ;
 global
  [ ]  reachability of sA/zA from sz1  ⇒  true
            where sz2 := (allPaths) sz1
            where s := () st(sz2)
            if eq_state(sA, s) and not empty(inter(zA, zn(sz2)))  end
end


//--- STRATÉGIES ---//
strategies for label
 implicit
  [ ]  trytrans  ⇒  dk(trylabel)  end
end

strategies for statezone
 implicit
  [ ]  allPaths  ⇒  time ; iterate*(dk(tranz))  end
  [ ]  tranz  ⇒  trans ; time  end
  [ ]  dt  ⇒ dk(DT)  end
end

end
```

A.3 Spécification d'un automate

La spécification d'un automate se fait en utilisant une syntaxe particulière introduite dans le module Terms suivant. Ce module fournit également des règles pour l'extraction des données d'une spécification. La spécification pour ce module contient les listes K, Q et S qui définissent les éléments qui permettent de construire les termes nécessaires.

```
module Terms[K,Q,S]

import
 global
  K Q S
  int
  identifier
  list[identifier]
  list[invariant]
```

```
    list[transition]
    ClockZone[K] ;
end

sort
    invariant     // spécification d'un invariant
    transition    // spécification d'une transition
    label         // les étiquettes des états
    state         // les états discrets
    statezone ;   // les zones d'états temporels
end

operators
  global

    FOR EACH a : identifier SUCH THAT a := (listExtract) elem(S) :
    { a  :  label ; }
    FOR EACH q : identifier SUCH THAT q := (listExtract) elem(Q) :
    { q  :  state ; }
    @ ':' @  :  ( state clockzone ) invariant ;
    @ , @ ':' @ , @ , @  :  ( state label clockzone list[clock] state )
                                                    transition ;

end

end
```

Pour l'exemple de la figure 1, on aura donc la spécification suivante, selon la syntaxe ELAN :

```
specification simple
    Clocks
        X Y
        nil
    States
        s0 s1 s2
        nil
    Labels
        a b
        nil
    Invariant
        s0 : true
        s1 : X<5 ∧ true
        s2 : true
        nil
    Transitions
        s0, a : X< 2 ∧ true, Y nil, s1 .
        s0, b : X>=1 ∧ true,   nil, s2 .
        s1, a : Y< 1 ∧ true,   nil, s2 .
        s1, b :          true, Y nil, s1 .
        s2, a : Y> 2 ∧ true,   nil, s0 .
        nil
end
```

Références

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henziger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138 :3–34, 1995.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126 :183–235, 1994.

[Alu98] Rajeev Alur. Timed automata. *(?)*, 1998.

[BCD+00] Peter Borovanský, Horatiu Cirstea, Hubert Dubois, Claude Kirchner, Hélène Kirchner, Pierre-Étienne Moreau, Christophe Ringeissen, and Marian Vittek. ELAN *V 3.4 User Manual*. LORIA, January 2000.

[Dil89] David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Automatic verification methods for finite state systems*, volume 407 of *LNCS*, pages 197–212. Springer-Verlag, 1989.

[Sch86] A. Schrijver. *Theory of Linear And Integer Programming.* John Wiley & Sons, 1986.

Table des matières