Analyse en avant des automates temporisés

Pierre-Alain REYNIER

Mémoire de DEA, DEA Algorithmique 2003/2004,
supervisé par Patricia Bouyer et François Laroussinie.

Résumé

Les *contraintes temps-réel* sont de plus en plus importantes dans les systèmes informatiques actuels, leur vérification devient donc indispensable. Les automates temporisés constituent un modèle très adapté pour la vérification de tels systèmes et sont d'ailleurs implémentés dans différents outils. La plupart de ces model-checkers utilise un algorithme d'analyse en avant à la volée pour décider de problèmes d'accessibilité. Pourtant, il a été prouvé récemment que si cet algorithme est bien correct dans le cadre des automates temporisés sans garde diagonale, il est en fait incorrect dans le cadre plus général des automates temporisés avec gardes diagonales. Le but de ce stage était de comprendre le rôle des gardes diagonales et de proposer des algorithmes corrects pour vérifier ces automates temporisés.

Remerciements

Je tiens à remercier Patricia Bouyer et François Laroussinie, qui ont supervisé mon stage, pour le temps qu'ils ont pris pour répondre à toutes mes questions et pour travailler avec moi, et pour leur bonne humeur qui rend ce travail encore plus agréable.

Je tiens également à remercier l'ensemble des membres du L.S.V. (ENS de Cachan) où j'ai effectué mon stage pour leur sympathie qui rend ce laboratoire très convivial.

Table des matières

Introduction **5**

I Présentation du problème **9**

1 Automates temporisés et décidabilité **10**
1.1 Préliminaires 10
1.2 Présentation des automates temporisés 11
1.3 Résultats de décidabilité 12

2 L'algorithme d'analyse en avant **14**
2.1 Analyse à la volée 14
2.2 Abstraction 16

3 Correction de l'algorithme... **18**
3.1 L'automate contre-exemple 18
3.2 Résultats positifs 20
3.3 Automates temporisés avec ou sans gardes diagonales... 21

II Quelle approche du problème ? **23**

4 Choix de la méthode **24**
4.1 Premières pistes 24
4.2 Présentation de la méthode par raffinements successifs 27

5 Structure de données pour l'analyse de systèmes temporisés **30**
5.1 Zones et DBMs 30
5.2 Opérations utilisées par l'algorithme 32

6 Choix de l'outil **33**
6.1 Quelques outils existants 33
6.2 Notre choix : CMC 33
6.3 Détail des fonctionnalités implémentées 34

III Traitement du problème 35

7 Etude des contre-exemples erronés : premières constatations 36
7.1 Transition erronée . . . 36
7.1.1 Premières notations et définitions . . . 36
7.1.2 La transition erronée n'est pas toujours diagonale . . . 37
7.2 Transition instable initiale . . . 37
7.2.1 Définitions . . . 37
7.2.2 La transition instable initiale n'est pas toujours diagonale . . . 39
7.3 Quand les gardes diagonales se cachent... . . . 39

8 Deux méthodes pour choisir les gardes diagonales en cause 41
8.1 Critère pour la dédiagonalisation . . . 41
8.1.1 Définition . . . 41
8.1.2 Cas remarquables . . . 42
8.1.3 Choix des gardes diagonales . . . 44
8.2 Pistage des erreurs . . . 46
8.2.1 Exemples et définitions . . . 46
8.2.2 Étude de l'algorithme de Floyd et historique des erreurs . . . 48
8.2.3 Méthode de choix . . . 49

9 Récapitulatif des algorithmes proposés 51
9.1 Algorithmes basés sur la méthode naïve . . . 51
9.2 Algorithmes basés sur la méthode par raffinements successifs . . . 53

Conclusion 55

Bibliographie 58

A Calculs de CMC 59

Introduction

De nos jours, l'informatique prend une place prépondérante dans une grande et croissante variété d'applications. Nous pouvons citer les transports (avionique, ligne Météor, systèmes de guidage), les télécommunications (téléphones portables avec systèmes de cryptages, Internet), la finance (systèmes sécurisés de paiement sur Internet) et la recherche spatiale (lancement de fusées, de satellites). Toutes ces applications ont des enjeux essentiels, comme la vie de nombreuses personnes pour les transports, ou des sommes monétaires immenses pour les paiements en ligne ou la recherche spatiale. Aussi nous attendons de ces systèmes d'être irréprochables. Pourtant, la liste des échecs retentissants de tels systèmes est malheureusement très longue (exposition à de trop fortes radiations dûes au système Thérac 25, crash du système téléphonique AT&T, crash d'Ariane 5,...).

Aussi, certaines de ces applications doivent désormais répondre à des normes, commes il en existe concernant des produits alimentaires ou des réalisations du génie civil. Ainsi, les protocoles de télécommunications doivent être testés avant d'être mis sur le marché. Cette phase de test a toujours existé, mais n'a pas toujours été suffisante. En effet ces tests se limitent généralement à des scénarios attendus par les ingénieurs qui réalisent le système. La batterie de telles expériences pourra être toujours plus grande, elle ne sera sûrement jamais exhaustive.

La vérification formelle a pour but de décider si un système vérifie ou non une propriété, si possible de façon entièrement automatique. Pour cela, le système doit d'abord être traduit dans un formalisme qui permet de modéliser des systèmes réels. Puis la propriété doit elle aussi être traduite dans une logique ou un langage permettant de caractériser des comportements du système. Enfin il s'agit de développer des algorithmes décidant si une propriété donnée exprimée dans une certaine logique est vérifiée ou non par un système décrit dans un certain formalisme.

Selon le type de systèmes étudiés, et le type de propriétés que l'on souhaite tester, le modèle et la logique utilisés ne seront pas les mêmes. Nous étudions dans ce stage un modèle prenant en compte le temps réel : *les automates temporisés*. Les automates d'états finis sont des systèmes de transitions à états finis qui permettent d'étudier l'ordre relatif des actions comme par exemple "se réveiller" avant de pouvoir faire l'action "se lever" (à moins d'être somnanbule). Mais ce formalisme ne permet pas de tester si on a mis moins de 5 minutes à se lever, une fois réveillé. Les automates temporisés sont des automates finis enrichis d'horloges qui vont donner des informations quantitatives sur l'écoulement du temps.

Les automates temporisés ont été introduits en 1990 par Alur et Dill [AD90]. Depuis, ils ont été beaucoup étudiés d'un point de vue théorique, et plusieurs outils réalisent une implémentation des résultats théoriques ainsi développés. Certains de ces outils ont même été utilisés lors d'études de systèmes réels afin de déceler (avec succès) des erreurs de conception. Citons par exemple la correction d'une erreur dans un protocole de contrôle audio/video de Bang & Olufsen par l'outil UPPAAL [HSLL97].

Le problème de base de la vérification est celui de l'accessibilité. En effet, cela revient à tester des propriétés de sûreté qui constituent la plupart des propriétés réellement étudiées. Un algorithme d'analyse en avant est largement utilisé pour décider ce problème. Cet algorithme, bien qu'utilisé avec succès par le passé, a été remis en cause récemment par Patricia Bouyer [Bou04]. Elle a en effet montré qu'il ne pouvait être correct pour la classe générale des automates temporisés avec gardes diagonales car il calculait en fait une surapproximation des états accessibles. Cependant, l'algorithme est correct pour la classe des automates temporisés sans garde diagonale, ce qui constitue la classe la plus couramment utilisée. Le but de ce stage consiste à comprendre le rôle des gardes diagonales dans ce "bug" et à proposer des modifications à apporter à l'algorithme classique d'analyse en avant de sorte à ce qu'il soit correct pour les automates temporisés ayant des gardes diagonales.

L'idée qui nous a guidé dans ce stage était basée sur la remarque suivante : il existe une construction permettant, étant donné un automate temporisé avec gardes diagonales, de construire un automate temporisé reconnaisssant le même langage sans garde diagonale. Nous pourrions penser qu'alors le problème est résolu, mais cette construction est malheureusement coûteuse du point de vue de la complexité puisqu'elle est exponentielle en le nombre de gardes diagonales. L'idée consiste donc à éliminer par cette construction un nombre minimum de gardes diagonales de sorte à essayer d'éviter une explosion de la taille de la structure. Pour cela, nous chercherons à déterminer les gardes diagonales rendant incorrect l'algorithme classique en étudiant les erreurs commises par ce dernier. Ce principe porte le nom de raffinement successif basé sur l'étude de contre-exemples erronés.

Le présent mémoire s'organisera comme suit. Dans une première partie, nous rappellerons les bases théoriques nécessaires afin de présenter le plus précisément possible le problème qui nous intéressera. Ensuite, nous justifierons notre approche du problème en montrant pourquoi des méthodes plus simples ne peuvent pas s'appliquer. Nous présenterons aussi l'outil utilisé pour les expérimentations pratiques. Pour finir, nous présenterons l'étude des contre-exemples que nous avons menée. Nous récapitulerons alors les algorithmes proposés pour répondre à notre problème.

Première partie

Présentation du problème

Chapitre 1

Automates temporisés et décidabilité

Dans ce premier chapitre, nous allons présenter le modèle des automates temporisés, introduit par Alur et Dill [AD90], [AD94]. Ce modèle est utilisé en vérification afin de tenir compte de propriétés liées à l'écoulement du temps.

1.1 Préliminaires

Afin d'ajouter au modèle des automates finis la notion de temps, nous allons utiliser des variables appelées horloges.

Domaine de temps. Nous considérerons comme *domaine de temps* $\mathbb{T}$ l'ensemble $\mathbb{Q}^+$ des nombres rationnels positifs ou l'ensemble $\mathbb{R}^+$ des nombres réels. Σ désigne un ensemble fini d'*actions*. Une *séquence de dates* sur $\mathbb{T}$ est une séquence croissante finie $\tau = (t_i)_{1\leq i\leq p} \in \mathbb{T}^*$. Un *mot temporisé* $w = (a_i, t_i)_{1\leq i\leq p}$ est un élément de $(\Sigma \times \mathbb{T})^*$, où $\sigma = (a_i)_{1\leq i\leq p}$ est un mot de Σ^* et $\tau = (t_i)_{1\leq i\leq p}$ une séquence de dates sur $\mathbb{T}$ de même longueur. La date t_i correspond intuitivement à l'instant auquel l'action a_i a eu lieu.

Valuations d'horloges. Nous considérons un ensemble X d'horloges. Une *valuation d'horloges* sur X est une application $v : X \to \mathbb{T}$ qui assigne à chaque horloge une valeur de temps. L'ensemble des valuations d'horloges sur X est noté $\mathbb{T}^X$. Deux opérations sont possibles sur ces valuations : laisser s'écouler du temps (les horloges augmentent toutes à la même vitesse), et remettre certaines horloges à zéro. Soit $t \in \mathbb{T}$, la valuation $v + t$ est définie par $(v + t)(x) = v(x) + t, \forall x \in X$. Pour un sous-ensemble C de X, on note $[C \leftarrow 0]v$ la valuation définie par $([C \leftarrow 0]v)(x) = \begin{cases} 0 \text{ si } x \in C, \\ v(x) \text{ si } x \in X \setminus C. \end{cases}$

Contraintes d'horlorges. Étant donné un ensemble d'horloges X, nous introduisons deux ensembles de contraintes d'horloges sur X.
Le plus général des deux, noté $\mathcal{C}(X)$, est défini par la grammaire suivante :

$\varphi ::= x \sim c \mid x - y \sim c \mid \varphi \wedge \varphi \mid \mathit{true}$ où $x, y \in X, c \in \mathbb{Z}, \sim \in \{<, \leq, =, \geq, >\}$.

Nous utiliserons également un deuxième ensemble, constitué exclusivement de *contraintes non-diagonales*, noté $\mathcal{C}_{nd}(X)$, défini par la grammaire :

$\varphi ::= x \sim c \mid \varphi \wedge \varphi \mid \mathit{true}$ où $x, y \in X, c \in \mathbb{Z}, \sim \in \{<, \leq, =, \geq, >\}$.

Remarquons que dans le modèle initial proposé par Alur et Dill, seules les contraintes non diagonales étaient utilisées.

Une valuation v satisfait une contrainte simple $x \sim c$ si $v(x) \sim c$. Nous noterons dans ce cas $v \models x \sim c$. Nous étendons naturellement cette définition à toute contrainte d'horloge φ.
Si k désigne un nombre entier, une *contrainte d'horloges k-bornée* est une contrainte d'horloges n'impliquant que des constantes comprises entre $-k$ et k.

1.2 Présentation des automates temporisés

Nous pouvons à présent donner la définition formelle d'un automate temporisé et de son comportement.

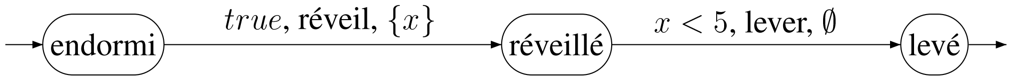
Automates temporisés.
Un *automate temporisé* sur $\mathbb{T}$ est un 6-uplet $\mathcal{A}= (\Sigma, Q, T, I, F, X)$ où :
- Σ est un alphabet fini d'actions,
- Q est un ensemble fini d'états (de contrôle),
- X est un ensemble fini d'horloges,
- $T \subseteq Q \times \mathcal{C}(X) \times \Sigma \times 2^X \times Q$ est un ensemble fini de transitions,
- $I \subseteq Q$ est l'ensemble des états initiaux et
- $F \subseteq Q$ est l'ensemble des états finaux.

Comportement d'un automate temporisé.
Un *chemin* dans $\mathcal{A}$ est une séquence finie de transitions consécutives :
$P = q_0 \xrightarrow{\varphi_1, a_1, C_1} q_1 \ldots q_{p-1} \xrightarrow{\varphi_p, a_p, C_p} q_p$ où, pour chaque $1 \leq i \leq p$, $(q_{i-1}, \varphi_i, a_i, C_i, q_i) \in T$.
Un tel chemin est dit *acceptant* s'il part d'un état initial ($q_0 \in I$) et se termine dans un état final ($q_p \in F$).
Une *configuration* de l'automate temporisé $\mathcal{A}$ est un couple $\langle q, v\rangle \in Q \times \mathbb{T}^X$.
Une exécution de l'automate suivant le chemin P est une séquence de la forme :
$\langle q_0, v_0\rangle \xrightarrow[t_1]{\varphi_1, a_1, C_1} \langle q_1, v_1\rangle \ldots \langle q_{p-1}, v_{p-1}\rangle \xrightarrow[t_p]{\varphi_p, a_p, C_p} \langle q_p, v_p\rangle$ où $\tau = (t_i)_{1 \leq i \leq p} \in \mathbb{T}^*$ est une séquence de dates et $(v_i)_{1 \leq i \leq p}$ un p-uplet de valuations d'horloges définies par :

$$\begin{cases} v_0(x) = 0, \forall x \in X \\ v_{i-1} + (t_i - t_{i-1}) \models \varphi_i \\ v_i = [C_i \leftarrow 0](v_{i-1} + (t_i - t_{i-1})) \end{cases} \qquad \text{avec la convention que } t_0 = 0.$$

L'étiquette de cette exécution est le mot temporisé $w = (a_1, t_1) \ldots (a_p, t_p)$. De plus, si le chemin P est acceptant, alors nous dirons que le mot w est accepté par l'automate temporisé $\mathcal{A}$. L'ensemble des mots temporisés acceptés par $\mathcal{A}$ est noté $\mathcal{L}(\mathcal{A})$ et appelé langage accepté par $\mathcal{A}$.

Un exemple.



Le langage accepté par cet automate est l'ensemble des mots temporisés $w = (\text{réveil}, t)(\text{lever}, t+x)$ avec $t \geq 0$ et $0 \leq x < 5$.

1.3 Résultats de décidabilité

Problèmes étudiés. Dans le domaine de la vérification, une question fondamentale concernant les automates temporisés consiste à déterminer si le langage accepté est vide. Ce problème est appelé le *problème du vide*. Une classe d'automates temporisés est dite *décidable* si le problème du vide est décidable pour tout élément de cette classe. Nous nous intéresserons dans la suite au *problème d'accessibilité discrète* qui consiste à décider si un état de contrôle de l'automate temporisé est accessible. Notons que ces deux problèmes (vacuité du langage et accessibilité discrète) sont en fait équivalents.
Du point de vue de la vérification, le problème d'accessibilité est un problème de base. D'une part parce que les autres problèmes sont en général permet de répondre à des propriétés de sûreté : garantir que le système n'entrera jamais dans un certain ensemble d'états d'erreur.
Convention importante : Par la suite, nous nous intéresserons exclusivement au problème d'accessibilité vu comme un problème de sûreté, i.e. que nous identifierons états finaux et états d'erreurs que le système doit éviter.

Automate des régions. Alur et Dill ont démontré [AD94] que le problème du vide est décidable pour les automates temporisés. Leur preuve est basée sur la construction de l'*automate des régions*. Celui-ci consiste en fait en un graphe fini simulant l'automate temporisé basé sur une relation d'équivalence d'indice fini définie sur les valuations d'horloges. Il s'agit d'identifier deux valuations v et v' telles que le comportement de l'automate temporisé soit le même à partir des configurations $\langle q, v\rangle$ et $\langle q, v'\rangle$ pour tout état q de l'automate. Le graphe quotient de cette relation d'équivalence est appelé *graphe des régions*. C'est simplement une partition finie de l'ensemble $\mathbb{T}^X$ des valuations d'horloges possibles, pour laquelle chaque élément est dénommé "région".

Nous allons faire quelques remarques élémentaires sur les propriétés que doit vérifier cette partition. Ces remarques sont illustrées sur des exemples présentés en figure 1.

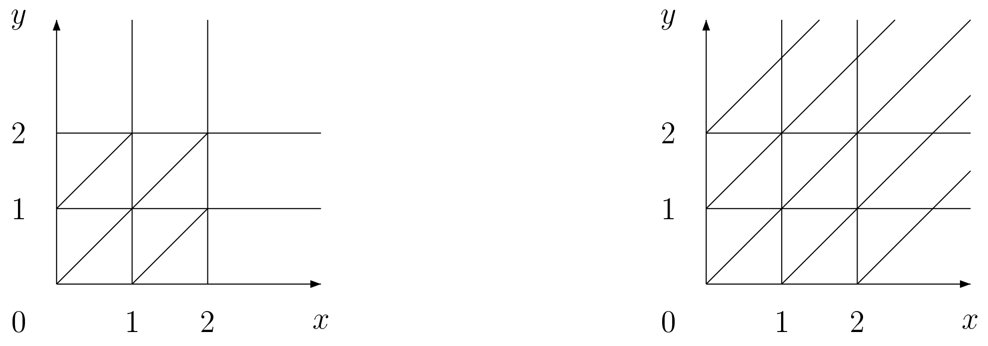


FIG. 1 – Graphes des régions associés à un automate temporisé ayant deux horloges x et y et 2 pour constante maximale. A gauche, sans garde diagonale, à droite, avec.

Compatibilité entre régions et contraintes. Deux valuations d'une même région doivent satisfaire les mêmes contraintes. (découpe selon les contraintes)

Compatibilité entre régions et écoulement du temps. Toutes les valuations d'une même région doivent avoir le même successeur immédiat, i.e. qu'en laissant écouler le temps, les valuations doivent atteindre en premier la même région. (découpe selon les diagonales)

Résultats. La taille de l'automate des régions ainsi défini est exponentielle en la taille de l'automate temporisé. Comme l'accessibilité dans les automates finis est un problème NlogSPACE-facile, on obtient que le problème de l'accessibilité dans les automates temporisés est PSPACE-facile. C'est même un problème PSPACE-complet [AL02].

Chapitre 2

L'algorithme d'analyse en avant

La construction présentée dans la partie précédente étant trop coûteuse, d'autres méthodes sont utilisées pour décider le problème d'accessibilité. Celles-ci sont basées sur des techniques de construction à la volée afin d'éviter de construire l'ensemble des configurations. De plus, les ensembles de configurations seront représentés de manière symbolique.

2.1 Analyse à la volée

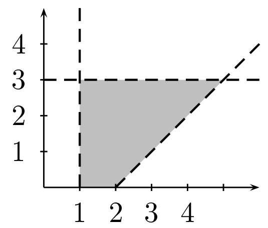
Il existe principalement deux algorithmes d'accessibilité à la volée : la recherche en avant, et la recherche en arrière. La recherche en avant (resp. en arrière) consiste à calculer tous les successeurs (resp. prédécesseurs) des configurations initiales (resp. finales). Rappelons que $\langle q, v\rangle$ est une configuration initiale si $q \in I$ et $v(x) = 0$ pour tout $x \in X$, et finale si $q \in F$.

L'algorithme d'analyse en arrière a de bonnes propriétés (terminaison, correction) mais souffre d'un défaut important : en pratique, les systèmes modélisés par des automates temporisés manipulent également des entiers, sur lesquels ils peuvent faire les opérations arithmétiques classiques ($+, -, \times, \div$). Ces opérations sont aisément réalisables dans un calcul en avant, mais au contraire elles sont difficiles à inverser : sachant qu'un entier $k = i \times j$ décrit un intervalle $[a, b]$, comment décrire les intervalles décrits par i et j ? Cette remarque justifie le fait que **nous nous intéresserons désormais exclusivement à l'analyse à la volée en avant**.

Les horloges rendant l'ensemble des configurations infini, l'algorithme utilise une représentation symbolique d'ensembles infinis de valuations : les zones.

Zones. Une zone est un ensemble de valuations défini par une contrainte d'horloges : la zone associée à la contrainte φ est $\{v \in \mathbb{T}^X \mid v \models \varphi\}$.

Exemple. La zone associée à la contrainte $\varphi = x > 1 \wedge y \leq 3 \wedge x - y < 2$ est illustrée sur la figure ci-contre.



Nous pouvons à présent décrire plus précisément le fonctionnement de l'algorithme de calcul en avant (algorithme 1). Il consiste simplement, en partant de l'état initial (nous supposons qu'il est unique) q_0 de l'automate, et de la zone initiale Z_0 réduite à la seule valuation qui associe 0 à chaque horloge, à calculer les successeurs en un pas, selon l'opérateur $Post$, puis à itérer cette opération jusqu'à obtenir la stabilisation ou la découverte d'un état final.

Algorithme 1 Analyse en avant à la volée.

Entrée : $\mathcal{A}$ un automate temporisé
Sortie : $\mathcal{L}(\mathcal{A}) = \emptyset$?
  Visités := $\emptyset$;
  En_Attente := $\{(q_0, Z_0)\}$;
  Répéter
    Piocher (q, Z) dans En_Attente ;
    Si q est un état final **Alors**
      Retourner "Un état final est accessible." ;
    Sinon
      Si il n'existe pas $(q, Z') \in$ Visités tel que $Z \subseteq Z'$ **Alors**
        Visités := Visités $\cup \{(q, Z)\}$;
        Successeurs := $\{(q', Post(Z, e)) \mid e$ transition de q à $q'\}$;
        En_Attente := En_Attente $\cup$ Successeurs ;
      Fin Si
    Fin Si
  Jusqu'à En_Attente $= \emptyset$;
  Retourner "Aucun état final n'est accessible." ;

L'opérateur *Post*. Deux types d'évolutions sont possibles dans un automate temporisé : laisser écouler du temps en restant dans l'état courant et franchir une transition. Considérons un couple (q, Z) (q est un état et Z une zone), et une transition e de cet état q vers un autre état q'. $Post(Z, e)$ retourne l'ensemble des valuations d'horloges accessibles à partir de celles appartenant à Z en laissant écouler du temps dans q, puis en franchissant e. Plus précisément :

$$Post(Z, e) = \{v' \mid \exists(v, t) \in (Z \times \mathbb{T}) \mid \langle q, v + t\rangle \xrightarrow{e} \langle q', v'\rangle\}$$

Ce calcul peut se décomposer en plusieurs étapes : notons $e = (q \xrightarrow{g,a,C} q')$, alors nous avons $Post(Z, e) = [C \leftarrow 0](g \cap \overrightarrow{Z})$ où $\overrightarrow{Z}$ désigne le *futur* de Z, i.e. $\overrightarrow{Z} = \{v + t \mid v \in Z$ et $t \in \mathbb{T}\}$.

Les opérations à réaliser sur les zones sont donc les suivantes : calcul du futur d'une zone, de l'instersection de deux zones, et remise à zéro au sein d'une zone de certaines horloges. Ces opérations, qui sont stables pour les zones (l'image d'une zone est une zone), sont illustrées sur le figure 2.

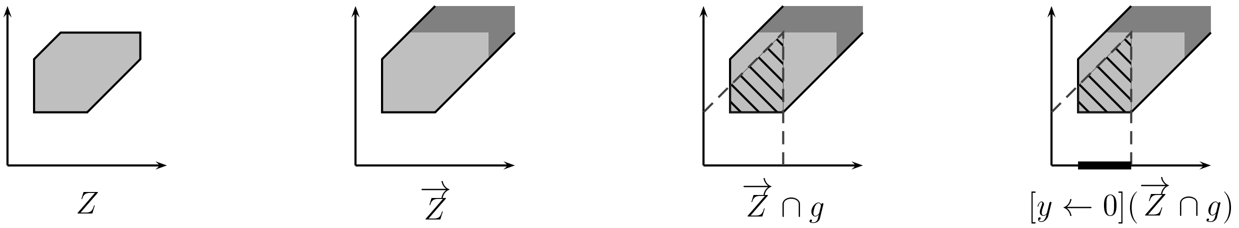
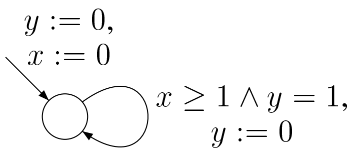


FIG. 2 – Détail de l'application de l'opérateur $Post$.

2.2 Abstraction

Bien que l'utilisation des zones permet une représentation symbolique d'ensembles de valuations, le nombre d'objets manipulés par l'algorithme reste infini. L'algorithme risque donc de ne pas terminer. Cela est en fait possible, comme le montre le contre-exemple représenté en figure 3. En effet, en partant de la zone initiale $(x = y = 0)$, nous obtenons l'ensemble des zones de la forme $(y \geq 0 \wedge x = y + n)$, pour n entier naturel.



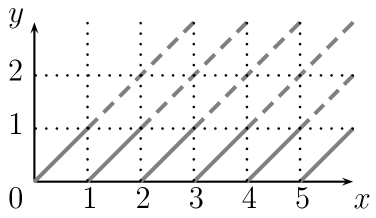


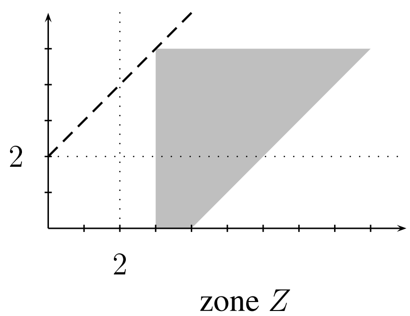
FIG. 3 – Un automate temporisé dont le calcul en avant ne termine pas.

Afin d'assurer la terminaison de l'algorithme, une idée naturelle consiste à limiter les calculs à un nombre fini de zones. Nous pouvons donc utiliser une abstraction identifiant certaines zones. Pour cela, nous nous inspirons de la construction du graphe des régions qui tient essentiellement compte des valeurs inférieures à la constante maximale du système.

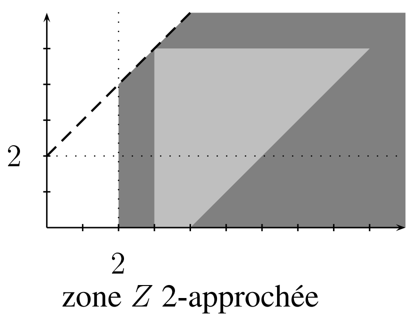
Zone k-bornée et k-approximation.
Une zone Z est dite *k-bornée* si elle est définie par une contrainte d'horloges qui est k-bornée. La *k-approximation* de Z est alors définie par :

$$\mathrm{Approx}_k(Z) = \bigcap_{\text{zones } k\text{-bornées } Z' \mid Z \subseteq Z'} Z'$$

La k-approximation est bien définie ainsi. En effet, l'ensemble des zones k-bornées est fini (il a été construit pour cela). A fortiori, l'ensemble des zones k-bornées contenant Z est fini lui aussi. De plus, ce dernier ensemble est non vide car la zone $\mathbb{T}^n$ est k-bornée et contient Z. L'intersection définie plus haut existe donc. De plus, elle constitue une zone k-bornée et contient Z, et est donc la plus petite zone, au sens de l'inclusion, ayant cette propriété.



zone Z



zone Z 2-approchée

FIG. 4 – 2-approximation d'une zone

Le nouvel algorithme obtenu en utilisant cette abstraction est l'algorithme 2. Il consiste simplement, après chaque application de *Post*, à appliquer l'abstraction de sorte que les éléments stockés dans les listes "Visités" et "En_Attente" soient en nombre fini. Remarquons que dans notre présentation de l'algorithme, nous avons placé la constante k d'extrapolation en entrée, mais dans les outils, celle-ci est définie à partir de l'automate temporisé $\mathcal{A}$ pris en entrée, par exemple en prenant la constante maximale apparaissant dans $\mathcal{A}$.

Algorithme 2 Analyse en avant à la volée utilisant la k-approximation

Entrée : $\mathcal{A}$ un automate temporisé et k la constante d'extrapolation
Sortie : $\mathcal{L}(\mathcal{A}) = \emptyset$?

```
Visités := ∅ ;
En_Attente := {(q0, Approx_k(Z0))} ;
Répéter
   Piocher (q, Z) dans En_Attente ;
   Si q est un état final Alors
      Retourner "Un état final est accessible." ;
   Sinon
      Si il n'existe pas (q, Z') ∈ Visités tel que Z ⊆ Z' Alors
         Visités := Visités ∪ {(q, Z)} ;
         Successeurs := {(q', Approx_k(Post(Z, e))) | e transition de q à q'} ;
         En_Attente := En_Attente ∪ Successeurs ;
      Fin Si
   Fin Si
Jusqu'à En_Attente = ∅ ;
Retourner "Aucun état final n'est accessible." ;
```

Pour valider un algorithme d'accessibilité, il y a essentiellement quatre propriétés à vérifier :

- la terminaison,
- la complétude (tout état accessible peut être déclaré comme tel par l'algorithme),
- la correction (tout état déclaré accessible par l'algorithme l'est réellement),
- l'implémentabilité.

L'abstraction utilisée a été construite pour assurer la première propriété.
La seconde propriété est elle aussi vérifiée car, comme nous l'avons vu, toute zone est incluse dans son image par Approx_k, et l'algorithme réalise donc une surapproximation de l'ensemble des états accessibles.
Le troisième point va être discuté en détail dans le chapitre suivant. Il constitue en fait le point critique.
Enfin, notons que la notion d'implémentabilité est assez subjective. Nous verrons cependant dans le chapitre 5 une structure de données adaptée permettant d'implémenter les zones, et dans le chapitre 6 nous présenterons différents outils utilisant cet algorithme.

Chapitre 3

Correction de l'algorithme...

L'algorithme 1 était bien correct puisqu'il réalisait un calcul exact. En revanche, nous avons vu qu'il ne terminait pas toujours. Nous avons donc introduit l'abstraction Approx_k qui assure la terminaison de l'algorithme 2. Il s'agit donc à présent de s'interroger sur la correction de cet algorithme. C'est précisément l'étude menée dans [Bou04]. Nous pouvons déjà remarquer que l'algorithme réalise à présent une surapproximation. En effet, par définition, nous avons $Z \subseteq \text{Approx}_k(Z)$ pour toute zone Z. Donc si un état q est accessible, l'algorithme va bien le déclarer accessible. En revanche, l'inverse n'est a priori pas forcément vrai, i.e. l'algorithme pourrait déclarer accessible un état qui ne l'est pas.

Alors que cet algorithme est implémenté dans différents outils et même utilisé dans des études de cas réels, il n'existait en fait pas de preuve complète de sa correction. Or nous allons voir qu'en réalité l'algorithme n'est pas correct dans le cas général des automates temporisés avec gardes diagonales.

3.1 L'automate contre-exemple

Nous allons étudier à présent l'automate temporisé $\mathcal{C}$ présenté ci-dessous et nous allons voir que l'algorithme 2 donne une réponse incorrecte sur cette entrée.

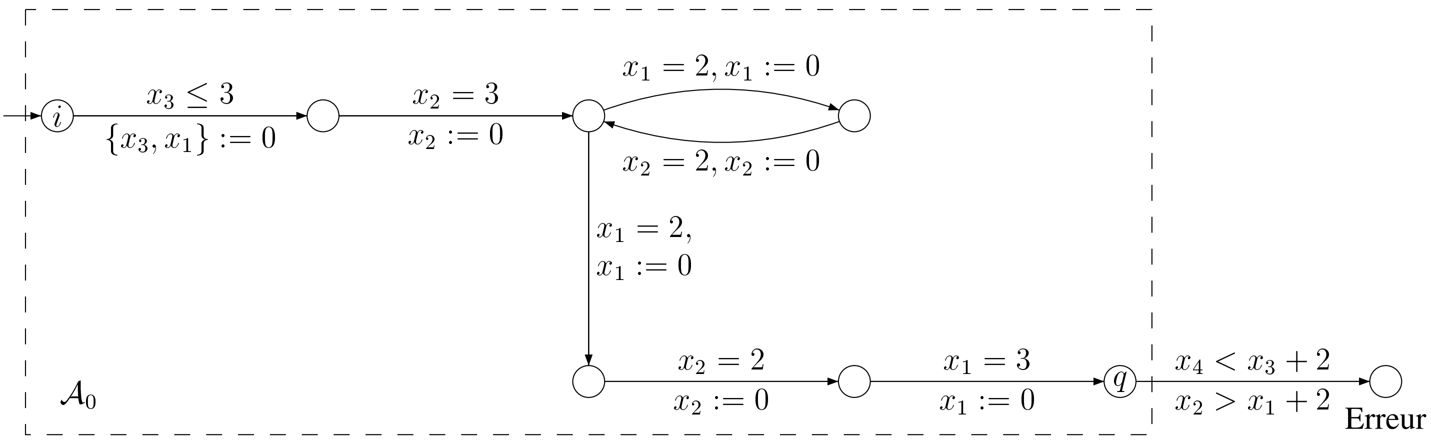


FIG. 5 – Un automate temporisé, $\mathcal{C}$

Notation. Nous notons $\mathcal{A}_0$ la partie de $\mathcal{C}$ encadrée.

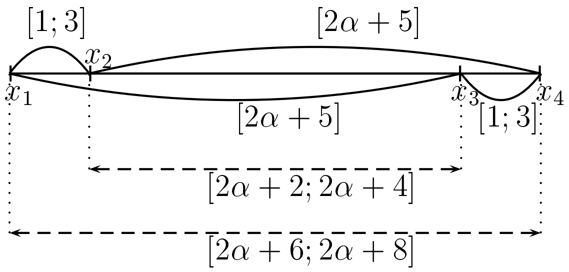


FIG. 6 – Les zones Z_α

Considérons un chemin de i à q dans l'automate $\mathcal{C}$. Si d est la date à laquelle est franchie la première transition, et si α est le nombre de fois que la boucle est prise sur ce chemin, alors la valuation v d'horloges en arrivant en q est définie par :

$$\begin{array}{ll} v(x_1) = 0 & \qquad v(x_3) = 2\alpha + 5 \\ v(x_2) = d & \qquad v(x_4) = 2\alpha + 5 + d \end{array}$$

Alors, en appliquant un calcul exact des successeurs de l'état initial, avec toutes les horloges mises à 0, l'ensemble des valuations qui peuvent être atteintes en q, quand la boucle est prise α fois, est défini par les relations (1) de la table 1. Cet ensemble constitue bien sûr une zone et est noté Z_α. Il est représenté en figure 6.

$$(1) \begin{cases} x_2 \geq 1 \\ x_3 \geq 2\alpha + 5 \\ x_4 \geq 2\alpha + 6 \\ 1 \leq x_2 - x_1 \leq 3 \\ 1 \leq x_4 - x_3 \leq 3 \\ x_3 - x_1 = 2\alpha + 5 \\ x_4 - x_2 = 2\alpha + 5 \end{cases} \qquad (2) \begin{cases} x_2 \geq 1 \\ x_3 > k \\ x_4 > k \\ 1 \leq x_2 - x_1 \leq 3 \\ 1 \leq x_4 - x_3 \leq 3 \\ x_3 - x_1 > k \\ x_4 - x_2 > k \end{cases}$$

TAB. 1 – Les équations des zones Z_α et $\mathrm{Approx}_k(Z_\alpha)$

Bien que ce ne soit pas explicite dans la description de Z_α par les équations (1), nous pouvons aisément déduire, et cela est très visible sur la représentation des zones de la figure 6, que si v est une valuation de Z_α, alors elle vérifie la contrainte très forte suivante :

$$v(x_4) - v(x_3) = v(x_2) - v(x_1) \tag{3}$$

En particulier, nous en déduisons que l'état Erreur n'est pas accessible.
Cependant, comme la condition (3) n'est pas expicite dans les équations (1), celle-ci va disparaître lorsque l'opérateur Approx_k va être appliqué. Plus précisément, soit k la constante d'extrapolation fixée, et soit α tel que $2\alpha + 2 > k$. Appliquons alors l'opérateur Approx_k à Z_α, la zone $\mathrm{Approx}_k(Z_\alpha)$ obtenue est définie par les équations (2) de la table 1. Remarquons que nous appliquons ici l'opérateur d'abstraction seulement à la zone calculée pour l'état q, et non

sur l'ensemble du chemin. Cela est en fait suffisant car la zone $\text{Approx}_k(Z_\alpha)$ est incluse dans la zone qui serait calculée par l'algorithme 2.

D'après ce contre-exemple, nous obtenons donc qu'il n'existe pas de constante k permettant d'appliquer l'algorithme 2 et de tester correctement l'accesibilité. Pour résumer, nous avons le résultat suivant :

Théorème 1 Pour l'automate temporisé $\mathcal{C}$, il n'existe pas de choix de la constante k tel que l'algorithme 2 appliqué avec l'opérateur Approx_k soit correct pour l'accessibilité.

Ce résultat précis sur cette abstraction va même se généraliser. En effet, supposons trouvée une abstraction Abs telle que :
- Si Z est une zone, Abs(Z) est également une zone,
- {Abs(Z) | Z est une zone} est fini,
- Abs est complète, i.e. $Z \subseteq$ Abs(Z),
- Abs est correcte pour l'accessibilité, i.e. que le calcul de (Abs $\circ$ Post)* est correct.

Comme Abs est d'image finie sur les zones, nous pouvons considérer k la plus grande constante apparaissant dans ces zones. Nous avons alors la propriété suivante :

$$\forall Z, Z \subseteq \text{Approx}_k(Z) \subseteq \text{Abs}(Z)$$

puisque, d'après le choix de k, Abs(Z) est une zone k-bornée contenant Z et que par définition $\text{Approx}_k(Z)$ est la plus petite de ces zones. Nous obtenons donc le

Corollaire 1 Pour l'automate temporisé $\mathcal{C}$, il n'existe pas d'opérateur Abs satisfaisant les conditions ci-dessus.

3.2 Résultats positifs

Les résultats de la partie précédente paraissent au premier abord très inquiétants pour les outils basés sur l'algorithme 2. En réalité, la plupart des systèmes étudiés avec ces outils étaient modélisés sans garde diagonale ([HSLL97], [BBP02]). Le résultat suivant donne alors une réponse positive.

Théorème 2 L'algorithme 2 est correct pour l'accessibilité dans les automates temporisés sans garde diagonale.

Afin de compléter la description faite de la situation, voici un dernier théorème.

Théorème 3 L'algorithme 2 est correct pour l'accessibilité dans les automates temporisés avec gardes diagonales et ayant moins de trois horloges. Il faut pour cela choisir la constante k différemment.

Ces deux résultats nous montrent d'une part à quel point le contre-exemple présenté est minimal du point de vue du nombre d'horloges mises en jeu, et d'autre part que le problème de correction auquel nous sommes confrontés provient de la présence des gardes diagonales.

3.3 Automates temporisés avec ou sans gardes diagonales...

Nous sommes donc amenés à présent à comparer la classe des automates temporisés sans garde diagonale, et celle des automates temporisés avec gardes diagonales. Nous avons tout d'abord un premier résultat du point de vue de l'expressivité : l'expressivité de ces deux modèles est la même :

Théorème 4 [BDGP98] Pour tout automate temporisé avec gardes diagonales $\mathcal{A}_d$, il existe un automate temporisé sans garde diagonale $\mathcal{A}_{nd}$ tel que $\mathcal{L}(\mathcal{A}_d) = \mathcal{L}(\mathcal{A}_{nd})$.

Ce résultat peut paraître surprenant. Etant donné un automate temporisé avec gardes diagoanles, il nous donne donc l'existence d'un automate temporisé sans garde diagonale acceptant le même langage temporisé. Cette propriété est même effective.

Considérons un automate temporisé $\mathcal{A}$ contenant la garde diagonale $g : x - y \leq c$. nous souhaitons retirer cette garde. Pour cela nous considérons deux copies de $\mathcal{A}$. La première correspondra intuitivement aux valuations respectant la garde g, la seconde la garde $\neg g$. Nous passons d'une copie à l'autre lorsque la valeur de vérité de $v \models g$ change, i.e. lors de remises à zéro des horloges x et y. Ceci est représenté sur la figure 7.

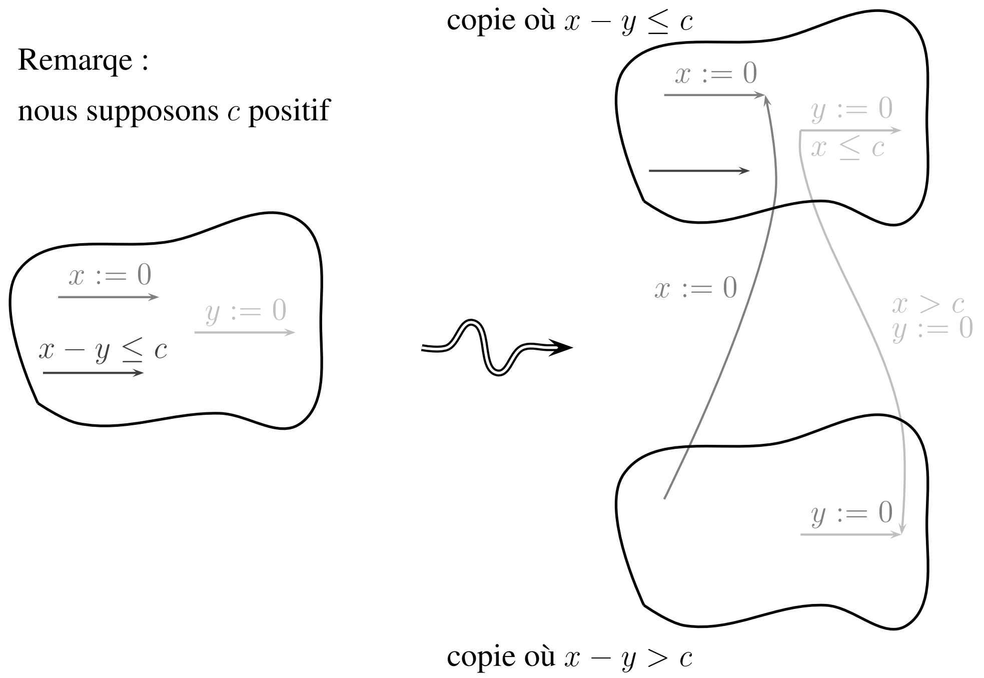


FIG. 7 – Retrait de la garde diagonale $x - y \leq c$

Du point de vue de la complexité, cette construction est malheureusement assez coûteuse. Ainsi, pour enlever une garde diagonale, nous avons doublé la taille de la structure, pour n gardes diagonales, nous la multipierons par 2^n. Cette construction nous donne donc un algorithme pour décider l'accessibilité dans les automates temporisés avec gardes diagonales qui consiste à faire un prétraitement pour transformer l'automate en un automate sans garde diagonale, puis lui appliquer l'algorithme classique. Cependant cette méthode n'est pas satisfaisante du point de vue de la complexité puisque la phase de prétraitement est exponentielle.

Le but de ce stage consiste alors à chercher à mieux comprendre le rôle des gardes diagonales, et leur comportement vis-à-vis de l'algorithme 2 de sorte à proposer une alternative pour vérifier les automates temporisés avec gardes diagonales sans les éliminer toutes, et sans revenir aux régions. Nous allons voir dans la partie suivante quelle méthode nous avons adoptée pour approcher ce problème.

Deuxième partie

Quelle approche du problème ?

Chapitre 4

Choix de la méthode

Rappelons tout d'abord que nous avons vu que l'algorithme calcule une surapproximation de l'ensemble des états accessibles, et donc que ses seules erreurs sont des faux positifs, i.e. qu'il peut déclarer accessibles des états qui ne le sont pas. Nous parlerons de *contre-exemple erroné*. Pour vérifier les réponses de l'algorithme, il suffit de lui demander de retourner la trace du chemin qui l'a mené à cet état, et de vérifier que ce chemin est correct. Cependant, nous allons voir que ce n'est pas forcément suffisant.

4.1 Premières pistes

Intuition 1. Un premier point de vue consiste à penser qu'il suffit de vérifier les réponses de l'algorithme 2. Ceci suppose que si l'algorithme 2 renvoie un contre-exemple qui s'avère être erroné, alors il n'existe pas de vrai contre-exemple dans l'automate temporisé étudié. C'est par exemple le cas dans le contre-exemple $\mathcal{C}$ prouvant la non-correction de l'algorithme. Cette intuition est en fait fausse. Pour le voir, il faut observer précisément le fonctionnement de l'algorithme 2.

En effet, cette affirmation est étroitement liée à l'ordre dans lequel l'algorithme parcourt le graphe de dépliage associé à l'automate temporisé. L'idée sous-jacente est que l'algorithme peut trouver un contre-exemple erroné avant d'avoir pu rencontrer un vrai contre-exemple. Deux méthodes classiques de parcours existent : le parcours en largeur et le parcours en profondeur, la plus naturelle étant la première. Notons d'ailleurs que dans notre cas, cela consiste en fait simplement à avoir une gestion FIFO (resp. LIFO) de la file "En_Attente" pour le parcours en largeur (resp. en profondeur). Dans les deux cas, cette intuition peut facilement être niée à l'aide de contre-exemples adaptés. Nous présentons ici un contre-exemple dans le cas d'un parcours en largeur qui est celui le plus couramment implémenté.

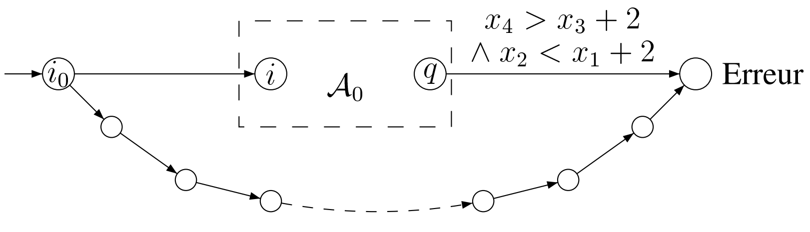


FIG. 8 – Contre-exemple pour le parcours en largeur

Fixons k la constante d'extrapolation choisie et notons $l(k)$ la longueur du contre-exemple erroné minimal associé à $\mathcal{C}$. Le chemin du bas dans l'automate représenté en figure 8 est un chemin sans contrainte de longueur strictement plus grande que $l(k)$. Ceci assure que la découverte du contre-exemple erroné du haut intervient avant celle du vrai contre-exemple du bas et permet de nier cette première intuition.

Remarque : Comme nous l'avons expliqué, l'automate proposé ici dépend de la constante k choisie pour l'extrapolation. Cela peut paraître insatisfaisant. Alors que l'automate $\mathcal{C}$ constitue un contre-exemple valable pour toute constante d'extrapolation k, il est dans notre cas impossible de trouver un tel automate, valable pour tout k. En effet, il s'agit de remarquer que si un automate possède un vrai contre-exemple, alors il existe une valeur de k qui permet d'obtenir un vrai contre-exemple en premier. Notons l la longueur du plus court vrai contre-exemple et posons $\mathcal{E}= \{$zones Z obtenues selon un chemin de longueur $\leq l\}$. Il suffit de prendre pour k la plus grande constante apparaissant dans les zones appartenant à $\mathcal{E}$. Nous sommes alors assurés que le calcul (approché) réalisé par l'algorithme 2 coincidera sur l'ensemble des chemins de longueur inférieure ou égale à l avec le calcul (exact) réalisé par l'algorithme 1, et donc qu'aucun contre-exemple erroné ne sera obtenu avant ce vrai contre-exemple.

Intuition 2. Un raffinement de l'idée précédente consiste alors à penser qu'il pourrait suffir, une fois un contre-exemple trouvé, de le tester et, s'il s'avère être erroné, de poursuivre le travail de l'algorithme. Malheureusement, l'algorithme, dans son comportement, tient compte du passé puisqu'il fait des comparaisons avec les éléments de ¨Visités" et il peut à cause de cela "rater" un vrai contre-exemple. C'est ce que montre l'étude de l'automate suivant.

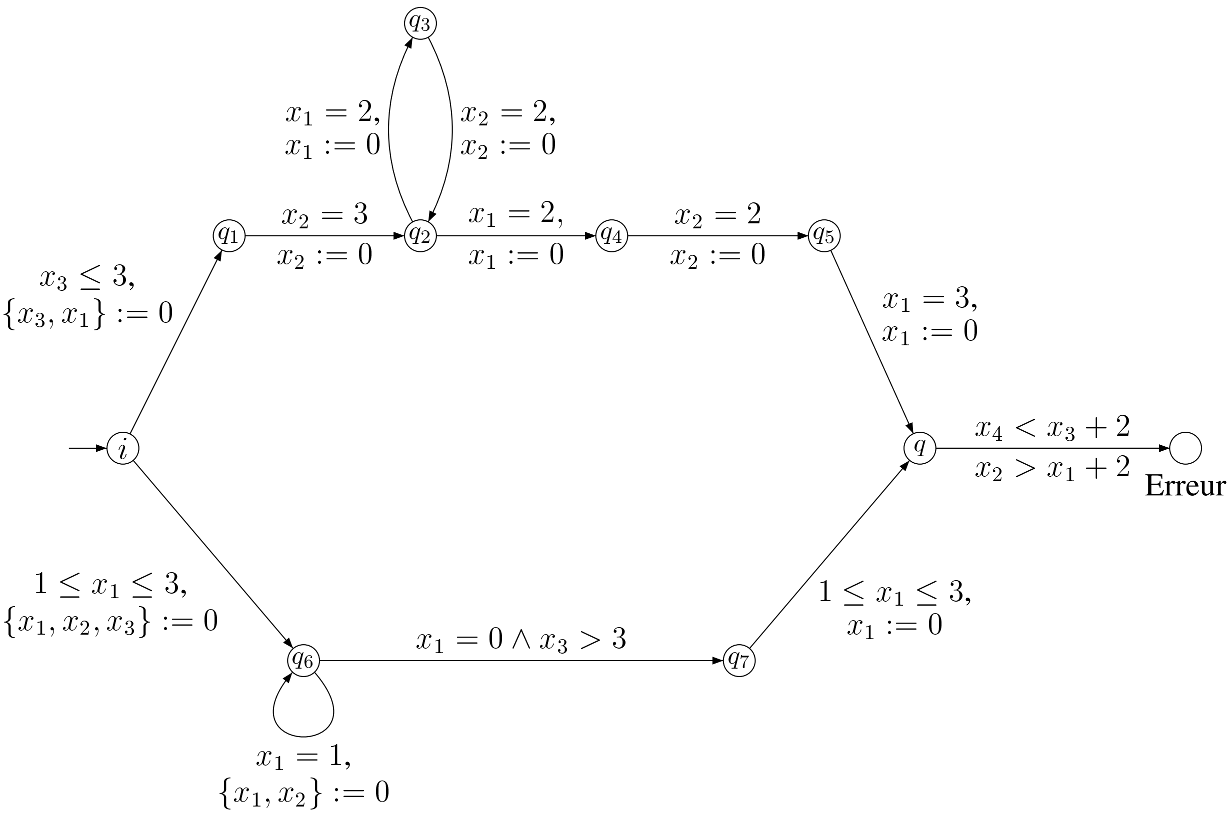


FIG. 9 – Contre-exemple $\mathcal{A}_2$ pour les problèmes de passé

Nous allons étudier l'automate $\mathcal{A}_2$ afin de montrer que l'algorithme, même s'il est relancé après sa première erreur, ne trouvera jamais le vrai contre-exemple.
Cet automate est construit en deux parties. La partie du haut correspond à l'automate $\mathcal{C}$, la partie du bas à un autre automate qui donne un vrai contre-exemple. Par la suite, nous prenons pour constante d'extrapolation la constante maximale de $\mathcal{A}_2$, c'est-à-dire $k = 3$.
Quand nous avons étudié le contre-exemple original en section 3.1, nous avons calculé la zone Z_α obtenue en entrant dans l'état q après avoir pris α fois la boucle. De la même manière, nous pouvons calculer la zone W_β obtenue en entrant en q après avoir pris β fois la boucle sur le chemin du bas : ($\alpha \geq 0$ et $\beta \geq 4$)

$$Z_\alpha = \left\{ \begin{pmatrix} 0 \\ t \\ 2\alpha+5 \\ 2\alpha+5+t \end{pmatrix}, t \in [1,3] \right\} \text{ et } W_\beta = \left\{ \begin{pmatrix} 0 \\ t \\ \beta \\ \beta+t' \end{pmatrix}, t, t' \in [1,3] \right\}$$

Nous constatons alors aisément que pour $\alpha \geq 0$ et $\beta \geq 4$, nous avons

$$\text{Approx}_k(W_\beta) \subseteq \text{Approx}_k(Z_\alpha). \tag{1}$$

Remarque 1 : L'automate accède à l'état q d'abord par le chemin du haut.
En effet, avant de pouvoir franchir la garde $x_1 = 0 \wedge x_3 > 3$, il doit avoir pris au moins 4 fois la boucle sur q_6, tandis que par le haut il peut atteindre directement l'état q.

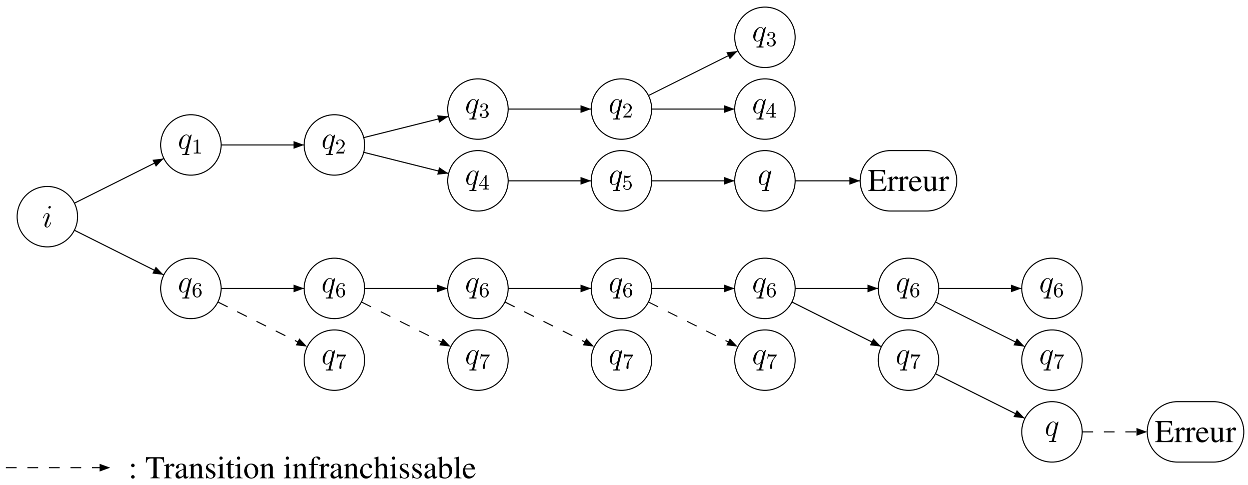


FIG. 10 – Graphe de dépliage associé au parcours de l'algorithme dans $\mathcal{A}_2$.

Remarque 2 : Par la suite, quand il accède à q par le bas, la zone associée est toujours déjà présente dans l'ensemble "Visités" et n'est donc pas traitée.
En effet, quand l'algorithme arrive par le haut en q, c'est (pour le calcul exact) avec la zone Z_0. Le calcul approché donne lui (c'est facile à vérifier) la zone $\text{Approx}_k(Z_0)$. Par ailleurs, quand il arrive par le bas par la suite, c'est, comme nous venons de l'expliquer, toujours avec des valeurs de β plus grandes que 4. Le calcul exact donne donc les zones W_β pour $\beta \geq 4$. Le calcul approché quant à lui donne simplement les approximations des zones précédentes (facile à vérifier à nouveau).

Ces deux remarques, associées à la propriété (1), nous donnent donc le comportement de l'algorithme : il va commencer par trouver le faux contre-exemple minimal associé à $\mathcal{C}$, puis quand il pourra accéder par le bas à q, et sera donc sur le chemin d'un vrai contre-exemple, ne mènera jamais ses calculs au bout car estimera avoir déjà traité ces zones. Il ne trouvera donc jamais le vrai contre-exemple. Nous venons donc de démontrer que pour cet automate, la première erreur de l'automate l'empêche de trouver les vrais contre-exemples par la suite.

Intuition 3. A la vue de cet échec, une nouvelle idée est naturelle : tenir compte dans la suite de l'algorithme des erreurs remarquées lors de la détection d'un contre-exemple erroné. Cela consisterait à retirer de l'ensemble "Visités" les zones liées à un contre-exemple qui s'est avéré être erroné. Nous souhaiterions par exemple dans le cas précédent que l'algorithme puisse remarquer que la zone $\text{Approx}_k(Z_0)$ a mené à un contre-exemple erroné, et doit donc à nouveau être prise en compte. Cependant, si nous souhaitons que l'algorithme traite à nouveau cette zone, nous risquons de perdre la terminaison de l'algorithme. Nous perdons en effet le test d'inclusion qui permettait d'assurer la terminaison. Considérons à nouveau l'automate $\mathcal{C}$, retraiter cette zone, et l'ensemble des zones obtenues sur le chemin associé au contre-exemple erroné va nous obliger à traiter un nombre infini de contre-exemples erronés puisque chaque tour de boucle supplémentaire donne un "nouveau" contre-exemple erroné. Notons également que l'algorithme 2 peut non seulement "rater" un vrai contre-exemple par la suite (cf $\mathcal{A}_2$), mais il peut aussi avoir déjà "éliminé" un chemin menant à un vrai contre-exemple.

Une fois toutes ces intuitions rejetées, nous nous sommes donc convaincus que la solution à notre problème n'était pas triviale, et qu'il fallait chercher une méthode plus compliquée.

4.2 Présentation de la méthode par raffinements successifs

Le choix de la méthode utilisée pour résoudre le problème s'appuie sur plusieurs remarques. Tout d'abord, nous avons l'intuition que des "bugs" tels que celui présenté en section 3.1 sont très rares et donc que l'algorithme classique fonctionne correctement sur la majorité des instances réelles. En outre, nous avons la possibilité de détecter ces erreurs très facilement car tester un contre-exemple est très facile (cf plus loin). Enfin, nous avons une technique pour éradiquer ces "bugs" consistant à retirer les gardes diagonales en cause mais celle-ci est coûteuse en terme de complexité. Nous n'avons donc pas intérêt à l'appliquer systématiquement à toutes les gardes diagonales, mais plutôt d'une façon "paresseuse", i.e. seulement aux gardes diagonales **à l'origine** des erreurs de l'algorithme, s'il en existe.

Cette méthode consiste donc à raffiner successivement le modèle, en lui ôtant des gardes diagonales. Pour choisir celles-ci, nous nous appuierons sur les contre-exemples erronés retournés par l'algorithme. Un schéma illustrant le fonctionnement global de la méthode est représenté sur la figure 11. Les différentes opérations à réaliser seront donc les suivantes :

- test du vide, réalisé à l'aide de l'algorithme 2 (semi-correct)
- test de la validité d'un contre-exemple, réalisé par l'algorithme 3 présenté plus bas,
- étude d'un contre-exemple erroné, présentée en détail dans la partie 3,
- raffinement du modèle.

Ce type de méthode est de plus en plus utilisé de nos jours. Il a été introduit par Clarke et al [CGJ+00]. Il s'agissait alors de réduire le nombre d'états d'un système fini en utilisant une abstraction raffinée à chaque itération. Cette méthode peut aussi être utilisée dans le cadre des systèmes infinis. Les problèmes sont alors souvent indécidables, et cette méthode permet d'obtenir des semi-algorithmes. Dans ces deux exemples, il est essentiel que le raffinement progresse strictement à chaque itération, i.e. que le contre-exemple erroné étudié soit éliminé par le raffinement suivant. C'est là une différence majeure avec notre cadre. En effet, dès lors que nous retirons du système au moins une garde diagonale à chaque raffinement, nous obtenons la terminaison et la correction de notre algorithme. Cette propriété provient du fait que le système possède un nombre fini de gardes diagonales et que l'algorithme 2 est correct s'il n'y a pas de garde diagonale.

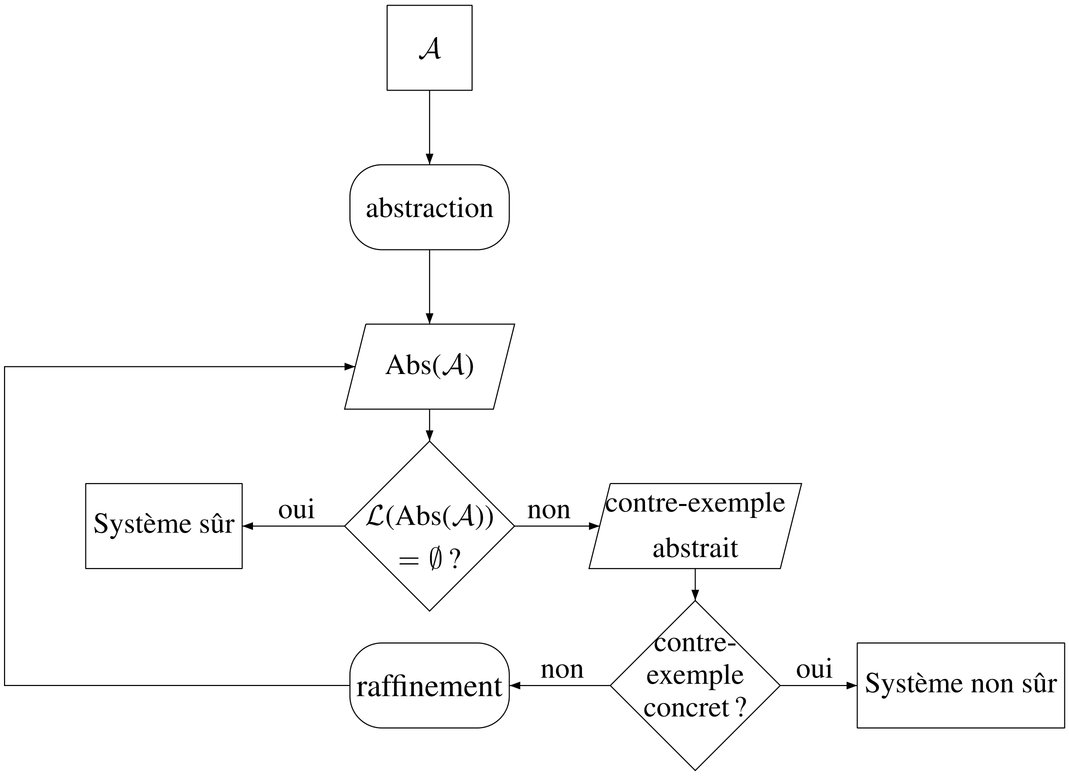


FIG. 11 – Accessibilité par raffinements successifs.

Revenons à présent sur les opérations nécessaires à cette méthode. Pour commencer, intéressons-nous au test de la validité d'un contre-exemple. Celui-ci est très facile à réaliser. En effet, pour tester si un contre-exemple abstrait provient d'un contre-exemple concret, il suffit de réaliser le calcul exact selon la trace donnée. Dans ce cas, nous n'avons pas de problèmes de terminaison puisque nous ne parcourons qu'un chemin fini. Si le calcul exact donne des zones non vides, alors le contre-exemple abstrait est un "vrai contre-exemple", puisque nous ne nous intéressons qu'à des problèmes d'accessibilité discrète. Nous donnons ci-dessous l'algorithme 3 qui teste si un contre-exemple abstrait est erroné ou non et qui renvoie, s'il est erroné, le préfixe du contre-exemple associé à des calculs exacts non vides.

Algorithme 3 Test d'un contre-exemple abstrait

Entrée : $\mathcal{A}$ un AT et $P = [t_1, \ldots, t_p]$ un chemin dans $\mathcal{A}$.

$Z := Z_0$;
$q := q_0$;
$i := 1$;
Tant Que $Z \neq \emptyset$ et $i \leq p$ **Faire**
  $t := P[i]$;
  $i := i + 1$;
  $q := q'$ (*si* t *va de* q *à* q')
  $Z :=$ *Post*(Z, t) ;
Fin Tant Que
Si $Z \neq \emptyset \wedge i = p + 1$ **Alors**
  Retourner "Oui" ;
Sinon
  Retourner $P[1..i - 1]$;
Fin Si

Il reste enfin à discuter de la méthode choisie pour réaliser le raffinement du modèle. Il s'agit, étant donné un ensemble de gardes diagonales, de retirer celles-ci du système. Nous avons déjà présenté en section 3.3 la technique générale. Nous pouvons appliquer cette méthode, et modifier l'ensemble du système, ou bien chercher à réaliser un raffinement "paresseux" [HJMS02]. Ceci consiste à n'effectuer la transformation que sur la branche associée au contre-exemple étudié. Cette technique présente l'avantage de rendre le raffinement plus économique, mais elle risque aussi d'augmenter le nombre d'itérations nécessaires au procédé de raffinement. En effet, nous risquons alors de devoir retirer les mêmes gardes diagonales à différents endroits du système.

L'étape cruciale de la méthode va donc être l'analyse des contre-exemples erronés. Celle-ci doit nous permettre de choisir une garde ou un ensemble de gardes diagonales que l'on estime être coupable(s) de l'erreur. Cette phase sera traitée dans la partie 3. Afin de réaliser cette étude, nous observerons précisément l'exécution de l'algorithme 2 sur certains exemples. Nous utiliserons également certaines propriétés de la structure de données utilisée pour l'implémentation de l'algorithme. Celle-ci est présentée dans le chapitre suivant.

Chapitre 5

Structure de données pour l'analyse de systèmes temporisés

Les algorithmes vus jusqu'à présent utilisent tous la représentation symbolique des zones. Dans ce chapitre, nous allons présenter une structure de données permettant d'implémenter facilement les zones. Nous vérifierons ensuite que toutes les opérations qui nous sont nécessaires sont aisément implémentables.

5.1 Zones et DBMs

DBM. Une *matrice à différences bornées* (nous dirons DBM) pour n horloges est une matrice carrée de dimension $(n+1) \times (n+1)$ contenant des paires

$$(\prec, m) \in \mathbb{V} = (\{<, \leq\}) \times \mathbb{Z} \cup \{(<, \infty)\}$$

Nous noterons les $(n+1)$ horloges $x_0, \ldots, x_n$. L'horloge x_0 sert seulement de référence, i.e. que c'est une “fausse" horloge dont la valeur est supposée toujours être 0. À une DBM $M = (\prec_{i,j}, m_{i,j})_{0 \leq i,j \leq n}$ est associé le sous-ensemble suivant de $\mathbb{T}^n$, qui constitue une zone : (avec la convention $v(x_0) = 0$)

$$[\![M]\!] = \{v : \{x_1, \ldots, x_n\} \to \mathbb{T} \mid \forall\, 0 \leq i,j \leq n, v(x_i) - v(x_j) \prec_{i,j} m_{i,j}\}$$

Exemple. La zone définie par les équations $x_1 > 3 \wedge x_2 \leq 5 \wedge x_1 - x_2 < 4$ peut être par exemple représentée par les deux DBMs suivantes :

$$\begin{pmatrix} (\leq, 0) & (<, -3) & (<, \infty) \\ (<, \infty) & (\leq, 0) & (<, 4) \\ (\leq, 5) & (<, \infty) & (\leq, 0) \end{pmatrix} \text{ et } \begin{pmatrix} (<, \infty) & (<, -3) & (<, \infty) \\ (\leq, \infty) & (<, \infty) & (<, 4) \\ (\leq, 5) & (<, \infty) & (\leq, 0) \end{pmatrix}$$

Forme normale. Avec cette définition, plusieurs DBMs peuvent définir la même zone. Ceci pose problème car il n'est alors pas possible, étant données deux DBMs M_1 et M_2, de tester de manière syntaxique si $[\![M_1]\!] = [\![M_2]\!]$. Une forme normale a donc été introduite afin de représenter sans ambiguïté les zones par des DBMs. La forme normale d'une DBM donnée est simplement l'unique DBM qui, parmi celles qui représentent la même zone, a les coefficients les plus petits. Cette DBM “minimale” représente alors les contraintes les plus fortes. Une fois définis un ordre et une addition sur les éléments de $\mathbb{V}$, nous pouvons utiliser un algorithme classique de plus court chemin, par exemple l'algorithme de Floyd, pour calculer la forme normale.

Ordre sur $\mathbb{V}$ *:* Si $(\prec, m), (\prec', m') \in \mathbb{V}$, alors

$$(\prec, m) \leq (\prec', m') \Leftrightarrow \begin{cases} m < m' \\ \text{ou} \\ m = m' \text{ et } (\prec = \prec' \text{ ou } \prec' = \leq). \end{cases}$$

Addition sur $\mathbb{V}$ *:* Si $(\prec, m), (\prec', m') \in \mathbb{V}$, alors $(\prec, m) + (\prec', m') = (\prec'', m'')$

$$\text{avec } m'' = m + m', \text{ et } \prec'' = \begin{cases} \leq & \text{si } \prec = \prec' = \leq \\ < & \text{sinon} \end{cases}$$

Algorithme 4 Algorithme de Floyd permettant de calculer la forme normale

Entrée : $M = (M_{i,j})_{0 \leq i,j \leq n}$ une DBM.
Sortie : M en forme normale

```
Pour i = 0 à n Faire
  Pour j = 0 à n Faire
    Pour k = 0 à n Faire
      M_{i,j} := min(M_{i,j}, M_{i,k} + M_{k,j})
    Fin Pour
  Fin Pour
Fin Pour
```

Propriétés de la forme normale.
Nous noterons dans la suite $\phi(M)$ la DBM sous forme normale associée à la DBM M. Elle vérifie les propriétés suivantes :

Correction : $[\![\phi(M)]\!] = [\![M]\!]$

Unicité : $[\![M]\!] = [\![M']\!] \Rightarrow \phi(M) = \phi(M')$

La forme normale représente les contraintes les plus fortes :
$$\phi(M) \leq M, \text{ i.e. } \forall\, 0 \leq i, j \leq n, \phi(M)_{i,j} \leq M_{i,j}$$

Inclusion : $[\![M]\!] \subseteq [\![M']\!] \Leftrightarrow \phi(M) \leq M' \Leftrightarrow \phi(M) \leq \phi(M')$

Test du vide. Nous disposons de la propriété suivante :
Soit $M = (\prec_{i,j}, m_{i,j})_{0 \leq i,j \leq n}$ une DBM (pas nécessairement sous forme normale), alors les propriétés suivantes sont équivalentes :
(i) $[\![M]\!] = \emptyset$
(ii) Il existe un cycle strictement négatif dans M, i.e. il existe une séquence d'indices distincts $(i_1, \ldots, i_{l-1})$ telle que (en posant $i_l = i_1$)

$$(\prec_{i_1,i_2}, m_{i_1,i_2}) + \ldots + (\prec_{i_{l-1},i_l}, m_{i_{l-1},i_l}) < (\leq, 0)$$

(iii) $\varphi(M)$ contient un terme $(\prec, m)$ avec $m < 0$ ou $m = 0 \wedge \prec = <$ sur la diagonale

5.2 Opérations utilisées par l'algorithme

Nous devons enfin nous assurer que toutes les opérations dont a besoin l'algorithme sont aisément implémentables grâce aux DBMs ([CGP99]).

Futur. Soit $M = (\prec_{i,j}, m_{i,j})_{0 \leq i,j \leq n}$ une DBM sous forme normale. Définissons la DBM $\overrightarrow{M} = (\prec'_{i,j}, m'_{i,j})_{0 \leq i,j \leq n}$ par :

$$\begin{cases} (\prec'_{i,j}, m'_{i,j}) = (\prec_{i,j}, m_{i,j}) \text{ si } j \neq 0 \\ (\prec'_{i,j}, m'_{i,j}) = (<, \infty) \text{ sinon} \end{cases}$$

Alors nous avons $[\![\overrightarrow{M}]\!] = \overrightarrow{[\![M]\!]}$ et de plus la DBM $\overrightarrow{M}$ est sous forme normale.

Intersection. Soient $M = (\prec_{i,j}, m_{i,j})_{0 \leq i,j \leq n}$ et $M' = (\prec'_{i,j}, m'_{i,j})_{0 \leq i,j \leq n}$ deux DBMs (pas nécessairement sous forme normale). Définissons $M'' = (\prec''_{i,j}, m''_{i,j})_{0 \leq i,j \leq n}$ par :

$$\forall i, j, (\prec''_{i,j}, m''_{i,j}) = \text{min}((\prec_{i,j}, m_{i,j}), (\prec'_{i,j}, m'_{i,j})).$$

Alors nous avons $[\![M'']\!] = [\![M]\!] \cap [\![M']\!]$. Remarquons que nous ne pouvons pas assurer que M'' soit sous forme normale, même si M et M' l'étaient.

Remise à zéro. Soit $M = (\prec_{i,j}, m_{i,j})_{0 \leq i,j \leq n}$ une DBM sous forme normale. Définissons la DBM $M_{x_k:=0} = (\prec'_{i,j}, m'_{i,j})_{0 \leq i,j \leq n}$ par :

$$\begin{cases} (\prec'_{i,j}, m'_{i,j}) = (\prec_{i,j}, m_{i,j}) \text{ si } i, j \neq k \\ (\prec'_{k,k}, m'_{k,k}) = (\prec'_{k,0}, m'_{k,0}) = (\prec'_{0,k}, m'_{0,k}) = (\leq, 0) \\ (\prec'_{i,k}, m'_{i,k}) = (\prec_{i,0}, m_{i,0}) \text{ si } i \neq k \\ (\prec'_{k,i}, m'_{k,i}) = (\prec_{0,i}, m_{0,i}) \text{ si } i \neq k \end{cases}$$

Alors nous avons $[\![M_{x_k:=0}]\!] = [x_k \leftarrow 0][\![M]\!]$ et de plus la DBM $M_{x_k:=0}$ est sous forme normale.

k-approximation. Soit $M = (\prec_{i,j}, m_{i,j})_{0 \leq i,j \leq n}$ une DBM sous forme normale. Définissons la DBM $\overline{M}^k = (\prec'_{i,j}, m'_{i,j})_{0 \leq i,j \leq n}$ par :

$$\begin{cases} (\prec'_{i,j}, m'_{i,j}) = (\prec_{i,j}, m_{i,j}) \text{ si } |m_{i,j}| \leq k \\ (\prec'_{i,j}, m'_{i,j}) = (<, \infty) \text{ si } m_{i,j} > k \\ (\prec'_{i,j}, m'_{i,j}) = (<, -k) \text{ si } m_{i,j} < -k \end{cases}$$

Alors nous avons $[\![\overline{M}^k]\!] = \text{Approx}_k([\![M]\!])$ mais la DBM $\overline{M}^k$ n'est pas sous forme normale. S'il n'y a pas d'ambiguïté sur la constante d'extrapolation k, nous noterons $\overline{M}$ au lieu de $\overline{M}^k$.

Nous avons ainsi tous les outils pour implémenter les algorithmes 1, 2 et 3 en utilisant les DBMs pour représenter les zones. Les DBMs sous forme normale constituent donc une structure de données très satisfaisante. C'est d'ailleurs la structure de données de base utilisée dans la plupart des outils sur les systèmes temporisés. Nous en présentons certains dans le chapitre suivant.

Chapitre 6

Choix de l'outil

6.1 Quelques outils existants

Ces méthodes sont implémentées dans plusieurs outils déjà utilisés avec succès dans l'industrie. Voici un panorama de certains outils existants et de leurs fonctionnalités :

- **Uppaal** : développé à Uppsala (Suède) et Aalborg (Dannemark) [LLPY97]
  - accessibilité, bloquage, fragment simple de TCTL
  - analyse en avant
    $\Rightarrow$ `http://www.uppaal.com`
- **HyTech** : développé à Berkeley (USA) [HHWT97]
  - pas de logique de spécification, riche langage de calcul, automates hybrides
  - analyse en avant et en arrière
    $\Rightarrow$ `http://www-cad.eecs.berkeley.edu/~tah/HyTech/`
- **CMC** : développé à Cachan (France) [LL98]
  - logique modale L_ν
  - méthode compositionnelle, analyse en avant
    $\Rightarrow$ `http://www.lsv.ens-cachan.fr/~fl/cmcweb.html`
- **Kronos** : développé à Grenoble (France) [BDM$^+$98]
  - Timed CTL
  - analyse en avant et en arrière
    $\Rightarrow$ `http://www-verimag.imag.fr/TEMPORISE/kronos/`

6.2 Notre choix : CMC

Parmi ces outils nous avons commencé par travailler avec HyTech qui permet beaucoup de souplesse et est très paramétrable. Malheureusement, HyTech ne réalise pas l'abstraction car il est basé sur des calculs exacts. Cela ne nous a donc pas permis de faire les comparaisons souhaitées : avec ou sans abstraction.

Au cours de mon stage de première année à l'ENS Cachan, j'ai travaillé sur CMC pour y implémenter l'algorithme d'analyse en avant classique avec abstraction (algorithme 2). La partie "analyse en avant" est donc encore en test dans l'outil CMC. Mais la connaissance de l'outil le rend parfaitement adaptable à nos besoins. De plus, CMC permet aussi de faire des calculs sur les zones, représentées naturellement par des DBMs et affiche ces DBMs de façon agréable, ce qui facilite la comparaison de zones. Enfin, CMC est construit de façon modulaire, ce qui le rend aisément modifiable.

6.3 Détail des fonctionnalités implémentées

Pour mener à bien le stage, plusieurs fonctionnalités ont été rajoutées. Tout d'abord, l'algorithme 2 ne renvoyait pas encore de trace du contre-exemple trouvé, s'il en trouvait un. Ceci a été implémenté en rajoutant un champ dans la structure de données associée aux configurations leur permettant de pointer vers leur "père". Il suffit ensuite de remonter ainsi jusqu'à l'unique configuration qui est son propre père : la configuration initiale.

Ensuite, il s'agissait de faire des calculs pas à pas sur les zones. CMC est bien adapté à cela, et possédait déjà des fonctions Future, Inter et Reset. En revanche, l'extrapolation était systématique, et on ne pouvait pas faire une simple normalisation. Nous avons donc rajouté les fonctions Floyd, Extra, et Post (Future, Inter et Reset réunies).

CMC permet donc à présent d'appliquer l'algorithme classique ainsi que de faire des calculs précis et contrôlés sur les zones, pour simuler pas à pas l'application de l'algorithme.

Troisième partie

Traitement du problème

Chapitre 7

Etude des contre-exemples erronés : premières constatations

Dans cette partie nous allons réaliser une étude détaillée des contre-exemples erronés afin de percevoir l'origine de l'erreur ou des erreurs de l'algorithme.

L'erreur la plus visible dans l'exécution de l'algorithme est le premier instant où le calcul exact donne une zone vide et donc s'arrête alors que le calcul approché continue. C'est ce que nous allons appeler la *transition erronée*.

Rappel : nous allons donner plusieurs exemples d'automates dans ce chapitre. Tous sont basés sur l'automate $\mathcal{A}_0$, et le fait que lors du passage du calcul exact au calcul approché (i.e. en utilisant Approx$_k$), la contrainte $x_4 - x_3 = x_2 - x_1$ est perdue.

7.1 Transition erronée

7.1.1 Premières notations et définitions

Afin de pouvoir étudier les contre-exemples erronés, nous devons d'abord formaliser les différentes notions entrant en jeu.

Contre-exemple abstrait. Un *contre-exemple abstrait* pour l'automate $\mathcal{A}$ est un chemin acceptant $P = t_1 \dots t_p$ dans $\mathcal{A}$ tel que la zone finale obtenue par le calcul approché est non vide.

Contre-exemple erroné. Un contre-exemple abstrait est *erroné* si la zone finale obtenue par le calcul exact le long de ce contre-exemple est vide.

Notons désormais Z_0 la zone contenant l'unique valuation v telle que $\forall x \in X, v(x) = 0$, et $Z_1, \dots, Z_p$ (resp. $Z'_1, \dots, Z'_p$) les zones obtenues selon le chemin P par le calcul exact (resp. approché). Pour la suite du rapport, il peut être utile de fixer quelques notations complémentaires : nous noterons en général M la DBM en forme normale représentant la zone Z. Ainsi, selon les notations précédentes, pour le chemin P, les DBMs associées au calcul exact seront notées $M_0, \dots, M_p$ et celles associées au calcul approché $M'_0, \dots, M'_p$. Formellement, nous avons donc $\begin{cases} M_{i+1} = Post(M_i, t_{i+1}) \\ M'_{i+1} = \overline{Post(M'_i, t_{i+1})}^k \end{cases}$

Transition et garde erronées. Etant donné un contre-exemple erroné P, et selon les notations précédentes, la *transition erronée* associée à P est la transition t_j avec $j = \min\{i \mid Z_i = \emptyset\}$, qui est bien défini puisque par hypothèse $Z_p = \emptyset$. La garde de cette transition sera appelée *garde erronée*.

7.1.2 La transition erronée n'est pas toujours diagonale

Comme nous savons que c'est la présence de gardes diagonales qui rend incorrect l'algorithme, nous pourrions penser que cette première garde erronée est forcément diagonale (ou contient une garde diagonale). Ce n'est en fait pas le cas comme le montre ce premier contre-exemple. (figure 12)

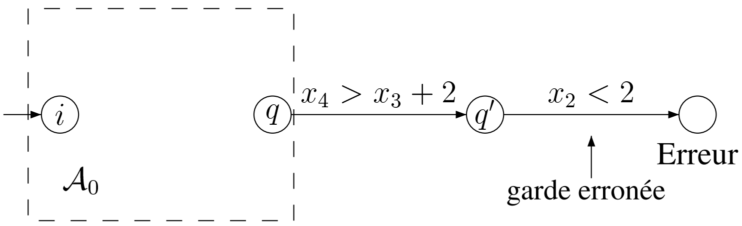


FIG. 12 – La garde erronée peut être non diagonale.

Pour comprendre le fonctionnement de ce contre-exemple, il s'agit de remarquer que nous avons simplement cherché à transformer la garde d'origine : $x_2 - x_1 < 2 \wedge x_4 - x_3 > 2$. Pour cela, nous avons choisi la première des deux gardes diagonales et nous l'avons "dédiagonalisée" : nous avons trouvé une garde non diagonale qui l'implique et qui est franchissable. En effet, il est toujours vrai que $x_1 \geq 0$ et comme nous imposons de surcroît $x_2 < 2$, il en découle $x_2 - x_1 < 2$.

7.2 Transition instable initiale

7.2.1 Définitions

Après cette première constatation, nous sommes donc amenés à chercher à comprendre pourquoi une garde (non diagonale) peut être franchissable dans le calcul exact et pas dans le calcul approché. Il est facile de voir que si cette garde est simple (i.e. qu'elle n'est pas la conjonction de plusieurs gardes atomiques), cela impose que le coefficient associé à cette garde n'est pas le même dans les deux DBMs en forme normale associées aux zones obtenues dans le calcul exact et le calcul approché. Le calcul de CMC présenté en annexe A associé à l'automate de la figure 12 corrobore cette remarque. Ceci nous amène donc à étudier un nouvel aspect des contre-exemples : le fait que certains coefficients des DBMs sont altérés.

Petits coefficients et partie bornée.
Soit $M = (\prec_{i,j}, m_{i,j})_{0 \leq i,j \leq n}$ une DBM. Notons k la constante d'extrapolation. Un coefficient

(i, j) de M est appelé *petit coefficient* si $|m_{i,j}| \leq k$. Nous appelerons *partie bornée de M* l'ensemble de ses petits coefficients et le noterons Bornée(M).

Etant donné un contre-exemple erroné P, et selon les notations introduites dans la section précédente, nous considérons les suites $M_1, \ldots, M_p$ (resp. $M'_1, \ldots, M'_p$) des DBMs en forme normale associées au calcul exact (resp. approché) selon P.

Petits coefficients erronés.
Soit un indice $1 \leq k \leq p$, nous dirons que le coefficient (i, j) de la matrice M_k est un *petit coefficient erroné* si c'est un petit coefficient de M_k et que l'on a $M_k[i, j] < M'_k[i, j]$.

La remarque faite précédemment s'exprime comme suit : étant donné un contre-exemple erroné tel que la garde erronée soit une garde simple, alors il existe un coefficient de la DBM du calcul approché précédant cette garde qui est un petit coefficient erroné.

Comme nous nous intéressons dans cette étude aux gardes des transitions, il s'agit à présent de lier ces coefficients erronés aux gardes qui ont causé ces erreurs. C'est le but de la définition suivante.

Transition et garde instables initiales.
Nous nous plaçons toujours dans le même cadre et supposons de plus que la garde erronée est une garde simple. Nous la notons t_k. Ceci implique donc l'existence d'un petit coefficient erroné dans la matrice M'_{k-1}. En particulier, nous avons que Bornée$(M'_{k-1}) \neq$ Bornée(M_{k-1}). Posons alors $j = \min\{1 \leq l \leq k-1 \mid \text{Bornée}(M'_l) \neq \text{Bornée}(M_l)\}$. j est donc bien défini et nous dirons que la transition (resp. la garde de la transition) t_j est la *transition* (resp. la *garde*) *instable initiale* du contre-exemple erroné P.

Le fait qu'il existe des petits coefficients des DBMs du calcul approché qui sont faux montre que le calcul approché fait dans ce cas de "graves erreurs". Nous pouvons donc penser que cela est dû à des gardes diagonales et semble être une excellente piste pour la suite. Nous résumons sur la figure 13 les différentes définitions introduites jusqu'à présent.

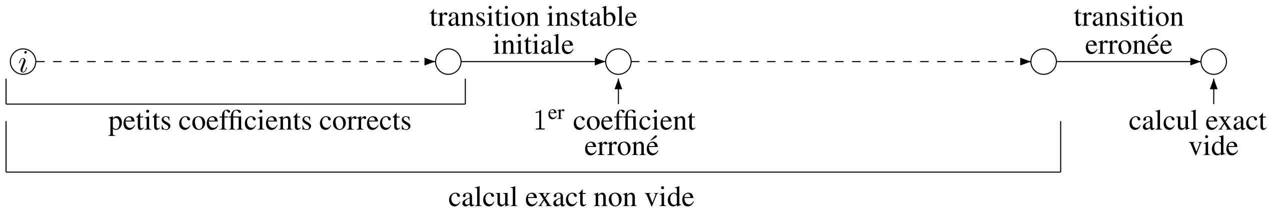


FIG. 13 – Situation générale.

Nous nous intéresserons donc par la suite à la partie du contre-exemple comprise entre la garde instable initiale et la garde erronée, puisqu'il semble que ce soit dans cette zone que les erreurs se créent et causent le "bug".

7.2.2 La transition instable initiale n'est pas toujours diagonale

Avec tout ce que nous venons de voir, il pourrait être naturel de penser que seules les gardes diagonales peuvent être instables initiales. Ce n'est malheureusement pas le cas, comme le montre le contre-exemple représenté en figure 14. Pire encore, non seulement la garde instable initiale peut être non diagonale, mais en fait **des petits coefficients erronés peuvent apparaître au cours d'un calcul approché dans un automate sans garde diagonale**. C'est le cas dans ce contre-exemple dont les calculs associés effectués à l'aide de CMC sont présentés en annexe A.

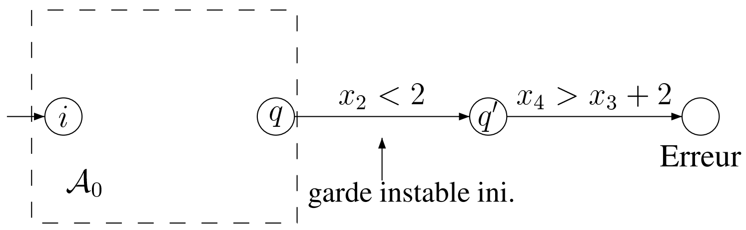


FIG. 14 – La garde instable initiale peut être non diagonale.

Remarque : cela peut, au premier abord, paraître contradictoire avec la correction de l'algorithme classique sans garde diagonale. En fait, il n'en est rien. En effet, si nous observons le coefficient modifié par la garde non diagonale, celui-ci n'est pas testable par des gardes non diagonales. Plus précisément, le coefficient modifié ici est $x_4 - x_3 \prec_{4,3} m_{4,3}$. Les horloges x_3 et x_4 sont de plus toutes deux "non bornées", au sens où, si Z désigne la zone associée à l'état q dans le calcul présenté, nous avons : $\forall v \in Z, v(x_3) > K \wedge v(x_4) > K$. Dans le contre-exemple de la figure 12, nous avons trouvé pour les horloges x_1 et x_2 une conjonction de gardes non diagonales g telle que $g \Rightarrow x_2 - x_1 < 2$ et g est franchissable dans le calcul approché. Ce n'est pas possible dans ce cas, intuitivement car les horloges x_3 et x_4 sont non bornées : pour les tester, il faut les ramener dans la partie bornée, mais cela n'est possible qu'à l'aide d'une remise à zéro. Cette remise à zéro effacerait alors l'erreur placée sur une différence d'horloges. La preuve formelle du fait que c'est impossible repose sur la correction de l'algorithme dans le cas non diagonal.

Pour conclure sur ce point, nous pouvons remarquer que nous avons évoqué deux types d'objets qui se distinguent. Ainsi la garde $x_2 < x_1 + 2$ semble être "dédiagonalisable" tandis que la garde $x_4 > x_3 + 2$ semble ne pas l'être. De même, le coefficient associé à la différence d'hologes $x_2 - x_1$ est "testable" tandis que le coefficient $x_3 - x_4$ ne l'est pas. Ces propriétés paraîssent étroitement liées et dépendantes du contexte. Nous présenterons quelques résultats sur ce thème dans le chapitre suivant.

7.3 Quand les gardes diagonales se cachent...

Nous avons présenté deux types de gardes aisément détectables : la garde erronée et la garde instable initiale. Nous avons cependant montré que l'une comme l'autre peuvent être non diagonales. Une question se pose alors : l'une et l'autre peuvent-elles être simultanément non diagonales ? La réponse à cette question est positive et l'automate présenté en figure 15 constitue un exemple pour cette situation particulière. Encore une fois, les calculs de CMC associés sont présentés en annexe A.

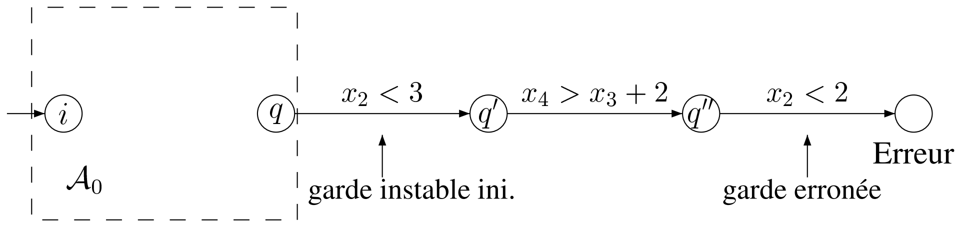


FIG. 15 – Les gardes instables initiales et erronées peuvent toutes deux être non diagonales.

À nouveau, nous allons décrire la construction de ce contre-exemple. Comme dans le contre-exemple de la figure 14, la garde diagonale $x_2 < 3$, associée à $x_1 \geq 0$ (toujours vérifiée), simule la garde diagonale $x_2 - x_1 < 3$. Cette garde rend erroné le petit coefficient $x_4 - x_3 \prec_{4,3} m_{4,3}$. Comme nous l'avons expliqué plus haut, il n'est a priori pas possible de tester cette erreur sans garde diagonale car les horloges x_3 et x_4 sont non bornées. Afin de contourner cette difficulté, le contre-exemple que nous présentons utilise, comme nous l'avons fait dans le contre-exemple de la figure 12, la garde diagonale $x_4 > x_3 + 2$ pour créer une nouvelle erreur testable par une garde non diagonale. En effet, dans le calcul exact, la garde $x_4 > x_3 + 2$ entraine $x_2 > x_1 + 2$. La garde $x_2 < 2$, associée à $x_1 \geq 0$, simule à nouveau la garde $x_2 < x_1 + 2$, et est donc erronée.

Ce contre-exemple présente plusieurs caractéristiques très intéressantes. Nous les résumons ici :

- le moment où l'algorithme se trompe peut avoir lieu avec une garde non diagonale (i.e. la garde erronée peut être non diagonale),
- la première erreur n'est pas forcément due à une garde diagonale (i.e. la garde instable initiale peut être non diagonale),
- la première erreur apparaissant sur un contre-exemple erroné n'est pas forcément à l'origine de l'erreur finale de ce contre-exemple,
- de nouvelles erreurs peuvent apparaître au cours du contre-exemple,
- il peut y avoir plusieurs erreurs simultanément, mais d'origines différentes,
- la garde instable initiale n'est pas la seule à causer des erreurs.

Ces remarques nous montrent que le comportement des erreurs est difficile à cerner, et que les gardes diagonales ne sont pas les seules responsables. Il faut donc pouvoir tester si un calcul peut être obtenu par un automate temporisé sans garde diagonale (et donc est correct pour l'accessibilité). Nous allons développer deux études plus précises dans le chapitre suivant afin de satisfaire cet objectif.

Chapitre 8

Deux méthodes pour choisir les gardes diagonales en cause

Nous pouvons à présent chercher à dégager des méthodes permettant de déterminer les gardes diagonales responsables. Pour cela, nous allons proposer deux approches. Dans un premier temps, nous allons définir un critère afin de caractériser les gardes diagonales pouvant être remplacées par des gardes non diagonales. Ensuite nous chercherons à construire un historique des erreurs, i.e. à établir les liens entre les différentes erreurs apparaissant au cours du contre-exemple.

8.1 Critère pour la dédiagonalisation

8.1.1 Définition

Nous avons déjà évoqué le fait que certaines gardes diagonales peuvent être "dédiagonalisées". C'est-à-dire que nous pouvons trouver des gardes non diagonales qui donnent des calculs similaires, que ce soit le calcul exact ou approché. Ces gardes diagonales doivent être considérées intuitivement comme des gardes diagonales "innocentes" puisque nous savons que l'algorithme 2 est correct dans le cas non diagonal. D'autres gardes diagonales en revanche ne peuvent pas être ainsi tranformées. Ce sont celles qui créent des erreurs testables par des gardes non diagonales. Nous allons présenter un critère permettant de caractériser ces gardes.

Garde diagonale dédiagonalisable Une garde simple diagonale g d'une transition t d'un chemin P dans automate temporisé $\mathcal{A}$ est dite *dédiagonalisable* s'il existe un automate temporisé sans garde diagonale et un chemin P' dans cet automate tels qu'à l'issue de ce chemin, les zones correspondant aux calculs exact et approché soient les mêmes que celles obtenues à l'issue de t suivant P.

Cette définition permet d'assurer que les éventuelles erreurs faites à l'issue de g pourraient être causées par des gardes non diagonales, ce qui signifie que l'algorithme 2 n'a pas encore fait d'erreur puisqu'il est correct dans le cas non diagonal.
Nous connaissons une opération permettant de retirer une garde diagonale. Nous allons nous inspirer de sa construction pour déterminer si une garde diagonale est ou non dédiagonalisable.

Considérons donc un contre-exemple erroné P et une garde diagonale $g = y - x \sim c$ apparaissant sur P telle que toutes les gardes précédentes sont non diagonales. L'application de la construction présentée en section 3.3 dans ce cas est particulière car l'automate associé à un chemin est un automate linéaire. Aussi, il suffit en fait de remonter la garde g seulement jusqu'à la dernière remise à zéro de l'une ou l'autre des deux horloges x et y. Ceci est représenté sur la figure 16.

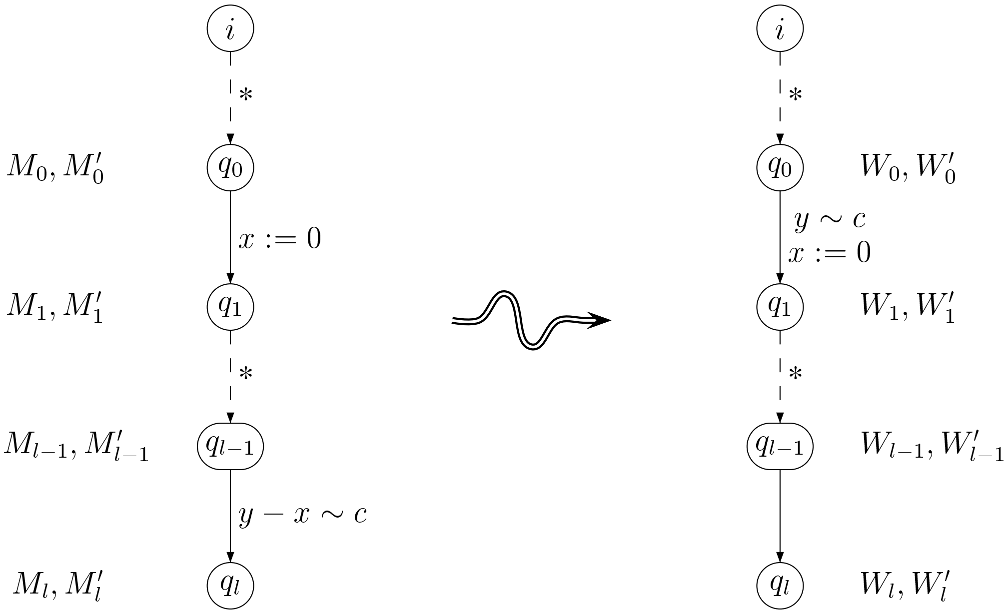


FIG. 16 – Dédiagonalisation d'un automate temporisé linéaire

Introduisons à présent des notations. Nous notons, comme indiqué sur la figure 16, $q_0, \ldots, q_l$ la partie du chemin qui nous intéresse. Les DBMs en forme normale correspondant au calcul exact (resp. approché) avant dédiagonalisation sont notées $M_0, \ldots, M_l$ (resp. $M'_0, \ldots, M'_l$). Pour l'automate après dédiagonalisation, ces DBMs sont notées $W_0, \ldots, W_l$ et $W'_0, \ldots, W'_l$.

Nous savons que cette méthode conserve le calcul exact, i.e. qu'à l'issue de la garde contenant la garde g, nous avons la même DBM pour chaque calcul exact. Selon nos notations, ceci s'écrit $M_l = W_l$. Nous souhaiterions que le comportement de notre algorithme soit le même, et donc que les zones obtenues avec le calcul approché soient les mêmes. Cette condition s'écrit ainsi :

$$M'_l = W'_l \tag{*}$$

Nous avons pour conclure les implications suivantes :

$$M'_l = W'_l \implies \text{la garde } g \text{ est dédiagonalisable} \implies \text{la garde } g \text{ n'est pas coupable}$$

8.1.2 Cas remarquables

Il est possible de raffiner le critère présenté précédemment. Il existe en effet plusieurs cas remarquables dans lesquels nous pouvons préciser les résultats. Nous présentons pour commencer un critère permettant de conclure directement que la garde est dédiagonalisable.

Proposition : S'il n'y a aucun petit coefficient erroné après la transition associée à la garde diagonale g, i.e. si Bornée(M_l) = Bornée(M'_l), alors nous avons toujours $M'_l = W'_l$.

preuve : Notons k la constante d'extrapolation. Rappelons que nous notons $\overline{M}$ la DBM obtenue en extrapolant la DBM M et $[\![M]\!]$ la zone associée à M. Selon nos notations, comme l'opérateur d'extrapolation est croissant et qu'il réalise une surapproximation, nous avons (récurrence) :

$$\forall i, [\![W_i]\!] \subseteq [\![\overline{W_i}]\!] \subseteq [\![W'_i]\!] \tag{1}$$

Comme le chemin dans le nouvel automate a des contraintes plus restrictives que celles de l'ancien chemin, et comme Approx$_k$ est croissant, nous avons les inclusions suivantes (récurrence) :

$$\forall i, [\![W'_i]\!] \subseteq [\![M'_i]\!] \tag{2}$$

Réécrivons à présent nos hypothèses :
- (i) Bornée(M_l) = Bornée(M'_l) entraine $\overline{M_l} = \overline{M'_l}$ et comme M'_l a été obtenue par l'opérateur d'abstraction, elle est déjà k-bornée et nous avons donc $[\![\overline{M_l}]\!] = [\![M'_l]\!]$,
- (ii) L'égalité des deux calculs exacts entraine $M_l = W_l$ et donc $[\![\overline{M_l}]\!] = [\![\overline{W_l}]\!]$.

En appliquant successivement (1) et (2) pour $i = l$, puis (i) et (ii), nous obtenons :

$$[\![W_l]\!] \subseteq [\![\overline{W_l}]\!] \subseteq [\![W'_l]\!] \subseteq [\![M'_l]\!] = [\![\overline{M_l}]\!] = [\![\overline{W_l}]\!]$$

Comme nous retrouvons deux fois le terme $[\![\overline{W_l}]\!]$, nous en déduisons que la suite d'inégalités entre ces deux termes est en fait une suite d'égalités, et en particulier nous avons $[\![M'_l]\!] = [\![W'_l]\!]$ et donc, par unicité de la forme normale, $M'_l = W'_l$. CQFD

Ce premier critère nous permet donc d'éviter de faire certains tests. De plus, sa démonstration nous a permis de mieux comprendre la situation. Ainsi, nous avons toujours la propriété suivante :

$$[\![W_l]\!] = [\![M_l]\!] \subseteq [\![W'_l]\!] \subseteq [\![M'_l]\!] \tag{3}$$

Nous avons pour l'instant discuté de l'égalité entre $[\![W'_l]\!]$ et $[\![M'_l]\!]$. L'autre inclusion apporte un renseignement plus précis dans le cas d'une garde non dédiagonalisable. Supposons en effet que $[\![M_l]\!] = [\![W'_l]\!]$ et $[\![W'_l]\!] \subsetneq [\![M'_l]\!]$. Nous venons alors de trouver une garde telle que, si elle est ôtée, le calcul approché réalisé par l'algorithme 2 coincide avec le calcul exact de l'algorithme 1. Cette garde constitue donc un excellent choix.

Exemples. Donnons à présent quelques exemples pour illustrer ces différentes situations.

Premier cas : la garde est dédiagonalisable, mais il y a un petit coefficient erroné (ce cas montre que la réciproque de la proposition est fausse). Pour cela, considérons le chemin suivant :

$$x_2 \geq 1$$

$$\xrightarrow[\{x_1, x_3\} := 0]{x_3 \leq 3} \xrightarrow[x_2 := 0]{x_2 = 3} \xrightarrow[x_1 := 0]{x_1 = 2} \xrightarrow[x_2 := 0]{x_2 = 2} \xrightarrow[x_1 := 0]{x_1 = 3} \xrightarrow[\wedge\, x_2 < 2]{x_2 \geq x_1 + 1} \xrightarrow{x_4 > x_3 + 2}$$

Nous obtenons dans ce cas (cf calculs en annexe) la situation suivante :

– $[\![W'_6]\!] = [\![M'_6]\!]$
– $[\![W_6]\!] \subsetneq [\![W'_6]\!]$
La garde est donc dédiagonalisable mais nous n'aurions pas pu nous en rendre compte à partir du critère présenté dans la Proposition.

Deuxième cas : la garde n'est pas dédiagonalisable, mais ne suffit pas pour obtenir un calcul approché coincidant avec le calcul exact.

$$x_2 < 2$$

$$\xrightarrow[\{x_1, x_3\} := 0]{x_3 \leq 3} \xrightarrow[x_2 := 0]{x_2 = 3} \xrightarrow[x_1 := 0]{x_1 = 2} \xrightarrow[x_2 := 0]{x_2 = 2} \xrightarrow[x_1 := 0]{x_1 = 3} \xrightarrow[\wedge\, x_4 > x_3 + 2]{x_2 < x_1 + 2}$$

Cette fois, les résultats sont les suivants :
– $[\![W'_6]\!] \subsetneq [\![M'_6]\!]$
– $[\![M_6]\!] \subsetneq [\![W'_6]\!]$
La garde n'est donc pas dédiagonalisable, et le nouveau calcul approché est donc plus proche du calcul exact que l'ancien. Cependant, il n'est pas encore égal au calcul exact comme le montre le deuxième point, et il reste donc des erreurs.

Troisième cas : la garde n'est pas dédiagonalisable et après l'avoir dédiagonalisée, le calcul approché coincide avec le calcul exact.

$$x_4 > 2$$

$$\xrightarrow[\{x_1, x_3\} := 0]{x_3 \leq 3} \xrightarrow[x_2 := 0]{x_2 = 3} \xrightarrow[x_1 := 0]{x_1 = 2} \xrightarrow[x_2 := 0]{x_2 = 2} \xrightarrow[x_1 := 0]{x_1 = 3} \xrightarrow[\wedge\, x_2 < x_1 + 2]{x_4 > x_3 + 2}$$

Comme annoncé, nous avons cette fois :
– $[\![W'_6]\!] \subsetneq [\![M'_6]\!]$
– $[\![M_6]\!] = [\![W'_6]\!]$
La garde n'est donc pas dédiagonalisable, et le nouveau calcul approché est égal au calcul exact. L'algorithme 2 ne fait donc plus d'erreurs, ce qui nous satisfait pleinement !

8.1.3 Choix des gardes diagonales

Nous allons présenter ici plusieurs méthodes issues du critère précédent. L'intérêt de chacune d'entre elles est discutable et ce ne sont pas les seules.

• Méthode rapide.
Si nous privilégions la rapidité, ce critère peut nous permettre d'obtenir rapidement une garde suspecte. Pour cela, il suffit de tester chacune des gardes diagonales rencontrées le long du contre-exemple et de retourner la première ne vérifiant pas la condition $(*)$.

• Méthode exhaustive.
Nous pouvons également construire un ensemble $\mathcal{G}$ de gardes diagonales tel que si toutes ces gardes sont retirées, alors le contre-exemple étudié ne peut pas être réobtenu. En effet, le critère

développé précédemment repose sur la construction d'un automate temporisé équivalent sans garde diagonale. Plus précisément, il est possible d'itérer le procédé de dédiagonalisation successivement à chaque garde diagonale rencontrée le long du chemin, *en reprenant le résultat obtenu jusque là*. Nous construisons ainsi progressivement un automate temporisé sans garde diagonale ayant le même calcul exact. À chaque application du test à une garde g, si celle-ci ne vérifie pas $(*)$, alors nous l'ajoutons à $\mathcal{G}$. Ce procédé itératif est illustré en figure 17. Le procédé s'arrête quand le nouveau calcul approché obtenu est vide. Nous n'obtiendrons donc plus ce contre-exemple erroné. Cette condition d'arrêt est valide car l'algorithme 2 est correct sans garde diagonale, et donc les calculs approchés et exacts sont vides (resp. non vides) simultanément. Pour conclure, il s'agit de noter que d'après la définition d'un contre-exemple erroné, à l'issue du chemin associé au contre-exemple erroné le calcul exact est vide.

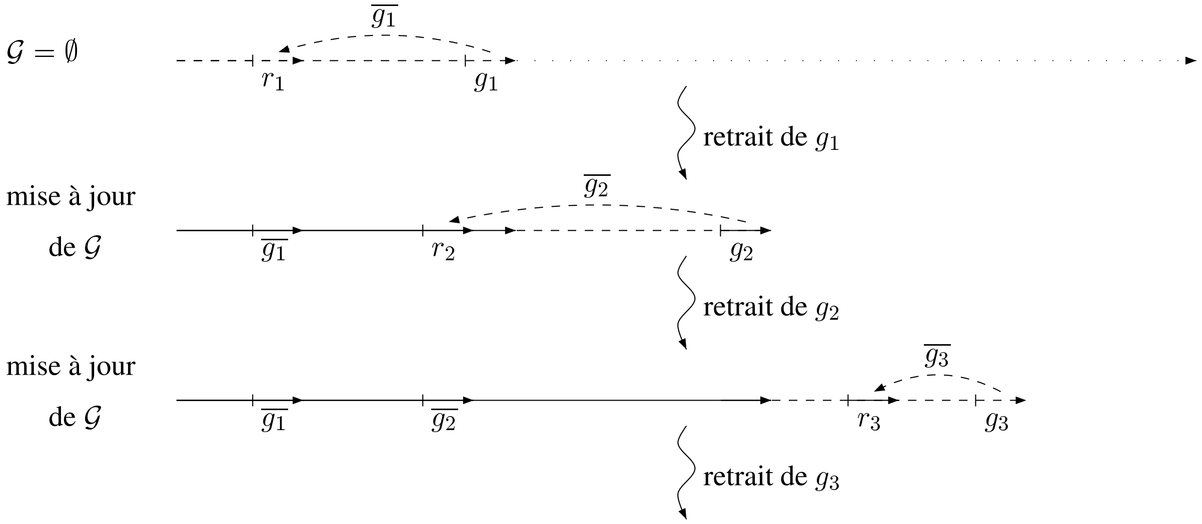


FIG. 17 – Itération du procédé de dédiagonalisation.

Donnons quelques éclaircissements sur la figure 17. L'idée consiste à appliquer successivement la procédure de dédiagonalisation à partir du contre-exemple erroné initial. Notons g_1 la première garde diagonale rencontrée le long de ce contre-exemple erroné. Nous remontons cette garde diagonale en une nouvelle garde non diagonale équivalente $\overline{g_1}$ au niveau de la dernière remise à zéro r_1 d'une des deux horloges intervenant dans g_1. Nous obtenons donc un premier automate temporisé sans garde diagonale dont le calcul exact est le même à l'issue de la transition portant g_1. Nous itérons ensuite ce procédé à partir de ce nouvel automate en cherchant la garde diagonale suivante, notée g_2. À chaque étape, nous mettons à jour l'ensemble $\mathcal{G}$ des gardes diagonales estimées responsables de l'erreur.

• Méthode intermédiaire.

Il s'agit simplement de modifier la condition d'arrêt de la méthode précédente. En effet, nous pouvons estimer l'erreur corrigée dès que le nouveau calcul approché coincide avec les calcul exact. Nous ne pouvons cependant pas garantir dans ce cas que de nouvelles erreurs ne peuvent pas survenir dans la suite du contre-exemple erroné.

Nous présentons ci-dessous l'algorithme 5 réalisant les méthodes exhaustive et intermédiaire, les deux conditions d'arrêt correspondantes y étant reproduites.

Algorithme 5 Choix des coupables

Entrée : un contre-exemple erroné
Sortie : un ensemble de gardes diagonales
  $\mathcal{G}:=\emptyset$;
  Répéter
    Prendre la première garde diagonale suivante g.
    Si il y a un coefficient erroné après g **Alors**
      Faire la dédiagonalisation de g.
      Récupérer ainsi les DBMs M_l, M_l', W_l et W_l' associées.
      Si $[\![W_l']\!] \subsetneq [\![M_l']\!]$ **Alors**
        $\mathcal{G}:=\mathcal{G}\cup\{g\}$
      Fin Si
    Fin Si
  Jusqu'à $[\![W_l']\!]=\emptyset$ *méthode exhaustive*
  Jusqu'à $[\![M_l]\!]=[\![W_l']\!]$ *méthode intermédiaire*
  Retourner $\mathcal{G}$.

8.2 Pistage des erreurs

Comme nous l'avons vu dans le chapitre précédent, les erreurs peuvent apparaître, disparaître, se propager, etc... Pourtant, à la fin du contre-exemple, quitte à couper la transition erronée de sorte à séparer les gardes simples constituant la garde erronée, nous pouvons nous ramener à un seul coefficient erroné responsable du faux contre-exemple. Notre souhait consiste donc, au cours du parcours du contre-exemple erroné, à maintenir à jour l'histoire de tous les coefficients erronés apaaraissant. Nous obtiendrons ainsi à l'issue du contre-exemple erroné l'ensemble des origines du coefficient erroné source du "bug", et donc l'ensemble des gardes diagonales coupables. Cet objectif est en fait très difficile, et nous ne pouvons pour l'instant que faire quelques propositions permettant de s'en approcher.

Nous allons d'abord présenter quelques exemples afin de montrer la complexité du problème, ainsi que les comportements classiques.

8.2.1 Exemples et définitions

Propagation simple. Nous commençons par un exemple simple. Nous considérons deux DBMs (une correspondant au calcul exact et l'autre au calcul approché) telles qu'il y a un petit coefficient erroné. Nous allons voir qu'une garde simple non diagonale peut propager cette erreur à un autre coefficient. Ce phénomène est relativement simple à détecter, nous allons expliquer pourquoi.
Nous considérons l'automate présenté sur la figure 18. La garde $x_4 > x_3 + 2$ est instable et à l'issue de cette garde le coefficient correspondant à $x_2 - x_1$ vaut $(<, -2)$ dans le calcul exact et $(\leq, -3)$ dans le calcul approché. La garde $x_1 = 0$, au lieu d'être erronée comme dans les

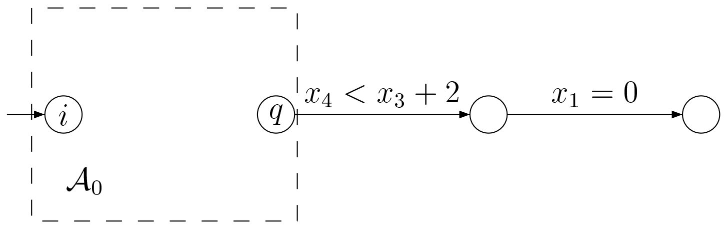


FIG. 18 – Propagation d'une erreur.

exemples précédents, va cette fois faire se propager l'erreur. En effet, le fait d'imposer une limite supérieure à x_1 va permettre à l'erreur sur $x_2 - x_1$ de se propager en une mauvaise borne supérieure pour x_2.

Ce type d'erreur semble facile à déceler car en étudiant l'action de l'algorithme de Floyd sur les DBMs, nous pouvons nous rendre compte que la nouvelle valeur du coefficient associé à $x_2 - x_0$ a été obtenue à partir de la garde $x_1 = 0$ et du coefficient erroné $x_2 - x_1 \leq 3$.

Disparition d'une erreur. Nous pourrions souhaiter systématiquement déclarer erronés les coefficients obtenus par un chemin contenant un coefficient erroné. Pourtant, ces coefficients ne sont pas toujours erronés. En effet, il est possible qu'une garde "corrige" cette erreur potentielle. Ainsi, si une contrainte était manquante dans le calcul approché mais, avant même qu'elle soit testée, cette contrainte est "rajoutée" par une garde qui l'implique, alors l'erreur est corrigée.

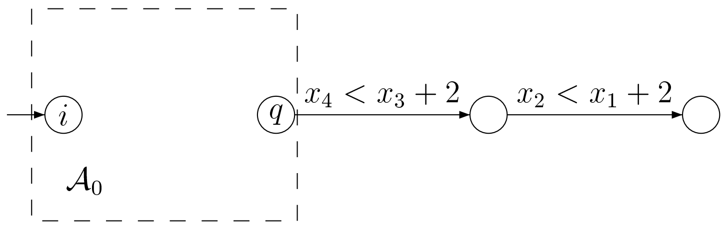


FIG. 19 – Disparition d'une erreur.

Cet exemple est une illustration directe de ce qui précède : la garde $x_4 < x_3 + 2$ implique la contrainte $x_2 < x_1 + 2$ dans le calcul exact, mais pas dans le calul approché. Mais la garde $x_2 < x_1 + 2$ est franchie immédiatement après et corrige donc l'erreur.

Apparition d'une erreur. Pourtant, tous les coefficients erronés ne proviennent pas d'erreurs précédentes. Il est également possible que des erreurs nouvelles apparaissent, i.e. qu'elles ne soient pas héritées d'autres erreurs. Considérons l'automate présenté en figure 20.

La première garde $x_4 < x_3 + 2$ provoque une première erreur au niveau du coefficient correspondant à $x_2 - x_1$. La seconde garde $x_4 > x_3 + 1$ provoque pour la même raison (implicitement, nous avons $x_4 - x_3 = x_2 - x_1$) une erreur au niveau du coefficient correspondant cette fois à $x_1 - x_2$. Cette seconde erreur est totalement indépendante de la première, nous dirons que c'est une erreur "nouvelle". La garde qui l'a provoquée ($x_4 > x_3 + 1$) sera qualifiée d'instable.

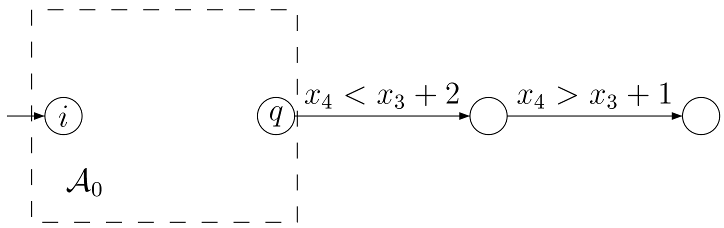


FIG. 20 – Apparition d'une erreur.

Erreur nouvelle et transition (garde) instable. En utilisant le vocabulaire précédemment introduit, nous considérons la transition t_l, dont nous supposons qu'elle n'est pas instable initiale. Nous supposons de plus que le coefficient (i, j) de la DBM M'_l est un petit coefficient erroné. Nous dirons que cette erreur est *nouvelle* si le coefficient (i, j) de la DBM associée à la zone *Post*$(\llbracket \overline{M_{l-1}} \rrbracket, t_l)$ est erroné. Nous dirons alors que t_l est une *transition instable* et que sa garde est une *garde instable*. Intuitivement, ceci signifie que même si toutes les erreurs précédentes sont corrigées, cette nouvelle erreur peut encore apparaître.

Il est facile de remarquer alors que la transition instable initiale est simplement la première des transitions instables.

Nous obtenons ainsi une famille de façons d'obtenir des erreurs. Malheureusement nous ne pouvons pas garantir le caractère exhaustif de cette famille.

8.2.2 Étude de l'algorithme de Floyd et historique des erreurs

Marquage des DBMs. Une méthode naturelle permettant d'étudier les coefficients erronés consiste alors à rajouter dans les DBMs des "marques" (ou *tags*) permettant de signaler les coefficients erronés. Nous considérerons dans la suite ces tags comme des formules du premier ordre sur un ensemble de variables. Par défaut, tous les tags sont étiquettés par "True". Il reste à définir comment se font les mises à jour de ces marques.

Nous considérons deux ensembles dénombrables de variables libres $(e_n)_{n\in\mathbb{N}}$ et $(g_n)_{n\in\mathbb{N}}$. Le premier sera utilisé pour signaler une erreur, tandis que le second marquera la présence de gardes diagonales. Nous associerons à toutes ces variables une garde diagonale, ou l'élément vide ε pour les erreurs dûes à l'origine à des gardes non diagonales. Nous considérons ensuite l'algorithme 6 mettant à jour les tags au cours du calcul de la forme normale d'une DBM.

Son fonctionnement est le suivant : si l'algorithme de Floyd trouve que le chemin entre i et j est plus court en passant par k, alors il met naturellement à jour la valeur de la distance entre i et j, mais également le tag associé à ce chemin. Pour cela, comme le nouveau chemin doit aller de i à k, puis de k à j, nous lui associons la conjonction des deux tags associés à ces deux chemins, car si le coefficient (i, k) est erroné, le nouveau coefficient (i, j) le sera probablement aussi. D'autre part, s'il trouve que le chemin est équivalent, alors afin de considérer l'ensemble des chemins donnant le poids minimal, nous ajoutons au tag $t_{i,j}$ le tag associé au chemin passant par k. Si les deux chemins sont erronés, alors nous utilisons une disjonction afin de transcrire le fait qu'une seule correction peut suffir. Dans le cas contraire nous utilisons une conjonction.

Algorithme 6 Floyd revisité

Entrée : $M = (\prec_{i,j}, m_{i,j}, t_{i,j})_{0 \le i,j \le n}$ une DBM marquée.
Sortie : M en forme normale avec les tags mis à jour
  Pour $i = 0$ à n **Faire**
    Pour $j = 0$ à n **Faire**
      Pour $k = 0$ à n **Faire**
        Si $(\prec_{i,k}, m_{i,k}) + (\prec_{k,j}, m_{k,j}) < (\prec_{i,j}, m_{i,j})$ **Alors**
          $(\prec_{i,j}, m_{i,j}) := (\prec_{i,k}, m_{i,k}) + (\prec_{k,j}, m_{k,j})$;
          $t_{i,j} := t_{i,k} \wedge t_{k,j}$;
        Sinon Si $(\prec_{i,k}, m_{i,k}) + (\prec_{k,j}, m_{k,j}) = (\prec_{i,j}, m_{i,j})$ **Alors**
          $t_{i,j} := t_{i,j} \vee (t_{i,k} \wedge t_{k,j})$; (si les deux chemins sont potentiellement erronés)
          $t_{i,j} := t_{i,j} \wedge (t_{i,k} \wedge t_{k,j})$; (si un des chemin peut ne pas être erroné)
        Fin Si
      Fin Pour
    Fin Pour
  Fin Pour

Il s'agirait par la suite de pouvoir déterminer de façon exacte si une erreur a été obtenue par propagation à partir d'une erreur précédente (erreur héritée), ou si c'est une garde instable qui vient de la créer (erreur nouvelle). Cela est en fait délicat car ces situations ne sont pas tout à fait disjointes. Une erreur peut en effet être à la fois héritée et nouvelle. Il faudrait dans ce cas lui associer les deux origines possibles et corriger ces deux sources. Ces problèmes ne sont pas encore éclaircis et nous espérons achever ce travail par la suite.

Nous envisageons pour cela de modifier la nature des tags afin d'ajouter une information quantitative sur l'erreur commise. Nous utiliserons alors des tags pour associer à toute variable d'erreur la différence entre la valeur erronée et la valeur exacte. Nous pouvons ainsi obtenir toutes les informations que nous désirons sur les chemins les plus courts entre deux points, en fonction de ce que l'on choisit de stocker dans les tags.

Pour l'instant, nous ne sommes pas parvenus à construire l'historique exact des erreurs, et devons nous limiter à une approximation. Pour cela, nous réalisons naturellement l'algorithme de Floyd modifié et prenons ainsi en compte l'ensemble des gardes diagonales ayant propagé une erreur. De plus, nous testons systématiquement si une erreur a également une origine nouvelle, et déterminons donc les gardes instables.

8.2.3 Méthode de choix

La construction faite précédemment permet de proposer une nouvelle méthode de choix. En parcourant le contre-exemple, nous construisons les DBMs marquées associées au calcul approché selon le chemin décrit par le contre-exemple erroné. À l'issue de ce calcul, nous déterminons le coefficient erroné, et la garde erronée associée. Nous retournons enfin l'ensemble des gardes diagonales associées à la marque de ce coefficient, ainsi que la garde erronée si celle-ci est diagonale.

Correction de cette méthode. Rappelons que dans notre cadre, la correction d'une méthode de choix est assurée si pour tout contre-exemple erroné, au moins une garde diagonale est sélectionnée. La description que nous avons faite de notre algorithme n'est pas encore assez précise pour permettre de démontrer formellement sa correction. Cependant, si les conditions énoncées ci-dessous sont vérifiées, alors la correction est asurée :

- garde instable (initiale ou non) : si celle-ci est diagonale, nous l'associons à toutes les erreurs qu'elle a créées.
- garde erronée : si celle-ci est diagonale, nous l'ajoutons à l'ensemble des gardes en cause.
- propagation : si une garde diagonale est intervenue dans la propagation d'une erreur, nous l'associons à l'erreur obtenue.

Avec cet algorithme, nous ne pouvons malheureusement pas garantir que le contre-exemple étudié va être éliminé lors du raffinement. Cependant, comme nous sélectionnons toujours au moins une garde à chaque itération, nous savons que ce contre-exemple va finir par être éliminé. Nous espérons par la suite pouvoir améliorer cette méthode de sorte à pouvoir garantir l'élimination directe du contre-exemple.

Chapitre 9

Récapitulatif des algorithmes proposés

Nous allons présenter dans ce chapitre plusieurs algorithmes afin de répondre au problème que nous nous sommes posé. Dans un premier temps, nous présentons des algorithmes reposant sur la "découpe" par rapport à toutes les gardes diagonales du système, puis ceux reposant sur la méthode par raffinements successifs.

9.1 Algorithmes basés sur la méthode naïve

Nous avons présenté en section 3.3 une méthode permettant de construire un automate temporisé équivalent sans garde diagonale. Nous avons mentionné que cette construction conduisait à un algorithme naïf constitué d'une phase de prétraitement durant laquelle toutes les gardes diagonales sont éliminées, puis de l'application de l'algorithme classique au nouvel automate construit. Nous donnons ci-dessous une présentation informelle de cet algorithme.

Algorithme 7 Analyse en avant des AT avec gardes diagonales avec prétraitement

---

Entrée : $\mathcal{A}$ un automate temporisé avec gardes diagonales
Sortie : $\mathcal{L}(\mathcal{A}) \neq \emptyset$?
  Déterminer $g_1, \ldots, g_n$ les gardes diagonales de $\mathcal{A}$;
  Construire $\mathcal{A}'$ obtenu en retirant successivement $g_1, \ldots, g_n$ à $\mathcal{A}$;
  Appliquer l'algorithme 2 à $\mathcal{A}'$ (avec k la constante maximale de $\mathcal{A}'$) ;
  Retouner la réponse de l'algorithme 2 ;

L'inconvénient de cette méthode est que la taille de $\mathcal{A}'$ vérifie :

$$\text{taille}(\mathcal{A}') = 2^n \times \text{taille}(\mathcal{A})$$

La complexité de l'algorithme est linéaire en le nombre de zones traitées et ce nombre est lui aussi multiplié au plus par le facteur 2^n. En effet, cette construction consiste à découper toutes les zones selon toutes les contraintes diagonales apparaissant. Remarquons cependant que notre but n'est pas réellement de retirer les gardes diagonales de l'automate, mais plutôt de retirer les sources possibles d'erreurs. Une solution consiste alors à forcer l'algorithme à faire attention, i.e. à ne jamais manipuler des zones ne satisfaisant pas une garde ni sa négation. Cela revient donc à exiger la propriété suivante lors de chaque étape de l'algorithme :

$$\forall Z \text{ intervenant dans le calcul}, \forall g \text{ (diagonale)}, Z \subseteq g \vee Z \subseteq \neg g \tag{1}$$

Chercher à vérifier la condition (1) revient au même que la construction rappelée plus haut, puisque chaque copie de l'automate correspondait à un cas ou à l'autre. D'un point de vue complexité, le nombre de zones traitées va donc être le même que celui obtenu avec l'algorithme naïf rappelé ci-dessus.

Une première implémentation incorrecte. Cette remarque a déjà été utilisée dans UPPAAL (Bengtsson et Yi [BY03]). Leur algorithme consiste à découper les zones en sous-zones de sorte à ce que l'équation (1) soit satisfaite. Malheureusement, le choix de leur constante d'extrapolation rend incorrect leur algorithme. En effet, ils extrapolent les zones selon la constante qu'ils ont calculée sans tenir compte des gardes diagonales du système. À cause de cela, les "régions" qu'ils considèrent ainsi n'ont pas les bonnes propriétés présentées en section 1.3. Il suffit de considérer le graphe suivant (pour deux horloges x et y).

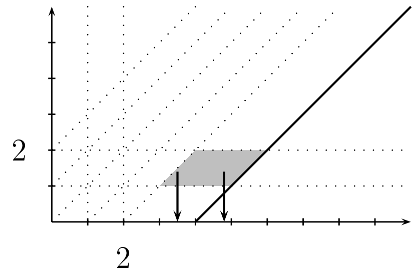


FIG. 21 – Une région ayant deux successeurs après une remise à zéro.

Dans ce cas le problème est facile à régler puisqu'il suffit de prendre en compte dans le calcul de la constante d'extrapolation les valeurs des gardes diagonales apparaissant dans l'automate.

Une implémentation économique. Une autre solution, qui se rapproche plus de la première, consiste à faire une phase de prétraitement, mais de façon économique. Ainsi, cela consiste à réaliser le même type de construction, de sorte à toujours s'assurer que les zones manipulées vérifient la propriété (1), mais nous allons laisser les gardes diagonales présentes, pour éviter d'avoir à stocker 2^n copies de l'automate. La figure suivante illustre la construction.

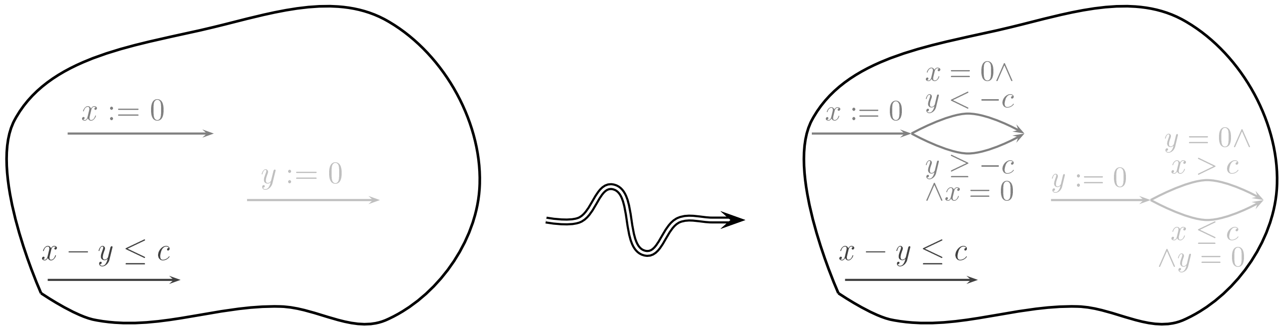


FIG. 22 – Modifications selon la garde diagonale $x - y \leq c$

Du point de vue de la taille de la nouvelle structure, nous voyons que pour chaque garde diagonale, et chaque remise à zéro d'une des horloges apparaissant dans cette garde, nous créons

un nouvel état et deux nouvelles transitions. Le nouvel automate obtenu est donc d'une taille linéaire en celle de l'ancien automate.
Considérons à présent l'algorithme associé qui dans une première phase de prétraitement modifie l'automate selon cette technique, puis dans une deuxième phase lui applique l'algorithme 2. Du point de vue de la complexité, si nous observons l'arbre de dépliage associé au parcours à la volée en avant de l'algorithme, nous pouvons constater que l'arbre obtenu est en bijection avec l'arbre obtenu selon la méthode naïve (les zones sont les mêmes, seuls les noms des états changent puisqu'il y a une seule copie). Le nombre et le type des objets manipulés sont donc les mêmes. A fortiori, les complexités en temps et en espace seront les mêmes pour la deuxième phase de l'algorithme. La première phase quant à elle doit avoir une complexité en temps similaire, mais une complexité en espace bien moindre (taille($\mathcal{A}$) contre $2^n\times$ taille($\mathcal{A}$)) .

Les méthodes que nous venons de présenter traitent toutes les gardes diagonales, sans aucune distinction. Pourtant, toutes les diagonales ne sont pas gênantes : comment n'éliminer que des gardes diagonales gênantes ? Nous allons à présent nous intéresser à des algorithmes qui n'effectuent les transformations que pour certaines gardes diagonales.

9.2 Algorithmes basés sur la méthode par raffinements successifs

Nous avons présenté dans la section 4.2 la méthode par raffinements successifs. Nous avons vu alors comment réaliser le test d'un contre-exemple, ainsi que le raffinement du modèle. Enfin, nous avons présenté dans le chapitre précédent plusieurs méthodes permettant de construire un ensemble de gardes diagonales par rapport auquel raffiner. Nous rappelons ici l'algorithme d'accessibilité général correspondant à cette méthode (algorithme 8).

Algorithme 8 Analyse en avant avec raffinements successifs

Entrée : $\mathcal{A}$ un automate temporisé
Sortie : $\mathcal{L}(\mathcal{A}) = \emptyset$?
Tant Que True **Faire**
Appliquer l'algorithme 2 à $\mathcal{A}$;
Si l'algorithme classique répond que $\mathcal{L}(\mathcal{A}) = \emptyset$ **Alors**
Retourner $\mathcal{L}(\mathcal{A}) = \emptyset$;
Sinon
Récupérer le contre-exemple ce retourné par l'algorithme 2 ;
Tester ce ;
Si ce se relève en un vrai contre-exemple ce' **Alors**
Retourner $\mathcal{L}(\mathcal{A}) \neq \emptyset$ accompagné de la preuve ce' ;
Sinon
$\mathcal{G}$:= Choixdesgardes(ce) ;
$\mathcal{A}$:= Raffinement ($\mathcal{A}, \mathcal{G}$) ;
Fin Si
Fin Si
Fin Tant Que

Comme nous l'avons expliqué dans la section 4.2, la seule condition nécessaire pour assurer la terminaison de l'algorithme 8 est que pour tout contre-exemple erroné, au moins une garde diagonale soit sélectionnée par l'algorithme de choix. Nous allons proposer ici plusieurs méthodes possibles pour cette phase de l'algorithme. Notre présentation n'est pas exhaustive et nos choix discutables. Nous motiverons donc les différentes heuristiques sous-jacentes.

Méthodes naïves. Un premier point de vue consiste à penser qu'il faut que la phase de choix prenne peu de temps, et qu'il est préférable d'avoir une méthode de choix très simple, et surtout peu coûteuse en terme de complexité. Dans ce cadre, nous ne cherchons pas à trouver des gardes responsables. Nous pouvons citer plusieurs méthodes naïves :

- tirer au hasard une garde diagonale parmi l'ensemble des gardes diagonales apparaissant sur le contre-exemple erroné,
- prendre la première garde diagonale rencontrée sur le contre-exemple erroné,
- prendre l'ensemble des gardes diagonales apparaissant sur le contre-exemple erroné.

Méthodes basées sur des gardes responsables. Bien que ces choix présentent l'avantage d'être très rapides à effectuer, il semble tout de même préférable de chercher une garde en cause dans l'erreur de l'algorithme. Pour cela, nous pouvons utiliser les méthodes développées dans le chapitre 8. Rappelons que nous avons présenté un critère de "dédiagonalisation" et réalisé un pistage des erreurs afin de développer ces méthodes. Nous en faisons un rappel ci-dessous :

• Critère de "dédiagonalisation" :

- méthode rapide : prendre la première garde diagonale susceptible de ne pas être dédiagonalisable,
- méthode exhaustive : construire un ensemble de gardes diagonales assurant l'élimination du contre-exemple erroné,
- méthode intermédiaire : construire un ensemble de gardes diagonales permettant de corriger sensiblement les calculs de l'algorithme 2.

• Pistage des erreurs :

nous avons présenté une méthode permettant, en déterminant l'origine des erreurs, de tenir à jour au cours du parcours du contre-exemple erroné pour chaque coefficient erroné un ensemble de gardes coupables. Cette méthode n'est toutefois pas encore aboutie.

Pour conclure ce chapitre, faisons une comparaison rapide des deux types d'algorithmes proposés. Tout d'abord, nous avons expliqué que des "bugs" comme celui présenté nous semblent rares. Il est donc possible que l'algorithme 1 retourne une réponse correcte. Ceci justifie le fait qu'il parait plus judicieux de ne retirer des gardes diagonales que si c'est nécessaire. D'autre part, observons le comportement de l'algorithme dans le cas le pire. Nous somment alors amenés à retirer toutes les gardes diagonales. Nous devons donc appliquer l'algorithme 2 à ce nouvel automate et obtenons alors la complexité des méthodes naïves. Les calculs précédents cette dernière étape étant de complexité négligeable devant l'application de l'algorithme 2 sur une instance de taille $2^n \times \text{taille}(\mathcal{A})$, nous obtenons donc une complexité totale comparable.

Conclusion

L'objet de ce travail était de comprendre le rôle des gardes diagonales vis-à-vis de l'opérateur d'extrapolation afin de mieux saisir le "bug" de l'algorithme 2 et de proposer des algorithmes corrects pour tester l'accessiblité dans les automates temporisés avec gardes diagonales. Ceci semble naturel afin de pouvoir décrire les systèmes réels dans le modèle plus riche des automates temporisés avec gardes diagonales.

Nous avons commencé par présenter précisément les limites de notre problème en nous appuyant sur l'étude menée dans [Bou04]. Nous avons alors vu pourquoi ce "bug" était dû aux gardes diagonales, et comment la méthode de retrait des gardes diagonales de [BDGP98] donne une première solution naïve insatisante du point de vue de la complexité.

L'étude de méthodes simples et naturelles a ensuite permis de motiver notre choix : utiliser une méthode par raffinements successifs. Pour cela, nous avons présenté les outils nécessaires, dont la structure de données des DBMs très utilisée dans l'implémentation d'outils basés sur des systèmes temporisés.

Nous avons alors pu mener une étude précise de la structure des contre-exemples erronés et chercher à dégager des propriétés des erreurs commises par l'algorithme 2. Nous avons ensuite montré l'existence de coefficients erronés dans les calculs réalisés par celui-ci. La grande surprise est provenue du fait que ces erreurs pouvaient également apparaître en absence de gardes diagonales. Nous avons alors cherché à développer des méthodes permettant de faire la disctinstion entre les deux cadres. Pour cela nous avons introduit un critère de "dédiagonalisation" ainsi qu'un historique des erreurs. Ces nouvelles notions nous ont permis par la suite de proposer plusieurs algorithmes répondant au problème posé initialement.

Nous n'avons cependant pas pu tirer de la notion d'historique des erreurs la caractérisation que nous souhaitions. Nous espérons par la suite pouvoir préciser le comportement de l'algorithme 2 et raffiner notre algorithme dans ce cas.

D'autre part, les algorithmes présentés ici pourraient être implantés dans un outil comme CMC. Ceci paraît tout à fait réalisable et constitue un premier pas nécessaire afin de pouvoir les comparer. En effet, ils pourraient ensuite être testés sur des cas réels afin d'obtenir des comparaisons en terme d'efficacité. Malheureusement il n'existe pas d'automates de référence pour tester ce type d'algorithme.

Pour conclure, il est essentiel d'insister sur le fait que les "bugs" de l'algorithme 2 sont rares, et que peu de gardes diagonales sont en cause. Les méthodes naïves raffinant systématiquement toutes les gardes diagonales de l'automate sont donc à exclure fermement. Il paraît plus judicieux de ne retirer que les gardes diagonales réellement gênantes.

Bibliographie

[AD90] Rajeev Alur and David Dill. Automata for modeling real-time systems. In *Proc. 17th International Colloquium on Automata, Languages and Programming (ICALP'90)*, volume 443 of *Lecture Notes in Computer Science*, pages 322–335. Springer, 1990.

[AD94] Rajeev Alur and David Dill. A theory of timed automata. *Theoretical Computer Science (TCS)*, 126(2) :183–235, 1994.

[AL02] Luca Aceto and François Laroussinie. Is your model-checker on time ? on the complexity of model-checking for timed modal logics. *Journal of Logic and Algebraic Programming (JLAP)*, 52–53 :7–51, 2002.

[BBP02] Béatrice Bérard, Patricia Bouyer, and Antoine Petit. Analysing the pgm protocol with uppaal. In *Proc. 2nd Workshop on Real-Time Tools (RT-TOOLS 2002)*. Federated Logic Conference (FLoC), Copenhagen, Denmark, 2002.

[BDGP98] Béatrice Bérard, Volker Diekert, Paul Gastin, and Antoine Petit. Characterization of the expressive power of silent transitions in timed automata. *Fundamenta Informaticae*, 36(2) :145–182, 1998.

[BDM+98] Marius Bozga, Conrado Daws, Oded Maler, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. KRONOS : a model-checking tool for real-time systems. In *Proc. 10th International Conference on Computer Aided Verification (CAV'98)*, volume 1427 of *Lecture Notes in Computer Science*, pages 546–550. Springer, 1998.

[Bou04] Patricia Bouyer. Forward analysis of updatable timed automata. *Formal Methods in System Design*, 24(3) :281–320, 2004.

[BY03] Johan Bengtsson and Wang Yi. On clock difference constraints and termination in reachability of timed automata. In *Formal Methods and Software Engineering, 5th International Conference on Formal Engineering Methods (ICFEM'03)*, 2003.

[CGJ+00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Proc. of Computer Aided Verification (CAV'00)*, 2000.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. Model checking. *The MIT Press, Cambridge, Massachusetts*, 1999.

[HHWT97] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HYTECH : A model-checker for hybrid systems. *Journal on Software Tools for Technology Transfer (STTT)*, 1(1–2) :110–122, 1997.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70, 2002.

[HSLL97] Klaus Havelund, Arne Skou, Kim G. Larsen, and Kristian Lund. Formal modelling and analysis of an audio/video protocol : An industrial case study using uppaal. In *Proceedings of the 18th Symposium on Real-Time Systems (RTS'97)*, pages 2–13. IEEE Comp. Soc. Press, 1997.

[LL98] François Laroussinie and Kim G. Larsen. CMC : A tool for compositional model-checking of real-time systems. In *Proc. IFIP Joint International Conference on Formal Description Techniques & Protocol Specification, Testing, and Verification (FORTE-PSTV'98)*, pages 439–456. Kluwer Academic, 1998.

[LLPY97] Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Efficient verification of real-time systems : Compact data structure and state-space reduction. In *Proc. 18th IEEE Real-Time Systems Symposium (RTSS'97)*, pages 14–24. IEEE Computer Society Press, 1997.

Annexe A

Calculs de CMC

Nous présentons dans cette annexe les codes sources des instructions des programmes CMC utilisés, ainsi que les sorties de CMC associées.

Définition de l'automate $\mathcal{C}$ page 18

```
Automata : ce  [
Clocks : x1,x2,x3,x4
Nodes : i, q1 , q2, q3, q4, q5, q, error
Transitions :
  i  -t1,       x3<=3,    {x1,x3}  -> q1,
  q1 -t2,       x2=3,       {x2}   -> q2,
  q2 -t3,       x1=2,       {x1}   -> q3,
  q3 -t4,       x2=2,       {x2}   -> q2,
  q2 -t5,       x1=2,       {x1}   -> q4,
  q4 -t6,       x2=2,       {x2}   -> q5,
  q5 -t7,       x1=3,       {x1}   -> q,
  q  -t,    x2<x1+2 ^ x4>x3+2    -> error

];

Table : T  {
        (ce) ;
        (t1) -> t1 ;
        (t2) -> t2 ;
        (t3) -> t3 ;
        (t4) -> t4 ;
        (t5) -> t5 ;
        (t6) -> t6 ;
        (t7) -> t7 ;
        (t)  -> t
        };

SReach(T,(ce:error));
```

Sortie associée

```
Flag : verbose = 0

% *

%
% System Reachability.
------------------------------------------
La constante maximale du système, par rapport à
laquelle on va extrapoler, est 3.
Configuration de depart:
L'état initial est le vecteur :(ce:i)
La zone est : x3 = 0 ; x2 = 0 ; x1 = 0 ; x4-x3<=0  ;
x4-x2 = 0; x4-x1 = 0; x3-x2<=0; x3-x1 = 0; x2-x1<=0;
La formule à vérifier est : (ce:error )

Voici le chemin associé au contre-exemple détecté :
(t1)
(t2)
(t5)
(t6)
(t7)
(t)

Le nombre d'éléments de Passed est 12
Proposition vérifiée !!
le calcul a duré 1 seconde(s).
```

Définition de l'automate $\mathcal{A}_2$ page 25

```
Automata : A2  [
Clocks : x1,x2,x3,x4
Nodes : i, q1, q2, q3, q4, q5, q,
q6, q7, error
Transitions :

#Partie associée à l'automate Cex
  i  -t1,   x3<=3,   {x1,x3}  -> q1,
  q1 -t2,   x2=3,     {x2}   -> q2,
  q2 -t3,   x1=2,     {x1}   -> q3,
  q3 -t4,   x2=2,     {x2}   -> q2,
  q2 -t5,   x1=2,     {x1}   -> q4,
  q4 -t6,   x2=2,     {x2}   -> q5,
  q5 -t7,   x1=3,     {x1}   -> q,


#Partie liée à la partie basse
  i  -u1, 1<=x1^x1<=3,{x1,x2,x3}-> q6,
  q6 -u2,   x1=1,     {x1,x2}  -> q6,
  q6 -u3, x1=0^x3>3            -> q7,
  q7 -u4, 1<=x1^x1<=3,   {x1}   -> q,


#la fameuse transition erronée
  q  -t, x2>x1+2 ^ x4<x3+2    -> error

];


Table : T  {
        (A2) ;
        (t1) -> t1 ;
        (t2) -> t2 ;
        (t3) -> t3 ;
        (t4) -> t4 ;
        (t5) -> t5 ;
        (t6) -> t6 ;
        (t7) -> t7 ;

        (u1) -> u1 ;
        (u2) -> u2 ;
        (u3) -> u3 ;
        (u4) -> u4 ;

        (t)  -> t
        };


SReach(T,(A2:error));
```

Sortie associée

```
% load "A2";
% System Reachability.
-----------------------------------------
La constante maximale du système, par rapport
à laquelle on va extrapoler, est 3
Configuration de depart:
L'état initial est le vecteur :(A2:i)
La zone est : x3 = 0 ; x2 = 0 ; x1 = 0; x4-x3<=0;
x4-x2 = 0; x4-x1 = 0; x3-x2<=0; x3-x1 = 0; x2-x1<=0;
La formule à vérifier est : (A2:error )

Voici le chemin associé au contre-exemple détecté :
(t1)
(t2)
(t5)
(t6)
(t7)
(t)
Le nombre d'éléments de Passed est 17
Proposition vérifiée !!
le calcul a duré 1 seconde(s).

% ControlAbstraction:=1;
Flag : ControlAbstraction = 1
% Cmax:=7;
Flag : Cmax = 7
% load "A2";
% System Reachability.
-----------------------------------------
Vous avez choisi d'extrapoler par rapport à la
constante Cmax qui est 7
Configuration de depart:
L'état initial est le vecteur :(A2:i)
La zone est : x3 = 0 ; x2 = 0 ; x1 = 0 ; x4-x3<=0;
x4-x2 = 0; x4-x1 = 0; x3-x2<=0; x3-x1 = 0; x2-x1<=0;
La formule à vérifier est : (A2:error )

Voici le chemin associé au contre-exemple détecté :
(u1)
(u2)
(u2)
(u2)
(u2)
(u3)
(u4)
(t)
Le nombre d'éléments de Passed est 30
Proposition vérifiée !!
le calcul a duré 1 seconde(s).
```

Instructions de calculs associés à la figure 12 page 37

```
#Fichier ch_errone

K:=3;

M0 := Zone((x1=0 ^ x2=0 ^ x3=0 ^ x4=0));

M1:=Post(M0,(x3<=3),{x1,x3});

M2:=Post(M1,(x2=3),{x2});

M3:=Post(M2,(x1=2),{x1});

M4:=Post(M3,(x2=2),{x2});

M5:=Post(M4,(x1=3),{x1});

M6:=Post(M5,(x4>x3+2),{});

M7:=Post(M6,(x2<2),{});


Mprime1:=
Extra(K,Post(M0,(x3<=3),{x1,x3}));

Mprime2:=
Extra(K,Post(Mprime1,(x2=3),{x2}));

Mprime3:=
Extra(K,Post(Mprime2,(x1=2),{x1}));

Mprime4:=
Extra(K,Post(Mprime3,(x2=2),{x2}));

Mprime5:=
Extra(K,Post(Mprime4,(x1=3),{x1}));

Mprime6:=
Extra(K,Post(Mprime5,(x4>x3+2),{}));

Mprime7:=
Extra(K,Post(Mprime6,(x2<2),{}));
```

Sortie associée

```
Hybrid and Compositional  Model Checking
http://www.lsv.ens-cachan.fr/~fl/
version 1.0 b - (DBM)
% load "ch_errone";
K : (int) 3

% M6 : 5<= x3  ; 2< x4-x3<=3  ;
 x4-x2 = 5 ; x3-x1 = 5 ;

     0     x4    x3    x2    x1
0    <=0   <-7   <=-5  <-2   <=0
x4   <inf  <=0   <=3   <=5   <=8
x3   <inf  <-2   <=0   <3    <=5
x2   <inf  <=-5  <=-2  <=0   <=3
x1   <inf  <-7   <=-5  <-2   <=0


% M7 : empty zone

empty zone


% Mprime6 : 2< x4-x3<=3  ; 2<= x3-x2 ;
 3< x3-x1 ; 1<= x2-x1<=3  ;

     0     x4    x3    x2    x1
0    <=0   <-3   <-3   <=-1  <=0
x4   <inf  <=0   <=3   <inf  <inf
x3   <inf  <-2   <=0   <inf  <inf
x2   <inf  <-3   <=-2  <=0   <=3
x1   <inf  <-3   <-3   <=-1  <=0


% Mprime7 : x2 <2  ; 2< x4-x3<=3  ;
 2<= x3-x2 ; 3< x3-x1 ; 1<= x2-x1 ;

     0     x4    x3    x2    x1
0    <=0   <-3   <-3   <=-1  <=0
x4   <inf  <=0   <=3   <inf  <inf
x3   <inf  <-2   <=0   <inf  <inf
x2   <2    <-3   <=-2  <=0   <2
x1   <1    <-3   <-3   <=-1  <=0
```

Instructions de calculs associés à la figure 14 page 39

```
#Fichier ch_instable


K:=3;

M0 := Zone((x1=0 ^ x2=0 ^ x3=0 ^ x4=0));

M1:=Post(M0,(x3<=3),{x1,x3});

M2:=Post(M1,(x2=3),{x2});

M3:=Post(M2,(x1=2),{x1});

M4:=Post(M3,(x2=2),{x2});

M5:=Post(M4,(x1=3),{x1});

M6:=Post(M5,(x2<2),{});

M7:=Post(M6,(x4>x3+2),{});




Mprime1:=
Extra(K,Post(M0,(x3<=3),{x1,x3}));

Mprime2:=
Extra(K,Post(Mprime1,(x2=3),{x2}));

Mprime3:=
Extra(K,Post(Mprime2,(x1=2),{x1}));

Mprime4:=
Extra(K,Post(Mprime3,(x2=2),{x2}));

Mprime5:=
Extra(K,Post(Mprime4,(x1=3),{x1}));

Mprime6:=
Extra(K,Post(Mprime5,(x2<2),{}));

Mprime7:=
Extra(K,Post(Mprime6,(x4>x3+2),{}));
```

Sortie associée

```
Hybrid and Compositional  Model Checking
http://www.lsv.ens-cachan.fr/~fl/
version 1.0 b - (DBM)
% load "ch_instable";
K : (int) 3


% M5 : 6<= x4 <=8  ; 5<= x3  ; x1 = 0;
 x4-x2 = 5 ; x3-x1<=5  ;

     0     x4    x3    x2    x1
0    <=0   <=-6  <=-5  <=-1  <=0
x4   <=8   <=0   <=3   <=5   <inf
x3   <=5   <=-1  <=0   <=4   <inf
x2   <=3   <=-5  <=-2  <=0   <=3
x1   <=0   <-3   <-3   <=-1  <=0

% M6 : x4 <7  ; 5<= x3  ; 1<= x4-x3 ;
 x4-x2 = 5 ; x3-x1 = 5 ;

     0     x4    x3    x2    x1
0    <=0   <=-6  <=-5  <=-1  <=0
x4   <7    <=0   <2    <=5   <7
x3   <6    <=-1  <=0   <=4   <=5
x2   <2    <=-5  <-3   <=0   <2
x1   <1    <=-6  <=-5  <=-1  <=0

% M7 : empty zone

empty zone

% Mprime5 : 3< x3  ; 1<= x2 <=3  ; x1 = 0 ;
 1<= x4-x3<=3  ; 3< x4-x2 ; 2<= x3-x2 ;

     0     x4    x3    x2    x1
0    <=0   <-3   <-3   <=-1  <=0
x4   <inf  <=0   <=3   <inf  <inf
x3   <inf  <=-1  <=0   <inf  <inf
x2   <=3   <-3   <=-2  <=0   <=3
x1   <=0   <-3   <-3   <=-1  <=0

% Mprime6 : x2 <2  ; 1<= x4-x3<=3  ;
 3< x4-x2 ; 2<= x3-x2 ; 3< x3-x1 ; 1<= x2-x1 ;

     0     x4    x3    x2    x1
0    <=0   <-3   <-3   <=-1  <=0
x4   <inf  <=0   <=3   <inf  <inf
x3   <inf  <=-1  <=0   <inf  <inf
x2   <2    <-3   <=-2  <=0   <2
x1   <1    <-3   <-3   <=-1  <=0

% Mprime7 : 2< x4-x3<=3  ; 2<= x3-x2 ;
 3< x3-x1 ; 1<= x2-x1<2  ;

     0     x4    x3    x2    x1
0    <=0   <-3   <-3   <=-1  <=0
x4   <inf  <=0   <=3   <inf  <inf
x3   <inf  <-2   <=0   <inf  <inf
x2   <inf  <-3   <=-2  <=0   <2
x1   <inf  <-3   <-3   <=-1  <=0
```

Instructions de calculs associés à la figure 15 page 40

```
#Fichier ch_lesdeux


K:=3;

M0 := Zone((x1=0 ^ x2=0 ^ x3=0 ^ x4=0));

M1:=Post(M0,(x3<=3),{x1,x3});

M2:=Post(M1,(x2=3),{x2});

M3:=Post(M2,(x1=2),{x1});

M4:=Post(M3,(x2=2),{x2});

M5:=Post(M4,(x1=3),{x1});

M6:=Post(M5,(x2 <3),{});

M7:=Post(M6,(x4>x3+2),{});

M8:=Post(M7,(x2<2),{});



Mprime1:=
Extra(K,Post(M0,(x3<=3),{x1,x3}));

Mprime2:=
Extra(K,Post(Mprime1,(x2=3),{x2}));

Mprime3:=
Extra(K,Post(Mprime2,(x1=2),{x1}));

Mprime4:=
Extra(K,Post(Mprime3,(x2=2),{x2}));

Mprime5:=
Extra(K,Post(Mprime4,(x1=3),{x1}));

Mprime6:=
Extra(K,Post(Mprime5,(x2<3),{}));

Mprime7:=
Extra(K,Post(Mprime6,(x4>x3+2),{}));

Mprime8:=
Extra(K,Post(Mprime7,(x2<2),{}));
```

Sortie associée

```
Hybrid and Compositional  Model Checking
http://www.lsv.ens-cachan.fr/~fl/
version 1.0 b - (DBM)
% load "ch_lesdeux";
K : (int) 3


% M6 : x4 <8  ; 5<= x3  ; 1<= x4-x3 ;
 x4-x2 = 5 ; x3-x1 = 5 ;

     0     x4    x3    x2    x1
0    <=0   <=-6  <=-5  <=-1  <=0
x4   <8    <=0   <3    <=5   <8
x3   <7    <=-1  <=0   <=4   <=5
x2   <3    <=-5  <-2   <=0   <3
x1   <2    <=-6  <=-5  <=-1  <=0

% M7 : 5<= x3  ; 2< x4-x3<3  ; x4-x2 = 5 ;
 x3-x1 = 5 ;

     0     x4    x3    x2    x1
0    <=0   <-7   <=-5  <-2   <=0
x4   <inf  <=0   <3    <=5   <8
x3   <inf  <-2   <=0   <3    <=5
x2   <inf  <=-5  <-2   <=0   <3
x1   <inf  <-7   <=-5  <-2   <=0

% M8 : empty zone

empty zone

% Mprime6 : x2 <3  ; 1<= x4-x3<=3  ; 3< x4-x2;
 2<= x3-x2 ; 3< x3-x1 ; 1<= x2-x1 ;

     0     x4    x3    x2    x1
0    <=0   <-3   <-3   <=-1  <=0
x4   <inf  <=0   <=3   <inf  <inf
x3   <inf  <=-1  <=0   <inf  <inf
x2   <3    <-3   <=-2  <=0   <3
x1   <2    <-3   <-3   <=-1  <=0

% Mprime7 : 2< x4-x3<=3  ; 2<= x3-x2 ;
 3< x3-x1 ; 1<= x2-x1<3  ;

     0     x4    x3    x2    x1
0    <=0   <-3   <-3   <=-1  <=0
x4   <inf  <=0   <=3   <inf  <inf
x3   <inf  <-2   <=0   <inf  <inf
x2   <inf  <-3   <=-2  <=0   <3
x1   <inf  <-3   <-3   <=-1  <=0

% Mprime8 : x2 <2  ; 2< x4-x3<=3  ; 2<= x3-x2;
 3< x3-x1 ; 1<= x2-x1 ;

     0     x4    x3    x2    x1
0    <=0   <-3   <-3   <=-1  <=0
x4   <inf  <=0   <=3   <inf  <inf
x3   <inf  <-2   <=0   <inf  <inf
x2   <2    <-3   <=-2  <=0   <2
x1   <1    <-3   <-3   <=-1  <=0
```

Présentation des résultats associés aux trois exemples de dédiagonalisation de la section 8.1.2.

Les instructions ne sont pas présentées, elles sont similaires à celles vues dans les exemples précédents.

Exemple 1 :

```
% M6 : x4 <7  ; 5<= x3  ; 1<= x4-x3 ; x4-x2 = 5 ;
 x3-x1 = 5 ;

        0       x4      x3      x2      x1
0       <=0     <=-6    <=-5    <=-1    <=0
x4      <7      <=0     <2      <=5     <7
x3      <6      <=-1    <=0     <=4     <=5
x2      <2      <=-5    <-3     <=0     <2
x1      <1      <=-6    <=-5    <=-1    <=0

% Mprime6 : x2 <2  ; 1<= x4-x3<=3  ; 3< x4-x2 ;
 2<= x3-x2 ; 3< x3-x1 ; 1<= x2-x1 ;

        0       x4      x3      x2      x1
0       <=0     <-3     <-3     <=-1    <=0
x4      <inf    <=0     <=3     <inf    <inf
x3      <inf    <=-1    <=0     <inf    <inf
x2      <2      <-3     <=-2    <=0     <2
x1      <1      <-3     <-3     <=-1    <=0

% W6 : x4 <7  ; 5<= x3  ; 1<= x4-x3 ; x4-x2 = 5 ;
 x3-x1 = 5 ;

        0       x4      x3      x2      x1
0       <=0     <=-6    <=-5    <=-1    <=0
x4      <7      <=0     <2      <=5     <7
x3      <6      <=-1    <=0     <=4     <=5
x2      <2      <=-5    <-3     <=0     <2
x1      <1      <=-6    <=-5    <=-1    <=0

% Wprime6 : x2 <2  ; 1<= x4-x3<=3  ; 3< x4-x2 ;
 2<= x3-x2 ; 3< x3-x1 ; 1<= x2-x1 ;

        0       x4      x3      x2      x1
0       <=0     <-3     <-3     <=-1    <=0
x4      <inf    <=0     <=3     <inf    <inf
x3      <inf    <=-1    <=0     <inf    <inf
x2      <2      <-3     <=-2    <=0     <2
x1      <1      <-3     <-3     <=-1    <=0
```

Exemple 2 :

```
% M6 : empty zone

empty zone
% Mprime6 : 2< x4-x3<=3  ; 2<= x3-x2 ; 3< x3-x1 ;
 1<= x2-x1<2  ;

        0       x4      x3      x2      x1
0       <=0     <-3     <-3     <=-1    <=0
x4      <inf    <=0     <=3     <inf    <inf
x3      <inf    <-2     <=0     <inf    <inf
x2      <inf    <-3     <=-2    <=0     <2
x1      <inf    <-3     <-3     <=-1    <=0

% W6 : empty zone

empty zone
```

```
% Wprime6 : 2< x4-x3<=3  ; 3< x3-x2 ; 1<= x2-x1<2  ;

        0       x4      x3      x2      x1
0       <=0     <-3     <-3     <=-1    <=0
x4      <inf    <=0     <=3     <inf    <inf
x3      <inf    <-2     <=0     <inf    <inf
x2      <inf    <-3     <-3     <=0     <2
x1      <inf    <-3     <-3     <=-1    <=0
```

Exemple 3 :

```
% M6 : empty zone

empty zone
% Mprime6 : 2< x4-x3<=3  ; 2<= x3-x2 ; 3< x3-x1 ;
 1<= x2-x1<2  ;

        0       x4      x3      x2      x1
0       <=0     <-3     <-3     <=-1    <=0
x4      <inf    <=0     <=3     <inf    <inf
x3      <inf    <-2     <=0     <inf    <inf
x2      <inf    <-3     <=-2    <=0     <2
x1      <inf    <-3     <-3     <=-1    <=0

% W6 : empty zone

empty zone
% Wprime6 : empty zone

empty zone
```

Instruction des calculs associés à la figure 18 page 47

```
#Fichier propagation


K:=3;

M0 := Zone((x1=0 ^ x2=0 ^ x3=0 ^ x4=0));

M1:=Post(M0,(x3<=3),{x1,x3});

M2:=Post(M1,(x2=3),{x2});

M3:=Post(M2,(x1=2),{x1});

M4:=Post(M3,(x2=2),{x2});

M5:=Post(M4,(x1=3),{x1});

M6:=Post(M5,(x4<x3+2),{});

M7:=Post(M6,(x1=0),{});



Mprime1:=
Extra(K,Post(M0,(x3<=3),{x1,x3}));

Mprime2:=
Extra(K,Post(Mprime1,(x2=3),{x2}));

Mprime3:=
Extra(K,Post(Mprime2,(x1=2),{x1}));

Mprime4:=
Extra(K,Post(Mprime3,(x2=2),{x2}));

Mprime5:=
Extra(K,Post(Mprime4,(x1=3),{x1}));

Mprime6:=
Extra(K,Post(Mprime5,(x4<x3+2),{}));

Mprime7:=
Extra(K,Post(Mprime6,(x1=0),{}));
```

Sortie associée

```
Hybrid and Compositional  Model Checking
http://www.lsv.ens-cachan.fr/~fl/
version 1.0 b - (DBM)
% load "propagation";
K : (int) 3

% M6 : 5<= x3  ; 1<= x4-x3<2  ; x4-x2 = 5;
 x3-x1 = 5 ;

     0     x4    x3    x2    x1
0    <=0   <=-6  <=-5  <=-1  <=0
x4   <inf  <=0   <2    <=5   <7
x3   <inf  <=-1  <=0   <=4   <=5
x2   <inf  <=-5  <-3   <=0   <2
x1   <inf  <=-6  <=-5  <=-1  <=0


% M7 : 6<= x4 <7  ; 5<= x3  ; x1 = 0 ;
 x4-x2 = 5 ; x3-x1<=5  ;

     0     x4    x3    x2    x1
0    <=0   <=-6  <=-5  <=-1  <=0
x4   <7    <=0   <2    <=5   <7
x3   <=5   <=-1  <=0   <=4   <=5
x2   <2    <=-5  <-3   <=0   <2
x1   <=0   <=-6  <=-5  <=-1  <=0


% Mprime6 : 1<= x4-x3<2  ; 3< x4-x2 ;
 2<= x3-x2 ; 3< x3-x1 ; 1<= x2-x1<=3  ;

     0     x4    x3    x2    x1
0    <=0   <-3   <-3   <=-1  <=0
x4   <inf  <=0   <2    <inf  <inf
x3   <inf  <=-1  <=0   <inf  <inf
x2   <inf  <-3   <=-2  <=0   <=3
x1   <inf  <-3   <-3   <=-1  <=0


% Mprime7 : 3< x3  ; 1<= x2 <=3  ; x1 = 0 ;
 1<= x4-x3<2  ; 3< x4-x2 ; 2<= x3-x2 ;

     0     x4    x3    x2    x1
0    <=0   <-3   <-3   <=-1  <=0
x4   <inf  <=0   <2    <inf  <inf
x3   <inf  <=-1  <=0   <inf  <inf
x2   <=3   <-3   <=-2  <=0   <=3
x1   <=0   <-3   <-3   <=-1  <=0
```

Instruction des calculs associés à la figure 19 page 47

```
#Fichier disparition

K:=3;

M0 := Zone((x1=0 ^ x2=0 ^ x3=0 ^ x4=0));

M1:=Post(M0,(x3<=3),{x1,x3});

M2:=Post(M1,(x2=3),{x2});

M3:=Post(M2,(x1=2),{x1});

M4:=Post(M3,(x2=2),{x2});

M5:=Post(M4,(x1=3),{x1});

M6:=Post(M5,(x4<x3+2),{});

M7:=Post(M6,(x2<x1+2),{});

Mprime1:=
Extra(K,Post(M0,(x3<=3),{x1,x3}));

Mprime2:=
Extra(K,Post(Mprime1,(x2=3),{x2}));

Mprime3:=
Extra(K,Post(Mprime2,(x1=2),{x1}));

Mprime4:=
Extra(K,Post(Mprime3,(x2=2),{x2}));

Mprime5:=
Extra(K,Post(Mprime4,(x1=3),{x1}));

Mprime6:=
Extra(K,Post(Mprime5,(x4<x3+2),{}));

Mprime7:=
Extra(K,Post(Mprime6,(x2<x1+2),{}));
```

Sortie associée

```
Hybrid and Compositional  Model Checking
http://www.lsv.ens-cachan.fr/~fl/
version 1.0 b - (DBM)
% load "disparition";
K : (int) 3

% M6 : 5<= x3  ; 1<= x4-x3<2  ; x4-x2 = 5;
 x3-x1 = 5 ;

     0     x4    x3    x2    x1
0    <=0   <=-6  <=-5  <=-1  <=0
x4   <inf  <=0   <2    <=5   <7
x3   <inf  <=-1  <=0   <=4   <=5
x2   <inf  <=-5  <-3   <=0   <2
x1   <inf  <=-6  <=-5  <=-1  <=0


% M7 : 5<= x3  ; 1<= x4-x3<2  ; x4-x2 = 5 ;
 x3-x1 = 5 ;

     0     x4    x3    x2    x1
0    <=0   <=-6  <=-5  <=-1  <=0
x4   <inf  <=0   <2    <=5   <7
x3   <inf  <=-1  <=0   <=4   <=5
x2   <inf  <=-5  <-3   <=0   <2
x1   <inf  <=-6  <=-5  <=-1  <=0


% Mprime6 : 1<= x4-x3<2  ; 3< x4-x2 ;
 2<= x3-x2 ; 3< x3-x1 ; 1<= x2-x1<=3  ;

     0     x4    x3    x2    x1
0    <=0   <-3   <-3   <=-1  <=0
x4   <inf  <=0   <2    <inf  <inf
x3   <inf  <=-1  <=0   <inf  <inf
x2   <inf  <-3   <=-2  <=0   <=3
x1   <inf  <-3   <-3   <=-1  <=0


% Mprime7 : 1<= x4-x3<2  ; 3< x4-x2 ; 2<= x3-x2 ;
 3< x3-x1 ; 1<= x2-x1<2  ;

     0     x4    x3    x2    x1
0    <=0   <-3   <-3   <=-1  <=0
x4   <inf  <=0   <2    <inf  <inf
x3   <inf  <=-1  <=0   <inf  <inf
x2   <inf  <-3   <=-2  <=0   <2
x1   <inf  <-3   <-3   <=-1  <=0
```

Instruction des calculs associés à la figure 20 page 48

```
#Fichier apparition

K:=3;

M0 := Zone((x1=0 ^ x2=0 ^ x3=0 ^ x4=0));

M1:=Post(M0,(x3<=3),{x1,x3});

M2:=Post(M1,(x2=3),{x2});

M3:=Post(M2,(x1=2),{x1});

M4:=Post(M3,(x2=2),{x2});

M5:=Post(M4,(x1=3),{x1});

M6:=Post(M5,(x4<x3+2),{});

M7:=Post(M6,(x4>x3+1),{});

Mprime1:=
Extra(K,Post(M0,(x3<=3),{x1,x3}));

Mprime2:=
Extra(K,Post(Mprime1,(x2=3),{x2}));

Mprime3:=
Extra(K,Post(Mprime2,(x1=2),{x1}));

Mprime4:=
Extra(K,Post(Mprime3,(x2=2),{x2}));

Mprime5:=
Extra(K,Post(Mprime4,(x1=3),{x1}));

Mprime6:=
Extra(K,Post(Mprime5,(x4<x3+2),{}));

Mprime7:=
Extra(K,Post(Mprime6,(x4>x3+1),{}));
```

Sortie associée

```
Hybrid and Compositional  Model Checking
http://www.lsv.ens-cachan.fr/~fl/
version 1.0 b - (DBM)
% load "apparition";
K : (int) 3

% M6 : 5<= x3  ; 1<= x4-x3<2  ; x4-x2 = 5;
 x3-x1 = 5 ;

     0     x4    x3    x2    x1
0    <=0   <=-6  <=-5  <=-1  <=0
x4   <inf  <=0   <2    <=5   <7
x3   <inf  <=-1  <=0   <=4   <=5
x2   <inf  <=-5  <-3   <=0   <2
x1   <inf  <=-6  <=-5  <=-1  <=0


% M7 : 5<= x3  ; 1< x4-x3<2  ; x4-x2 = 5 ;
 x3-x1 = 5 ;

     0     x4    x3    x2    x1
0    <=0   <-6   <=-5  <-1   <=0
x4   <inf  <=0   <2    <=5   <7
x3   <inf  <-1   <=0   <4    <=5
x2   <inf  <=-5  <-3   <=0   <2
x1   <inf  <-6   <=-5  <-1   <=0


% Mprime6 : 1<= x4-x3<2  ; 3< x4-x2 ;
 2<= x3-x2 ; 3< x3-x1 ; 1<= x2-x1<=3  ;

     0     x4    x3    x2    x1
0    <=0   <-3   <-3   <=-1  <=0
x4   <inf  <=0   <2    <inf  <inf
x3   <inf  <=-1  <=0   <inf  <inf
x2   <inf  <-3   <=-2  <=0   <=3
x1   <inf  <-3   <-3   <=-1  <=0


% Mprime7 : 1< x4-x3<2  ; 2<= x3-x2 ;
 3< x3-x1 ; 1<= x2-x1<=3  ;

     0     x4    x3    x2    x1
0    <=0   <-3   <-3   <=-1  <=0
x4   <inf  <=0   <2    <inf  <inf
x3   <inf  <-1   <=0   <inf  <inf
x2   <inf  <-3   <=-2  <=0   <=3
x1   <inf  <-3   <-3   <=-1  <=0
```