ÉCOLE NORMALE SUPÉRIEURE DE CACHAN

THÈSE

présentée
pour obtenir

LE GRADE DE DOCTEUR EN SCIENCES
DE L'ÉCOLE NORMALE SUPÉRIEURE DE CACHAN
Spécialité INFORMATIQUE

PAR

Emmanuel FLEURY

Automates temporisés avec mises à jour

Soutenue le 17 Décembre 2002 devant le jury composé de :

MM.	Marie-Claude GAUDEL	Présidente
	Paul GASTIN	Rapporteur
	Sergio YOVINE	Rapporteur
	Kim LARSEN	Examinateur
	François LAROUSSINIE	Examinateur
	Antoine PETIT	Directeur de thèse

Laboratoire **S**pécification et **V**érification – CNRS UMR 8643 ;
ENS de Cachan ; 61, Avenue du Président Wilson ; 94235 CACHAN Cedex

Table des matières

Table des figures

Introduction

There's always one more bug, even after that one is removed.
— Weinberg

La vérification prend une place croissante dans l'informatique moderne. En grande partie parce que nous nous reposons de plus en plus sur les systèmes informatiques. Que cela soit pour communiquer, pour nous déplacer, pour nous distraire ou encore pour nous soigner, nous utilisons des ordinateurs et les logiciels qui les accompagnent. De par cette généralisation de l'outil informatique, certains systèmes acquièrent plus d'importance que d'autres car ils ont la responsabilité de gérer des aspects sensibles de notre environnement. On appelle ces systèmes des *systèmes critiques*.

Pour la définir rapidement, la vérification est la part de l'informatique qui s'attache à prouver que ces 'systèmes critiques' possèdent bien les propriétés voulues par leur concepteur. Ce qui permet de réduire le nombre des défaillances possibles et d'augmenter la confiance que l'on peut accorder à ces systèmes.

Dans cette introduction[1] je vais, évidemment, évoquer les exemples catastrophiques habituels que nous utilisons tous lorsque quelqu'un nous demande ce que l'on fait dans la vie. Mais, à force de toujours parler de fusées qui s'écrasent, de banqueroutes éclairs, d'avions en détresse, j'ai fini par me demander si ces histoires, que nous citons de mémoire pour avoir entendu quelqu'un d'autre en parler, étaient réelles ou pas. Je me suis donc mis en quête de faits et non plus d'histoires de seconde main…[2]

J'avoue que ce que j'ai trouvé m'a à la fois rassuré et effrayé. Rassuré car j'ai à présent la conviction que la vérification est cruciale. Et effrayé car le nombre des exemples que j'ai découverts ne s'arrête pas à ceux que je vais citer, loin de là.

Après les cas que j'ai décidé de traiter, je parlerai brièvement dans cette introduction des différentes méthodes de vérifications et des particularités de chacune. Puis je me concentrerai sur les aspects qui sont traités dans cette thèse en dégageant ses principaux apports au domaine. Enfin, je terminerai par un plan informel du reste du document.

1 Pourquoi doit-on faire de la vérification ?

Dans un monde parfait, avec des programmeurs parfaits, du matériel parfait et un environnement favorable, la vérification n'aurait pas lieu d'être. Mais, le fait est que nous sommes loin

[1] Inspirée en partie du cours de Franck Van Breugel, '*Introduction to Program Verification*' donné à l'Université de York à Toronto au Canada (voir : `http ://www.cs.yorku.ca/franck/teaching/2000-01/3341/`).

[2] La majeure partie des informations que je cite ici sont tirées des archives du forum `comp.risks` que l'on peut consulter sur `http ://catless.ncl.ac.uk/Risks/`

de cette perfection ambiante. Les programmeurs font des erreurs, le matériel tombe en panne, et l'environnement est bien souvent hostile. Plutôt que d'essayer de rendre le monde parfait autour du système informatique, il semble plus raisonnable de rendre le système informatique aussi parfait que possible en éliminant les erreurs de conception.

La vérification sert justement à s'assurer que les logiciels vérifient certaines propriétés jugées cruciales par leur concepteur. Cependant, le coût de la vérification d'un système informatique est souvent très lourd. À la fois à cause des compétences engagées et du temps nécessaire à cette vérification. La question qui se pose alors vraiment est de savoir si les défaillances qui risquent de survenir peuvent avoir des répercussions qui justifient les investissements consentis.

Pourquoi ne pas simplement faire confiance aux programmeurs, avoir du matériel de rechange sous la main et supposer que l'environnement ne sera pas trop hostile en moyenne ?

Ces règles suffisent à la plupart des logiciels, pourquoi n'en serait-il pas de même pour les logiciels critiques ? Hélas, comme vont nous le démontrer les exemples qui suivent, cet optimisme béat est souvent générateur de catastrophes.

Nous allons commencer par deux problèmes typiques de bogues amplifiés par un réseau. Les réseaux sont parfois très sensibles aux problèmes lorsque les dits problèmes font entrer en résonnance leurs différents composants du réseau entre eux. Évidemment, les proportions que prennent alors les incidents dépassent de loin ce à quoi on aurait pu s'attendre.

1.1 Le crash d'AT&T

Tout bon informaticien qui se respecte connait le mot clef `switch` en langage C [KR88]. Il permet de faire un choix sur la valeur d'une variable. La syntaxe en est la suivante :

```
01   switch(i)
02   {
03    case 1:
04      function1();
05      break;
06
07    case 2:
08      function2();
09      break;
10
11    default:
12      defaultfunction();
13   }
```

Que se passe-t-il si l'on oublie le deuxième `break` ? (ligne 9)

Les fonctions `fonction2()` et `defaultfunction()` sont exécutée l'une à la suite de l'autre.

A priori, rien d'extraordinaire dans cette 'petite' erreur de programmation. Mais placée au mauvais endroit et au mauvais moment, cette 'petite' erreur peut avoir des conséquences désastreuses, les ingénieurs d'AT&T peuvent en témoigner.

AT&T gère la majeure partie des communications téléphoniques de la côte Est des États-Unis. Entre le milieu des années 80 et le début des années 90, elle possèdait un réseau d'environ

1400 commutateurs CCS7 qui avaient pour fonction de relayer tous les appels longue distance, internationaux et les numéros verts en '800'. Ces commutateurs tournaient sous un système d'exploitation d'AT&T appelé le '*System 6*'. Vers la fin de l'année 1989, 80 de ces commutateurs furent mis à jour avec le '*System 7*' qui était plus rapide et qui possédait plus de fonctionnalités. Bien sûr, avant de passer à un test réel, le '*System 7*' avait été intensivement testé et validé par AT&T. Le test en réel se passa sans encombre, tant et si bien que les mises à jour des commutateurs CCS7 du '*System 6*' vers le '*System 7*' commencèrent. Puis, courant décembre 1989, les programmeurs d'AT&T conçurent un patch pour le '*System 7*' qui devait réduire le temps de réponse des commutateurs et rendre la récupération des pannes plus sûre.

Ce patch n'avait pas été testé !

Tout se passa normalement jusqu'au 15 Janvier 1990. À 14h25, le commutateur de Manhattan (New York) dut redémarrer à cause d'un problème matériel. Il commença donc, comme prévu par le programme, par informer les commutateurs du réseau qu'il allait être indisponible pendant un court moment (le temps de redémarrer). Les autres commutateurs en prirent note et passèrent dans un mode où ils devaient mettre en attente toutes les communications qui étaient destinées au commutateur de Manhattan jusqu'à ce qu'il ait fini de redémarrer. L'ensemble du processus ne devait pas prendre plus de 4 à 6 secondes.

Après ce court laps de temps, le commutateur de Manhattan revint sur le réseau et signala aux autres commutateurs qu'ils pouvaient lui envoyer les appels mis en attente. À la réception de ce signal, les autres commutateurs entrèrent dans le mode de récupération des communications mises en attente. Et c'est là qu'était le problème.

Pendant une période d'environ un centième de seconde, les commutateurs qui recevaient au moins deux demandes de connexion téléphonique (ce qui avait une faible probabilité d'arriver) réinitialisaient une variable qui aurait dû rester inchangée (à cause d'un `break` manquant). Cette perte d'information était alors interprétée par le système comme un problème majeur et causait le redémarrage du commutateur.

Suite au redémarrage du CCS7 de Manhattan, seuls trois autres commutateurs eurent ce problème. Bien évidemment, ils avertirent le réseau qu'ils devaient redémarrer. Et à nouveau tous les commutateurs passèrent par la période de un centième de seconde de vulnérabilité. En moins d'une heure, le nombre des redémarrages grimpa en flèche, des dizaines de commutateurs redémarraient à chaque seconde et le réseau était totalement inutilisable.

Comme il s'agissait d'un jour de congés aux États-Unis (Martin Luther King Junior Day), les ingénieurs d'AT&T furent rappelés en urgence pour venir prêter main forte à leurs collègues qui étaient de garde et qui étaient complétement dépassés par les évènements. Fort heureusement, il restait 34 commutateurs CCS7 qui étaient encore sous '*System 6*' et qui, donc, n'avaient pas reçu le patch fatal. Un des ingénieurs d'AT&T fini par remarquer que ces machines étaient les seules à n'avoir jamais redémarré depuis le début de l'incident. Elles avaient du mal à gérer les demandes de mise en attente qui venaient de toutes parts, mais elles tenaient le coup. La décision fut prise de repasser tous les commutateurs sous '*System 6*' en attendant de comprendre plus en détail ce qui s'était passé. Ce n'est que vers 23h30 que l'activité du réseau revint à la normale. Soit **9 heures** plus tard. Le rapport officiel d'AT&T concernant cet incident a été joint en annexe (voir l'annexe A, section A.1 page 125).

On estime à 6 000 le nombre de communications interrompues et à 70 000 000 le nombre de coups de fil qui n'ont pu être passés. La réputation d'AT&T fut largement entachée par cet incident et plusieurs compagnies changèrent de prestataire après cet épisode.

L'analyse du problème prit plusieurs semaines. Et finalement, le bogue fut découvert après l'analyse du code C du patch.

1.2 Le ver de Morris

Le ver de Morris fait partie des légendes d'Internet. Il s'agit d'une des premières et des plus sérieuse attaques qu'ait jamais eue à affronter Internet. Pas moins de 6 000 machines furent infectées et durent être redémarrées *physiquement*[3] à plusieurs reprises.

De nos jours 6 000 machines ne représentent plus rien. N'importe quelle université ou multi-nationale possède plusieurs milliers d'ordinateurs individuels. Mais en 1988, l'année de l'attaque, les 6 000 machines dont je parle sont des VAX et des Sun-3 et constituent la colonne vertébrale de l'Internet. Encore à l'heure actuelle, immobiliser plusieurs milliers de backbones[4] revient à paralyser totalement le réseau Internet Français ou une bonne part du réseau Européen ou Américain.

Quel groupe subversif a lancé cette attaque ?
Quel gouvernement est derrière cet acte de guerre informatique ?
Quels intérêts étaient en jeu ?

Rien de tout cela, en fait, il s'agissait d'un étudiant de 23 ans qui s'amusait. Il a utilisé des failles du système BSD, du logiciel `sendmail` et `finger` pour créer un ver[5] qu'il voulait faire vivre de façon cachée sur Internet. Le problème est qu'il avait mal estimé la vitesse à laquelle le ver se reproduirait. En effet, ce ver se reproduisait tellement vite qu'il monopolisait toutes les ressources des machines qu'il avait infectées, les utilisant pour émettre son code vers d'autres ordinateurs.

Toute l'histoire commence le soir du 2 Novembre 1988 vers 18h00[6], lorsque Robert T. Morris, étudiant à l'Université de Cornell à Ithaca (NY) lance son ver sur le réseau du MIT Artificial Intelligence Laboratory. Très vite, le virus se propage jusqu'à la côte ouest via la liaison directe qui relie le MIT à Berkeley. Vers 19h00 la passerelle principale de Berkeley est infectée et le virus s'attaque aux machines plus internes. Vers 20h30, les ingénieurs système de Berkeley remarquent une charge anormale sur plusieurs serveurs du réseau, mais il est déjà trop tard. Au MIT, le virus a déjà conquis l'ensmble des machines, profitant de la nuit et de l'absence des ingénieurs système pour gangrener tout le réseau.

Pendant toute la nuit du Jeudi 2 Novembre 1988 au Vendredi 3 Novembre 1988, des centaines d'administrateurs systèmes vont se battre contre ce virus qui, lorsqu'il semble être éradiqué d'une machine, resurgit d'une autre machine du réseau pour re-infecter la machine guérie quelques instants plus tôt. L'un des premiers avis signalant ce virus à un large auditoire est retranscrit sur la figure 1 page suivante.

Pendant ce temps, Robert T. Morris commence à se rendre compte de l'ampleur du désastre et appelle un de ses amis à Harvard (Andy Sudduth). Par téléphone, il lui explique le fonctionnement du virus. Trop effrayé pour se manifester, il laissera Andy Sudduth poster un e-mail anonyme donnant les détails techniques du virus et comment l'arrêter. Malheureusement, cet

[3] C'est à dire que la machine ne pouvait plus être accédée de façon distante et que la présence physique de l'ingénieur système était requise

[4] Backbone : Ordinateur clef chargé de collecter et de rediriger les informations sur l'Internet.

[5] *Ver* : Virus informatique qui se propage via le réseau.

[6] Heure de New York.

```
Newsgroups: comp.risks
Subject: Virus on the Arpanet - Milnet
<Stoll@DOCKMASTER.ARPA> Thu, 3 Nov 88 06:46 EST
Re Arpanet "Sendmail" Virus attack November 3, 1988

Hi Gang!

It's now 3:45 AM on Wednesday 3 November 1988.
I'm tired, so don't believe everything that follows...

Apparently, there is a massive attack on Unix systems going on
right now.

I have spoken to systems managers at several computers, on both the
east & west coast, and I suspect this may be a system wide problem.

Symptom:  hundreds or thousands of jobs start running on a Unix
system bringing response to zero.
[...]

This virus is spreading very quickly over the Milnet.  Within the past 4
hours, I have evidence that it has hit >10 sites across the country,
both Arpanet and Milnet sites.  I suspect that well over 50 sites have
been hit.  Most of these are "major" sites and gateways.
[...]

This is bad news.

  Cliff Stoll dockmaster.arpa
```

FIG. 1 – *Une des premières annonces signalant le virus sur Internet.*

e-mail, posté à 00h34 le vendredi 3 Novembre 1988, sera bloqué jusqu'au 5 Novembre sur un backbone à cause du ver qui s'est étendu déjà sur une bonne partie de l'Internet américain.

Par une coïncidence fortuite, le congrès annuel de Berkeley sur Unix a lieu cette semaine là. Une quarantaine de personnalités importantes de l'Internet (ingénieurs système, programmeurs, etc) se trouvent loin de leur réseau. La plupart passent leur journée du vendredi au téléphone avec des administrateurs systèmes affolés, prennent le premier avion pour rentrer chez eux en catastrophe ou se joignent à l'équipe de Berkeley pour combattre le ver. Peu à peu la résistance s'organise et le code du ver est capturé par un certain nombre d'ingénieurs système qui se mettent à l'analyser (voir annexe A, section A.2 page 127).

Pendant encore plusieurs jours le ver va bloquer le réseau Internet et rendre la communication difficile via ce biais. Le samedi 4 Novembre 1988 a lieu une première présentation rapide du ver à un Workshop du congrès Unix de Berkeley et ce n'est que la semaine suivante, le samedi 11 Novembre 1988, que le code du ver est totalement analysé. La presse ne reste pas indifférente à l'affaire (voir annexe A, section A.3 page 129). Et finalement, Robert T. Morris est arrêté le dimanche 6 Novembre 1988 (voir annexe A, section A.4 page 131).

Comment cela a-t-il pu arriver ? Comment aurait-on pu l'éviter ?

Le virus utilisait seulement trois failles dont le célèbre problème de *dépassement de tampon* ('*buffer overflow*' ou '*stack smashing*') qui permet de réécrire l'exécutable d'un programme en passant à ce dernier un argument dont la taille dépasse la capacité prévue. Si le programme ne vérifie pas la taille de l'argument, il est alors possible de le modifier. Et si l'on est assez malin, on peut alors réécrire un programme qui nous donne les pleins pouvoirs sur ce processus.

Ce problème est actuellement toujours utilisé par les pirates. Le nombre des programmes et virus utilisant cette faille[7] est en forte augmentation ces derniers temps. Pourtant, les programmeurs doivent toujours vérifier à la main que le logiciel ne contient pas ce bogue. Alors que ce genre de problème peut typiquement être traité par des méthodes de vérification comme l'analyse statique (voir section 2.1).

Les banques ne sont pas exemptes de problèmes informatiques. On cite souvent les salles de marché comme exemple de lieu où se trouvent des logiciels critiques, mais quelles seraient réellement les pertes s'il y avait un bogue dans ces logiciels ? L'exemple de la Banque de New York est assez parlant dans ce domaine.

1.3 Le désastre de la '*Bank of New York*'

Que se passe-t-il lorsqu'on dépasse la capacité d'une variable ? Plus précisemment, que se passe-t-il si on code un entier sur 16 bits et que l'on atteint la valeur fatidique de 32.767 ? Il y a fort à parier qu'à l'incrémentation suivante notre variable contiendra la valeur 0. Ce problème peut s'avèrer génant, surtout lorsque de cette variable dépend beaucoup d'argent.

Le matin du Jeudi 21 Novembre 1985, une nouvelle version du logiciel de traitement des bons du Trésor fut chargé sur l'ordinateur de la Banque de New York qui gérait les transactions en temps-réel avec Wall Street. Vers 10h00, il apparut aux '*traders*' de la Banque de New York qu'une subite crise des bons du Trésor avait heurté le marché. La panique commença à se répandre dans les bureaux de la Banque de New York, les '*traders*' essayant

[7]On peut citer par exemple, le ver Lion (CERT IN-2001-03), ou encore le ver 'Code Rouge' (CERT CA-2001-19), ... Consulter le site du CERT pour plus d'informations (`http ://www.cert.org/`).

de sauver ce qu'ils pouvaient en investissant dans des métaux précieux (or, argent et platine). En cherchant à comprendre ce qui se passait certains pensèrent à consulter leurs sources habituelles. Étonnament, la crise des bons du Trésor ne semblait toucher que la Banque de New York et aucune autre banque ou place financière. Afin de laisser aux responsables de la Banque de New York le temps d'éclaircir ce mystère, le marché, qui aurait dû fermer à 14h30 ce jeudi, resta ouvert jusqu'à 1h30 du matin le vendredi suivant.

Le problème venait en fait du compteur qui dénombrait les demandes d'achat ou de vente sur les bons du Trésor. Habituellement codé sur 32 bits, celui du nouveau programme avait été codé sur 16 bits. Évidemment, lorsque la 32.768ème demande d'achat ou de vente est arrivée dans la file, le compteur a indiqué zéro et 32.768 demandes sont restées ignorées du système. Ce qui a provoqué la '*crise*' des bons du Trésor. Un article du Wall Street Journal relata l'incident, en citant les conséquences financières graves qu'eut ce dysfonctionnement (voir annexe A, section A.5 page 132). La Banque de New York dut tout de même emprunter 20 milliards de dollars au gouvernement pour payer ses dettes. Les intérets de cette somme, remboursée par la suite, sont de l'ordre de plusieurs millions de dollars.

Quittons un moment la terre et regardons vers les étoiles. Les sondes, les satellites et les lanceurs spatiaux sont une source inépuisable de systèmes embarqués avec des contraintes fortes (temps-réel, tolérance aux pannes, latence importante dans les communications, ...). De plus, chacune de ces missions représente un investissement colossal que seuls quelques gouvernements peuvent se permettre. Pourtant, la vérification semble encore avoir du mal à percer dans cette branche. Pour preuve le nombre des problèmes logiciels que l'on peut y dénombrer (Viking Lander en 1986, Mariner 1 en 1987, Phobos 1 en 1988, Phobos 2 en 1989, ...). Nous allons maintenant citer trois cas qui concernent ce genre d'appareils.

1.4 Phobos 1

Lorsqu'il s'agit d'espace, à peu près toutes les quantités que l'on considère sont multipliées par un facteur '*astronomique*'. Les distances, les vitesses, les coûts, ... Et cela est d'autant plus sensible qu'il faut parfois très peu de chose pour faire échouer une mission.

Le Samedi 10 Septembre 1988, le centre de contrôle russe de la mission *Phobos 1* en direction de Mars déménageait. Il quittait la Crimée pour rejoindre la banlieue de Moscou. Un programme de 20 à 30 pages a été envoyé à *Phobos 1* à cette occasion pour faire quelques réajustements concernant ce changement. Après avoir reçu le programme, la sonde, au lieu de prendre note des changements, réorienta ses panneaux solaires à l'opposé du Soleil. Dû à la latence des communications, le temps que le centre de contrôle reçoive le message d'acquittement de la sonde sur le mouvement des panneaux solaires et que la réponse du centre de contrôle demandant la correction immédiate du positionnement des panneaux solaires parvienne à la sonde, il était trop tard. La sonde avait épuisé ses batteries et ne répondait plus. L'ensemble de cette mission avait coûté l'équivalent d'un milliard de dollars.

Après analyse, il s'est avéré que l'opérateur qui avait tapé les 20 à 30 pages de programme avait omis UN caractère dans ce programme (voir annexe A, section A.6 page 133 et section A.7 page 133). Faire confiance aux seuls opérateurs humains, susceptibles de faire des erreurs, lorsqu'il s'agit de tels investissements est un pari dangereux. Pourtant, c'est ce que semble avoir choisi l'agence spatiale Russe. Mais les américains font aussi des choix surprenants...

1.5 Mars Pathfinder

Le temps-réel apporte un ensemble de problèmes assez intéressant à traiter. Il existe d'ailleurs une littérature importante sur ce sujet. Et particulièrement dans le domaine du *model-checking* (voir section 2.3). Pourtant les ingénieurs du *Jet Propulsion Laboratory* (JPL) semblent avoir décidé de s'en passer.

L'un des objectifs de la mission *Mars Pathfinder* était de faire atterrir une sonde mobile (*Sojourner*) à la surface de Mars et de lui faire faire, entre autre, des relevés topographiques et météorologiques. Jusqu'au moment de l'atterrissage sur Mars, le 4 Juillet 1997, la mission s'était déroulée sans incident particulier. L'atterrissage, lui-même, qui inaugurait une nouvelle technique tirant parti d'une gravité plus faible sur Mars que sur Terre, se passa à merveille. Après quelques jours à explorer le terrain alentour et à déplacer la sonde, les ingénieurs du JPL commencèrent à collecter les données météorologiques. À partir de ce moment, les choses commencèrent à se dégrader sérieusement.

Sojourner commença à réinitialiser son système informatique de façon aléatoire, mais suffisamment fréquemment pour empêcher la collecte d'informations sur son environnement. Après avoir vérifié qu'il ne s'agissait pas d'un problème matériel, les ingénieurs du JPL durent se résoudre à l'idée qu'il s'agissait d'un bogue dans leur système informatique. Le système informatique de Sojourner était basé sur un noyau temps-réel nommé *VxWorks* développé par *Wind River Systems*. Il permettait la planification préemptive de processus avec des priorités. Il faut savoir de plus que Sojourner contenait un '*bus de données*' qui peut être vu comme un espace mémoire partagé par tous les composants du système. La tâche de gestion du bus était souvent en priorité élevée afin de collecter les données qui se trouvaient sur le bus ou en mettre de nouvelles. L'accès au bus de données se faisait via un système d'exclusion mutuelle semblable à un sémaphore (mutex). Le processus voulant lire ou écrire sur le bus, prenait le mutex et ne le relachait que lorsqu'il avait fini, empêchant ainsi les autres processus d'accéder au bus de données pendant qu'il l'occupait.

Le problème venait en fait de la tâche chargée de collecter les données météorologiques. Cette tâche lançait un processus de basse priorité de façon sporadique et utilisait le bus de donnée pour faire passer ses informations au système central. Ce processus, lors de sa planification, engendrait la création d'un autre processus de haute priorité qui devait s'effectuer après que les données aient été écrites sur le bus et qui devait lire les données et les transférer dans la mémoire centrale.

Malheureusement, de temps à autre, un processus entrait en conflit avec ces processus. La tâche qui s'occupait d'envoyer les données collectées et traitées à la Terre et nécessitait aussi le bus de données pour envoyer les messages à l'émetteur. Cette tâche engendrait des processus de moyenne priorité. Et il arrivait que cette tâche demande une planification juste après que la tâche météorologique ait été planifiée. Comme la tâche de communication engendrait des processus de moyennes priorités, son processus passait devant celui de l'écriture des données météorologiques sur le bus. Mais elle restait de priorité inférieure à la tâche de lecture sur le bus de données.

On en arrivait donc à la situation suivante :

- La tâche de communication attendait que la tâche de lecture ait fini pour lancer son processus,
- La tâche de lecture, quant à elle, attendait que la tâche d'écriture ait fini pour pouvoir lire les données sur le bus,

– Enfin, la tâche d'écriture, attendait que la tâche de communication ait fini.

Deadlock !

Après un certain temps, un processus chargé de veiller sur la vivacité du système redémarrait le tout en réinitialisant du même coup toutes les mémoires. Ce qui avait pour résultat d'envoyer dans les limbes toutes les informations que la sonde avait stockées et se préparait à envoyer.

Comme le relate un message publié dans `comp.risks` (voir le résumé de l'annexe A, section A.8 page 134), les ingénieurs du JPL ont cherché le bogue dans leur programme en recréant les conditions dans lesquelles la sonde produisait l'erreur sur une copie du système qui était dans leur laboratoire. Après un grand nombre d'essais, le problème fut reproduit et isolé.

La vérification, et particulièrement le *model-checking* (voir section 2.3) aurait permis de trouver ce '*deadlock*' de façon automatique et sans avoir à lancer plusieurs centaines de fois le logiciel en espérant que le bogue se manifestera cette fois-ci. Après les Russes et les Américains, les Français eux-aussi ne sont pas en reste, alors que pourtant ils sont habitués à valider leurs logiciels.

1.6 Le crash du vol 501

Faire appel à des méthodes de vérification n'est pas toujours suffisant pour assurer la validité des logiciels. Il faut aussi prendre en considération la méthodologie que l'on utilise et s'assurer qu'elle n'interfère pas avec la correction du logiciel. L'exemple du vol 501 d'*Ariane V* montre que vérifier indépendamment des parties du logiciel n'équivaut pas à vérifier l'ensemble du logiciel. Particulièrement si l'on change le contexte dans lequel cette partie du logiciel s'exécute.

Le 4 Juin 1996, le premier vol du lanceur *Ariane V* se solda par un échec. Environ 40 secondes après son allumage, la fusée se brisait en deux provoquant ainsi son auto-destruction immédiate (voir le message de John Rushby sur `comp.risks`, figure 2). Le 13 Juin 1996, un groupe indépendant de scientifiques avec à sa tête J. L. Lions, fut nommé pour découvrir l'origine de cette défaillance. Le compte-rendu de ses investigations fut rendu public le 19 Juillet 1996[8]

Aussi étonnant que cela paraisse, car les ingénieurs qui ont conçu *Ariane V* utilisaient des techniques de vérifications, c'est le logiciel qui guidait le lanceur qui fut mis en cause. Mais avant de détailler ce qu'il s'est passé, je vous propose de découvrir un peu le système référence inertielle d'*Ariane V*.

Un *système de référence inertielle* (SRI) est un appareil qui permet de savoir très exactement où l'on se trouve, où l'on va et à quelle vitesse. On retrouve ce genre d'instrument dans quasiment tous les missiles balistiques et les lanceurs spatiaux. C'est un appareil à la fois crucial et fragile. *Ariane V* en possédait deux mis en parallèle qui fonctionnaient de concert. Si un problème apparaissait sur le premier, le second prenait immédiatement le relais.

Voici maintenant comment les choses semblent s'être passées.

En tout premier lieu l'erreur logicielle provient d'un module qui a été conçu et vérifié pour *Ariane IV*, mais qu'on avait omis de vérifier pour *Ariane V*. Ce module contenait une fonction

[8]Voir : `http ://www.cnes.fr/espace_pro/communiques/cp96/rapport_501/rapport_501.html`

```
Newsgroups: comp.risks
Subject: Ariane 5 failure
John Rushby <RUSHBY@csl.sri.com>
Wed 5 Jun 96 14:54:47-PDT

>From cnn's web page www.cnn.com:

Faulty computer blamed in Ariane rocket failure

Experts studying the moments before the Ariane-5 rocket explosion say
faulty computer software may be to blame for the rocket veering off
course. Apparently, the rocket was misfed information that made it think
it was not following the right path. The rocket then changed direction,
causing the upper part to began to break apart.
```

FIG. 2 – *Première annonce suspectant un problème logiciel sur Ariane-5.*

utile uniquement lorsque la fusée était sur sa plate-forme de lancement et qui aurait due être désactivée ensuite. Or, cette fonction, non seulement était active pendant le vol, mais c'est elle qui causa la perte d'*Ariane V*. Elle contenait une conversion d'un flottant codé sur 64 bits vers un entier signé codé sur 16 bits. Ce flottant était lié à la poussée des réacteurs du lanceur. Sur *Ariane IV*, la poussée des réacteurs n'avait jamais permis de dépasser les 16 bits de l'entier, mais sur *Ariane V*, la poussée du lanceur avait été augmentée, suffisamment pour ne plus pouvoir être codée sur un entier signé de 16 bits.

Lorsque la poussée des réacteurs atteint la limite des 16 bits, environs 37 secondes après l'allumage, la fonction retourna une erreur d'opérande car la conversion n'était pas protégée contre les erreurs d'opérandes, contrairement à d'autres conversions qui se trouvaient dans la même fonction. Cette erreur d'opérande fut interprétée par le système comme une défaillance du premier système de référence inertielle (SRI1) et le système commuta sur le second (SRI2). Évidemment, SRI2 avait eu le même problème et transmettait un diagnostic d'erreur sur l'erreur d'opérande qu'il venait de subir. Cependant, le système interpréta ce diagnostic d'erreur comme des données de vol et tenta de corriger sa trajectoire en fonction de ce que lui fournissait SRI2. Vue du sol, la fusée apparue comme '*folle*', corrigeant sans cesse sa trajectoire dans des directions complétement aléatoires. 39 secondes après l'allumage, l'angle d'attaque d'*Ariane V* atteignit un point critique qui fit que les boosters se détachèrent en provoquant ainsi l'autodestruction du lanceur en plein vol.

Les conclusions du rapport sur la défaillance d'*Ariane V* portent en grande partie sur la méthodologie utilisée pour la qualification et la validation des systèmes informatiques utilisés qui sont jugées insuffisantes.

Quittons l'espace pour revenir sur Terre. Il arrive que les industriels s'intéressent à la vérification pour des produits dont le coût par pièce est dérisoire comparé aux engins dont nous venons de parler. Le fait est que produire en grande quantité un produit défaillant dans sa conception cause un tort indéniable aux firmes. Soit qu'elles aient à mettre en place une coûteuse réparation du produit défaillant, soit qu'elles perdent une part de leur prestige et de la confiance que lui accordent ses clients. On peut citer par exemple, le protocole infra-rouge de

Bang & Olufsen qui fut vérifié et corrigé par Uppaal [HSLL97]. Mais, certains bogues passent au travers des mailles du filet.

1.7 Le bogue du processeur Pentium

Tous les processeurs ont des bogues. La plupart du temps le grand public ignore tout de ces problèmes qui influent peu sur le fonctionnement normal des logiciels et ne concernent que le petit monde des programmeurs. Cependant, lorsqu'une simple division flottante renvoie un résultat incorrect il est difficile de cacher la chose.

Thomas Nicely est chercheur en théorie des nombres. En 1994, il testait un algorithme de calcul distribué pour énumérer des nombres premiers sur un parc de machines à base de processeurs 486. En Mars 1994, il ajouta une machine à base de *Pentium* à cette expérience. Le 13 Juin 1994, il collecta les résultats obtenus pour faire le point sur l'évolution de l'expérience. Il faut ajouter que les ordinateurs impliqués dans cette expérience calculaient aussi des résultats connus en parallèle afin de repérer les problèmes éventuels. Au cours de l'analyse des résultats, en ce 13 Juin, il apparut que certains calculs étaient en contradiction avec les résultats connus. Après avoir corrigé plusieurs erreurs dans son programme et inspecté toutes les options de son compilateur, Thomas Nicely, relança les calculs espérant avoir résolu le problème.

Le 4 Octobre, collectant à nouveau les résultats, il constata que le problème était toujours là, mais cette fois-ci, il réussit à isoler le problème sur l'ordinateur à base de processeur *Pentium* uniquement. Persuadé d'avoir affaire à un problème dans son compilateur il essaye d'isoler ce bogue... sans succès. Le 17 Octobre 1994, il eut la chance d'accéder à un autre ordinateur *Pentium*. Et l'erreur était aussi présente sur cette machine. Par la suite il l'expérimenta avec différents logiciels sans passer par son compilateur. Ce qui éliminait l'hypothèse du compilateur. Qui plus est, le problème n'apparaissait pas sur les 486. À la suite de cette série d'expérimentations, il eut la certitude que le problème venait du processeur *Pentium*.

Le 24 Octobre 1994, il informa *Intel* via le service technique. Et le 2 Novembre 1994 il reçut un appel d'un manager d'*Intel*, lui signalant qu'effectivement le bogue avait été trouvé par les ingénieurs d'*Intel* et qu'il allait recevoir deux processeurs 'corrigés' le jour même. Après avoir testé les nouveaux processeurs avec les tests mis au points sur les puces boguées, il s'avèra que ces processeurs produisaient effectivement un calcul correct pour la division flottante.

La découverte de ce bogue est intervenue très tard dans le cycle de vie du processeur *Pentium*. Il avait déjà été distribué à des millions d'exemplaires chez des entreprises ou des particuliers. Qui plus est, le test qui permet de réveler le problème était extrêmement aisé (voir l'équation 1).

$$
\begin{aligned}
4195835.0/3145727.0 &= 1.333\ 820\ 449\ 136\ 241\ 000 \quad \text{(valeur correcte)} \\
4195835.0/3145727.0 &= 1.333\ 739\ 068\ 902\ 037\ 589 \quad \text{(bogue du \textit{Pentium})}
\end{aligned} \tag{1}
$$

Le problème se situait dans la fonction de division des flottants FDIV du *Pentium* qui adoptait un algorithme totalement différent de celui du 486. Certes, il arrivait à obtenir deux bits du résultats à chaque cycle, alors que le 486 n'en calculait qu'un par cycle. Mais cet algorithme faisait des hypothèses fausses pour le cas général mais vraies pour une grande partie des cas. Ainsi, sur des divisions flottantes aléatoires, la probabilité de rencontrer une erreur était d'environ $1/8.77E9$. Mais cette faible probabilité est de peu d'utilité lorsque la division que vous voulez faire est justement celle qui provoque l'erreur. Le résultat retourné avait un écart important par rapport à la valeur attendue, même en tenant compte des erreurs d'arrondis.

Subissant les attaques d'un concurrent direct, *Intel* dut proposer l'échange standard des processeurs bogués par des processeurs corrigés. De plus, une psychose du *Pentium* bogué se propagea et cela ralentit fortement les ventes du processeur.

Il existe des bogues qui peuvent avoir des conséquences plus graves que la perte de communications, de ressources ou d'informations. Certains logiciels se trouvent en situation de décider de la vie ou de la mort d'hommes ou de femmes. Fort heureusement, les incidents dans ce genre de logiciels sont extrêmement rares, mais rare ne veut pas dire inexistant.

1.8 Le Therac-25

La série de problèmes provoquée par le *Therac-25* [LT93] est sans doute l'exemple le plus marquant que j'ai pu trouver[9]. Pas seulement parce que les constructeurs de cet appareil ont refusé de remettre en cause leurs méthodes tout au long des incidents, mais surtout parce que chaque erreur était ponctuée par un mort ou un blessé grave. Au total, on dénombre six cas connus pour lesquels le *Therac-25* fut clairement identifié comme ayant eu un dysfonctionnement grave pendant la thérapie d'un patient. Pour quatre d'entre eux, la mort était au bout du chemin.

Les accélérateurs linéaires médicaux accélèrent les électrons afin de créer un faisceau à haute énergie qui peut être utilisé pour détruire des tumeurs avec un impact minimal sur les tissus sains environnants. Les tissus superficiels sont traités par un faisceau d'"électrons. Pour les tissus plus internes, on utilise un faisceau de photons dans la longueur d'onde des rayons-X qui peuvent se focaliser à l'intérieur des tissus.

L'*Atomic Energy Commission Limited* (AECL) était une société publique appartenant au gouvernement Canadien. Elle construisait toute sorte d'appareils liés au nucléaire dont des accélérateurs linéaires médicaux comme le *Therac-6* (6 MeV, rayons-X), le *Therac-20* (20 MeV, électrons et rayons-X) et enfin le *Therac-25* (25 MeV, électrons et rayons-X). Cette course aux hautes énergies est due au fait que plus l'appareil est capable de produire des faisceaux de haute énergie plus il épargne les tissus superficiels lorsqu'il s'agit de traiter des tumeurs profondes.

Le *Therac-25* était vu comme l'aboutissement de cette série d'accélérateurs linéaires médicaux, à la fois compact et puissant, il possédait de plus une interface utilisateur bien plus conviviale que les précédentes versions. On peut aussi ajouter qu'un changement majeur avait eu lieu entre le *Therac-20* et le *Therac-25*. Alors que la sécurité du *Therac-20* reposait sur des éléments matériels, le *Therac-25*, lui, faisait confiance au logiciel.

En 1976, le premier prototype du *Therac-25* fut utilisé en milieu hospitalier et fin 1982, le premier appareil fut vendu. En tout, 11 *Therac-25* furent vendus et utilisés, cinq aux États-Unis et six au Canada. Les six incidents eurent lieu entre 1985 et 1987 (date à laquelle tous les appareils furent retirés du marché).

Premier Incident – Kennestone Regional Oncology Center, Juin 1985 Le 3 Juin 1985, une patiente devait subir un traitement par irradiation aux électrons au niveau de la clavicule. Lors de son traitement, elle ressentit une forte chaleur à l'épaule qui se transforma rapidement en sensation de brûlure. Et s'en plaignit à l'opérateur. Le technicien lui répliqua

[9] Voir : `http ://courses.cs.vt.edu/cs3604/lib/Therac_25/Therac_1.html`

que ce n'était pas possible mais dut reconnaître que l'épaule de la patiente était effectivement chaude au toucher.

Une fois rentrée chez elle, la patiente eut des rougeurs cutanées ainsi qu'une transpiration intense au niveau de la zone du traitement. Elle perdit peu à peu l'usage de son épaule puis de son bras et souffrait énormément. Il fut établi plus tard qu'elle avait dû recevoir une dose de radiation de l'ordre de 15 000 à 20 000 rad[10]. Il faut savoir que les doses usuelles pour ce type de traitement étaient de l'ordre de 200 rads.

Bien que l'incident fut signalé à l'AECL, ni le technicien qui vint inspecter la machine, ni l'opérateur ne furent capable de reproduire l'incident. Et l'affaire fut classée.

Deuxième Incident – Ontario Cancer Foundation, Juillet 1985 Le 26 Juillet 1985, un patient vint pour sa 24ème thérapie par irradiation dans la région du bassin. L'opérateur entra les paramètres de la prescription et activa la machine. Mais le *Therac-25* s'éteint après environ 5 secondes sur un message d'erreur. Le dosimètre de la machine indiquait '*no dose*' et le mode était '*treatment pause*'. Comme la dose de radiation semblait ne pas avoir été délivrée, l'opérateur pressa à nouveau la touche 'P' (*Proceed*) s'attendant à ce que la machine délivre la dose prescrite précédemment. Il s'agissait là d'une opération standard décrite dans le manuel. Les techniciens qui travaillaient sur le *Therac-25* étaient habitués à des erreurs aléatoires sur des procédures apparemment réalisées à l'identique et conformément au manuel. L'opérateur répéta donc quatre fois le processus et à la cinquième fois, la machine passa en mode '*treatment suspend*'. L'opérateur appela donc le service technique de l'hôpital qui ne trouva rien d'anormal dans le fonctionnement de cette machine.

Après son traitement, le patient se plaignit d'avoir ressenti des sensations de brûlures intenses semblables à d'importantes décharges électriques dans la région du traitement. Six autres patients furent traités par la suite ce jour là sans incident notable. Le patient revint le 29 Juillet 1985 pour la suite de son traitement et se plaignit de sensations de brûlures, de douleur au bassin et d'une transpiration anormalement élevée dans la région du traitement. La machine fut déclarée hors-service et le patient fut immédiatement hospitalisé pour surexposition à des radiations. Il décéda le 3 Novembre 1985 d'un cancer généralisé extrêmement virulent et dont le foyer principal semblait être la région du bassin. Il fut estimé qu'il avait été exposé à des doses de radiation de l'ordre de 13 000 à 17 000 rads.

Suite à une analyse de l'incident, il fut estimé qu'il s'agissait d'une défaillance matérielle passagère sur des micro-commutateurs. Passagère car, à nouveau, les ingénieurs de l'AECL n'avaient pas pu reproduire ce problème sur la machine. L'AECL ajouta des composants matériels redondants pour 'corriger' ce problème.

Troisième Incident – Yakima Valley Memorial Hospital, Décembre 1985 Suite à l'incident précédent, tous les *Therac-25* furent modifiés. Durant Décembre 1985, une femme vint au *Yakima Valley Memorial Hospital* pour subir un traitement avec le *Therac-25*. Après l'un des traitements, elle développa des rougeurs cutanées sur l'une des zones du traitement. En dépit de cela, elle continua tout de même le traitement jusqu'en Janvier 1986 car la réaction ne fut pas diagnostiquée comme 'anormale'. Entre fin Janvier et début Février, ce problème de rougeur fut examiné plus attentivement mais aucune cause précise ne fut trouvée pour l'expliquer. Ce n'est qu'à la suite d'un second incident en Février 1987 que les médecins revinrent rétrospectivement sur le cas de cette patiente et finirent par découvrir qu'elle avait

[10] rad = radiation absorbed dose

développé un ulcère chronique de la peau et une nécrose des tissus sous-cutanés. Heureusement pour elle, une opération chirurgicale permit de la guérir sans qu'aucune séquelle ne se manifeste par la suite.

Quatrième Incident – East Texas Cancer Center, Mars 1986 Le 21 Mars 1986, un patient vint pour subir son neuvième traitement, il consistait à détruire une tumeur qui se situait dans son dos. Il était donc allongé face contre la table pendant le traitement. L'opérateur était isolé du patient dans une salle de contrôle pendant que le traitement opérait. En temps normal, il avait un contact avec le patient via un micro et une caméra, tous deux placés dans la salle de traitement. Cependant, ce jour là, micro et caméra étaient hors d'usage et le technicien opérait donc à l'aveuglette.

Lorsqu'il eut fini de rentrer les données, juste avant d'envoyer le rayon, l'opérateur remarqua qu'il avait fait une erreur. Il avait tapé 'x' pour rayons-X au lieu de 'e' pour électrons. C'était une erreur courante car les traitements par rayons-X étaient plus fréquents. Il plaça le curseur sur le champ et modifia la valeur. Puis, comme les autres paramètres étaient encore valides, il les confirma rapidement en pressant la touche '*return*' plusieurs fois. La machine afficha alors qu'elle était prête à fonctionner et il appuya sur la touche qui envoyait le rayon. Après un moment, la machine s'arrêta en affichant un message d'erreur qui indiquait que la dose délivrée n'avait pas été conforme à la demande. D'après le dosimètre de la machine, cette dose avait été bien inférieure à celle que le patient devait recevoir. À nouveau, l'opérateur valida rapidement les champs qui n'avaient pas changé puis envoya le rayon. La machine réagit exactement de la même manière et indiqua une dose inférieure à celle qui devait être délivrée. À ce moment là, le patient frappa à la porte de la cabine de l'opérateur. Il était tremblant, en état de choc et furieux contre l'opérateur.

Lors de la première tentative, le patient avait ressenti une sorte de choc électrique à l'endroit du traitement. Comme il s'agissait de son neuvième traitement, il sut que cela n'était pas normal, il commença à se redresser sur la table et reçut un deuxième choc électrique dans le bras gauche au moment ou l'opérateur essayait d'appliquer le traitement pour la deuxième fois. Puis, il se dirigea vers la cabine de l'opérateur.

D'après une analyse postérieure, on estima à 16 500 à 25 000 rads la dose reçue par le patient en moins d'une seconde et concentrée sur une surface d'un centimètre carré. Dans les semaines qui suivirent, il souffrit de douleurs chroniques dans le cou et les épaules, perdit l'usage de son bras gauche, avec des nausées chroniques suivies de vomissements. Son état empira de jours en jours jusqu'à sa mort cinq mois après l'accident. L'autopsie confirma une intense irradiation dans la zone du traitement et au bras gauche.

Les techniciens de l'AECL qui vinrent inspecter la machine dans les jours suivants l'accident essayèrent de trouver l'origine des chocs électriques en supposant qu'un court-circuit de la machine avait électrifié une partie de l'appareil. Évidemment, ils ne trouvèrent rien en ce sens qui puisse expliquer l'accident. Une fois de plus, le logiciel ne fut pas mis en cause. Et le 7 Avril 1986, la machine était de retour à l'hôpital et les traitements reprirent.

Cinquième Incident – East Texas Cancer Center, Avril 1986 Le 11 Avril 1986, soit quatre jours après le retour de la machine, un autre patient reçut un traitement pour un cancer de la peau localisé sur le visage. Le même opérateur que celui de l'incident précédent était aux commandes. À nouveau, l'erreur de la correction du 'x' en 'e' fut faite. L'opérateur valida donc rapidement les champs inchangés comme il avait l'habitude de le faire. De nouveau, la machine émit le message d'erreur indiquant une dose inappropriée. L'opérateur entendit le patient appeler à l'aide et se précipita dans la salle de traitement. La patient avait ressenti une vive brûlure au visage et avait eu un flash. Il était en état de choc. Il décéda trois semaines après l'incident. Son autopsie révéla qu'une partie importante de son lobe temporal droit avait été brûlé par une haute dose de radiations (environs 25 000 rads).

Cette fois-ci, l'opérateur, qui avait déjà assisté à deux accidents, essaya de reproduire l'incident. Il répéta la manoeuvre telle qu'il s'en rappelait jusqu'à obtenir à coup sûr le message d'erreur. Si bien que lorsque le technicien d'AECL vint pour réviser la machine suite à l'incident, l'opérateur put répéter exactement la suite de manipulations qui engendrait le problème.

En fait, il s'agissait d'un problème logiciel. Lorsque les paramètres de traitement étaient entrés trop vite sur le terminal (en moins de 8 secondes), le comportement du *Therac-25* était totalement aléatoire (problème de *race condition*). Typiquement, lorsqu'une erreur avait été faite (changement du '*x*' en '*e*'), l'opérateur ne ressaisissait pas les valeurs et ne faisait que valider les entrées. De temps en temps, la saisie était trop rapide et le *Therac-25* recevait des valeurs aléatoires. Évidemment, comme les doses reçues étaient aléatoires, il est fort possible que certains patients aient expérimenté l'incident sans en avoir conscience simplement parce que les doses reçues n'avaient pas été aussi importantes que les cas qui ont pu être decelés. Cette erreur de logiciel fut retrouvée sur le *Therac-20*, mais elle n'engendrait pas de problème particulier car cet appareil était doté de protections matérielles contre ce genre d'incidents.

La solution trouvée par les ingénieurs d'AECL contre ce problème fut de réinitialiser tous les champs à chaque fois que l'opérateur revenait à ce formulaire. Comme aucun opérateur n'était capable de remplir tous les champs en moins de 8 secondes, ils considéraient que le problème était réglé. D'autres problèmes furent découverts dans le logiciel. Et ils furent résolus de manière tout aussi 'efficace'.

Sixième Incident – Yakima Valley Memorial Hospital, 1987 Un sixième et dernier incident eut lieu le 17 Janvier 1987. Comme dans les incidents précédents la machine sembla ne pas fonctionner et indiqua '*no dose*'. Et à nouveau, le patient se plaignit de sensation de brûlure pendant le traitement. Le patient décéda en Avril de la même année. Mais souffrant d'un cancer en phase terminale, il fut difficile de connaître clairement la part de responsabilité du *Therac-25*.

L'investigation de l'AECL découvrit de nouveaux problèmes dans le logiciels. Et finalement, la décision d'arrêter l'exploitation du *Therac-25* fut prise.

Pour finir vous pourrez trouver en annexe A, section A.9 page 137 un article paru dans le Boston Globe qui décrit les incidents. Étant donné que cet article date de 1986, il ne relate pas tous les incidents connus à ce jour.

Espèrons que cette série d'exemples a permis de mieux appréhender l'impact qu'a de nos jours l'informatique sur notre environnement. Il faut de plus ajouter que les choses évoluent plutôt en faveur d'une prise de contrôle toujours plus importante par des systèmes informatiques de notre environnement. Ce qui veut dire que le nombre de systèmes critiques va certainement augmenter considérablement dans les années à venir. Les méthodes permettant de parer à ces catastrophes existent. Certains industriels y ont recours de façon habituelle. De plus, d'importantes avancées sont faites dans ce domaine chaque année. Nous allons dresser ici un panel non exhaustif des principales méthodes de vérification.

2 Les différentes techniques de vérification

Avec l'accroissement du nombre des systèmes critiques et des problèmes parfois spectaculaires entraînés par les erreurs de ceux-ci, la vérification informatique est à l'heure actuelle un domaine en pleine expansion. À la fois au niveau de l'industrie qui se plie de plus en plus volontiers à ces contraintes, qu'au niveau de la recherche fondamentale.

Étonnamment, lorsqu'on cherche une définition de la vérification en informatique, il suffit d'ouvrir son dictionnaire :

> **Vérification** *n.f.* Action de vérifier, de s'assurer de l'exactitude de quelque chose en le confrontant avec ce qui peut servir de preuve.

Cette définition laisse cependant planer un doute sur ce que l'on vérifie exactement. En effet, contrairement à un mythe largement répandu, le but de la vérification n'est pas de produire des logiciels exempts d'erreur. Cet exploit étant la plupart du temps extrêmement difficile, pour ne pas dire impossible.

Le but de la vérification informatique se veut plus pragmatique. Il s'agit de prouver qu'un ensemble de propriétés, que les concepteurs tiennent pour cruciales, est vrai sur le logiciel.

Par exemple, on aurait pu demander que les programmes qui ont été attaqués par le ver de Morris (voir 1.2 page 12) ne possèdent pas cette faille particulière, ou que les dépassements de variables soient interdits (Banque de New-York, voir 1.3 page 14 et Ariane V, voir 1.6 page 17), ou encore que les paramètres entrés par un opérateur soient bien ceux pris en compte par le logiciel (*Therac-25*, voir 1.8 page 20), *etc.*

Hormis ces propriétés, il peut y avoir des bogues. À vrai dire, il y en aura certainement, mais le choix des propriétés doit garantir que les bogues résiduels ne seront ni dangereux pour le système, ni pour ses utilisateurs.

Reste encore à prouver ces propriétés ! Pour ce faire, il existe plusieurs méthodes. Je vais ici en détailler trois (analyse statique, preuves automatiques et vérification de modèle). Sans aucun souci d'exhaustivité. Il semble simplement que ces trois approches sont les plus couramment utilisées dans le domaine de la vérification.

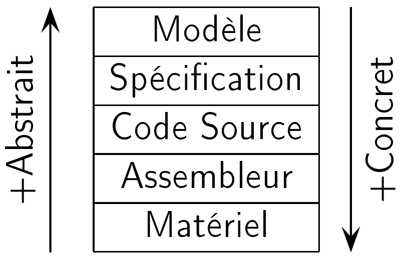

FIG. 3 – *Les différentes représentations abstraites/concrètes d'un logiciel.*

Ce qui différencie ces méthodes est essentiellement le niveau d'abstraction qu'elles utilisent pour représenter le logiciel à vérifier (voir figure 3). En effet, lorsqu'on parle d'un logiciel, parle-t-on de son code assembleur, de son code source, de ses spécifications ou de son algorithme ? Chacune des méthodes que nous allons présenter vérifie certaines propriétés particulières du logiciel. Ainsi, si l'on veut vérifier le code source ou le code assembleur, on utilisera une méthode d'analyse statique. Si, enfin, c'est l'algorithmique, ou encore les spécifications que l'on veut vérifier, on utilisera plutôt de la vérification de modèle ou de la preuve automatique.

2.1 Analyse statique (*Static Analysis*)

L'analyse statique correspond à l'ensemble des techniques qui permettent de déduire algorithmiquement des propriétés sur le comportement d'un logiciel à partir de l'analyse de son code source et/ou de son code assembleur.

Typiquement, on a recours à l'analyse statique lors de la compilation afin de détecter des bogues usuels ou afin d'optimiser le code assembleur obtenu, sans altérer le comportement du programme. Le compilateur va construire un arbre de syntaxe abstraite à partir du code source qui représente toutes les exécutions possibles. Cette structure étant facilement convertible en code assembleur par la suite. C'est sur cet arbre que le compilateur va vérifier les propriétés. Et c'est aussi cet arbre qu'il va modifier en vue d'optimiser le code assembleur. En prenant soin de vérifier qu'aucune de ces modifications n'altère le comportement du programme du point de vue de l'utilisateur (le programme d'origine et le programme optimisé doivent rester bisimilaires).

L'analyse statique permet de vérifier des propriétés telles que :

- La variable '`a`' est-elle réutilisée par la suite ?
- N'essaye-t-on jamais d'accéder à l'élément '`p+1`' d'un tableau qui n'en contient que '`p`' ?
- Les données contenues dans une variable peuvent-elles être corrompues par des accès sauvages à la mémoire ?
- ...

Cependant, cette technique a ses limites. Comme son nom l'indique, il s'agit d'analyse *statique* et non dynamique. C'est à dire que l'on ne peut réellement vérifier que les variables qui sont initialisées de façon statique dans le programme. Ce qui élimine d'emblée toutes les variables dont l'initialisation est dynamique.

Pourquoi ne serait-il pas possible d'appliquer la même méthode aux variables dynamiques ? Tout simplement parce que l'analyse statique revient à explorer toutes les exécutions possibles et que si certaines variables sont dynamiques, le nombre des exécutions possibles est infini.

Mais, peut-être existe-t-il un algorithme qui réduise le nombre de ces exécutions possibles à un nombre fini ? En fait, il a été prouvé que cela n'est pas possible. C'est un résultat connu sous le nom de théorème de Rice :

Théorème 1 (Rice). *Toute propriété extensionnelle*[11] *non triviale de programmes écrits dans un langage récursivement énumérable est indécidable.*

L'analyse statique permet donc de vérifier un grand nombre de propriétés mais reste limité à cause de son approche trop concrète du problème à des propriétés du code assembleur. Et rend inaccessible des propriétés de haut niveau qui impliquent le comportement de l'algorithme lui-même. À l'heure actuelle il existe des résultats qui combinent l'analyse statique et l'interprétation abstraite afin d'inclure une partie de ces variables dynamiques dans la vérification [CC76]. Mais, il ne s'agit là que d'approximations qui signalent des problèmes probables que le programmeur aura à vérifier par lui-même ensuite.

Pour conclure, seules des techniques intervenant à des niveaux d'abstraction plus élevés permettent de vérifier des propriétés plus générales sur le programme.

[11] *extensionnelle* signifie ici que la propriété doit ne dépendre que de la *sémantique* du programme, et pas de la façon dont il est codé (nombre de ligne, nombre de tests, ...)

2.2 Preuve assistée (*Theorem Proving*)

La vérification par preuve assistée [Rus00b, Rus00a] correspond à l'ensemble des techniques qui permettent de déduire, à travers l'utilisation d'un assistant de preuve (PVS [COR+95], COQ [HKC97]), des propriétés sur le comportement d'un logiciel, algorithme ou protocole à partir de l'analyse d'un modèle mathématique de celui-ci.

La méthode consiste à exprimer le programme et son environnement sous la forme d'un modèle mathématique. Ce modèle est obtenu soit par un traitement automatique du code source du programme, soit par une interprétation qu'en a faite un opérateur humain lorsque le programme est trop complexe pour être traité automatiquement. Quelle que soit la méthode utilisée pour concevoir ce modèle mathématique, il est important de prouver que les propriétés que l'on veut démontrer se comportent de façon identique sur le programme et sur le modèle mathématique que l'on obtient.

Il s'agit ensuite de traduire les propriétés voulues dans ce même formalisme, puis de les vérifier avec l'aide de l'assistant de preuve. On considère alors que les propriétés sont les énoncés d'un théorème que l'on essaye de prouver avec le modèle mathématique du programme et celui de son environnement comme axiomatique.

$$Theorem: \quad environment + program \vdash properties \tag{2}$$

L'assistant de preuve fournit alors un certain nombre de lemmes intermédiaires qui permettront de prouver le théorème et donc de certifier que le modèle du programme est valide. Puis, c'est à l'opérateur humain, le plus souvent avec l'aide de l'assistant de preuve, de réaliser les preuves de ces lemmes.

Cette méthode permet de vérifier un très grand nombre de propriétés [Rus99, PM95]. En grande partie parce qu'elle fait ponctuellement appel à l'opérateur humain pour pallier les lacunes de ses stratégies de preuves automatiques. Les étapes triviales étant déduites automatiquement par l'assistant de preuve. Cependant, la vérification par preuve assistée possède deux problèmes majeurs qui ont la même cause : *l'indécidabilité*.

La plupart des théories mathématiques ayant un pouvoir d'expression suffisant pour représenter les langages de programmation incluent inévitablement l'arithmétique. Or le théorème de Gödel établit que toute théorie mathématique contenant l'arithmétique est indécidable. De ce fait découlent deux problèmes insolubles :

1. **Pas de garantie de résultat !**
   Le théorème que l'on veut montrer peut s'avérer indécidable.

2. **L'opérateur humain sera toujours nécessaire**
   Un corrollaire authéorème de Gödel établit qu'il n'existe pas d'algorithme qui puisse énumérer toutes les preuves de l'arithmétique. Cela implique qu'il n'est pas possible de concevoir un logiciel de preuve automatique totalement autonome et qui inclut la théorie de l'arithmétique. Or la théorie de l'arithmétique est requise dans bon nombre de cas. De ce fait, l'assistant de preuve nécessitera toujours une aide humaine.

À l'heure actuelle, on utilise la preuve assistée conjointement avec le *model-checking* par le biais d'abstractions finies du système réalisées de façon automatique afin de réduire la complexité du système et de rester dans le cadre d'une théorie décidable [BLO98, SS99].

2.3 Vérification de modèles (*Model-checking*)

La vérification de modèles correspond à l'ensemble des techniques qui permettent de déduire algorithmiquement des propriétés sur le comportement d'un système (logiciel, algorithme ou protocole) à partir de l'exploration d'un modèle le représentant (automate fini, automate temporisé, réseau de Petri, algèbre de processus, ...).

La première étape consiste à établir un modèle du logiciel, algorithme ou protocole que l'on veut vérifier. Le formalisme de ce modèle est un système de transition étiquetées. Il convient ensuite de traduire les propriétés que l'on veut vérifier en formules logiques (LTL, CTL, CTL*, TCTL, FOL, ...). La dernière étape, c'est à dire la vérification proprement dite, est totalement prise en charge par le logiciel de vérification. Celui-ci va combiner le modèle et la formule logique et calculer l'ensemble des états accessibles en avant ($post^*$ ou '*forward analysis*') ou en arrière (pre^* ou '*backward analysis*'). S'il existe un chemin entre l'état initial et l'ensemble des états qui vérifient la formule, alors la propriété est vérifiée.

Il existe un certain nombre de formalismes permettant de modéliser et de vérifier des propriétés par ce biais :

- Les *réseaux de Petri* [Pet62] dont l'accessibilité est décidable [Kos82, May84] et dont le *model-checking* des formules LTL est décidable [Esp97]. Un certain nombre d'outils utilisent cette théorie comme base, on peut citer : DESIGN-CPN [CJM97], PAPETRI [BJP90] et PEP [Gra97].
- Les *automates à pile* dont l'ensemble des états accessibles est reconnaissable et calculable [FWW97, BEM97] et dont le *model-checking* des formules CTL* est décidable.
- Les *automates temporisés* [AD94] dont le vide est décidable ainsi que le model-checking de TCTL et d'un certain nombre d'autres logiques temporelles temporisées. Plusieurs outils utilisent cette théorie Kronos [DOTY96, BDM$^+$98], Uppaal [BLL$^+$96, LPY97, PL00] et CMC [LL98].
- Les *automates hybrides* [MMP91, Hen96] sont un cas à part. L'accessibilité est indécidable mais certains semi-algorithmes efficaces dans une grande partie des cas réels existe. L'outil le plus connu dans ce domaine étant HyTech [HH95, HHWT97].

La vérification de modèles pose deux problèmes cruciaux. Le premier apparaît lors de la conception du modèle. En effet, il est crucial de conserver une equivalence entre le modèle que l'on va utiliser et la réalité (au moins en ce qui concerne les propriétés que l'on vérifie). De cette équivalence dépend la validité de la preuve que l'on apporte. L'autre problème vient de l'explosion du nombre d'états du modèle que l'on vérifie. Ce problème se rencontre quasiment systématiquement sur les algorithmes de calcul de pre^* ou de $post^*$. C'est bien souvent ce problème qui empêche la vérification de systèmes trop complexes. Il est cependant possible de réduire le nombre des états par le choix d'abstractions qui préservent les propriétés voulues, tout en réduisant le nombre d'états à parcourir. C'est d'ailleurs l'étude d'une de ces abstractions qui motive cette thèse.

3 Contribution de cette thèse

Cette thèse se focalise sur la vérification de modèle, et plus précisément sur le modèle des automates temporisés [AD90, AD94]. Ce modèle permet de s'intéresser à la vérification de systèmes temporisés (*i.e.* incluant des variables temporelles) tout en garantissant la terminaison des algorithmes de *pre** et *post**. Nous proposons ici une extension de ce modèle ainsi qu'une analyse de la décidabilité et de l'expressivité de cette extension.

Le modèle d'Alur et Dill permettait déjà de remettre à zéro la valeur d'une partie des horloges lors d'une transition. Nous ajoutons à ce modèle différents types de mises à jour dont nous étudions l'impact sur la décidabilité du problème du vide et l'expressivité du modèle ainsi obtenu. Nous considérons deux types de mises à jour :

- *Les mises à jour déterministes*
  Qui permettent d'assigner à une horloge une valeur fixe qui est connue au moment où la transition est prise ($x := 2$, $x := y$, $x := y + 2$, $\ldots$),
- *Les mises à jour non déterministes*
  Qui permettent d'assigner à une horloge une valeur choisie de façon non déterministe ($x :< 2$, $x :\geq y$, $x :\neq 2$, $y + 2 \leq: x :< z + 3$).

Ce modèle, et les preuves qui l'accompagnent, ont été réalisé dans un travail en collaboration avec Antoine Petit, Patricia Bouyer et Catherine Dufourd. En septembre 1999, j'ai eu la chance de participer avec Patricia Bouyer, Laurent Fribourg et Antoine Petit à un projet du RNRT nommé CALIFE. Notre travail consistait à définir un modèle qui allait servir de base pour la réalisation d'un outils de vérification. Laurent Fribourg proposa pour cela le modèle des P-automates qu'il avait élaboré avec Béatrice Bérard afin de vérifier le protocole ABR [BF99, BFKM99, BFKM02]. Il nous demanda de réfléchir sur les liens qu'il y avait entre les automates temporisés et les P-automates. C'est au cours de cette étude que nous découvrirent peu à peu le modèle des *updatable timed automata*.

L'étude de la décidabilité de chacune des sous-classes formées par une mise à jour particulière a révélé un certain nombre de surprises. Notamment au niveau du rôle des gardes[12] $x - y \sim c$. Alors que dans le modèle original, les gardes $x - y \sim c$ n'influent aucunement sur les propriétés du modèle, elles jouent un rôle crucial dans le cadre des automates temporisés avec mises à jour. Cependant, une récente découverte [Bou02b] à mis à jour le fait que l'introduction de ces gardes $x - y \sim c$ génère un certain nombre de problèmes au niveau des algorithmes de *post** usuels et confirme l'intuition que nous avions eu.

On exposera au chapitre 3 page 55 les preuves d'indécidabilité qui se résument pour la plupart à une réduction à une machine à deux compteurs (machine de Minsky). Les preuves de décidabilité sont exposées au chapitre 4 page 63. Elles consistent à décrire un algorithme qui construit, à partir d'un automate temporisé avec mises à jour, un graphe des régions fini, puis un automate des régions. Cet algorithme reprend et étend la méthode originale d'Alur et Dill [AD94].

Enfin, nous modéliserons, grâce aux modèles que nous avons établis et en nous restreignant à la partie décidable, des algorithmes de gestion de buffers (FILO, FIFO, et EDF) ainsi qu'un protocole de la famille CSMA/XX (CSMA/CD ou Ethernet). Ce protocole est extrêmement répandu dans la vie courante. Le problème réside dans le fait qu'il repose sur un algorithme qui choisit de façon non déterministe la valeur d'une horloge. Or, cela n'est possible qu'à travers l'utilisation de mises à jour non déterministes. Comme nous le verrons dans cette étude de cas, les mises à jour simplifient grandement la phase de conception et le modèle en lui-même.

[12] Une *garde* est une condition sur une transition.

4 Plan de la thèse

Cette thèse s'articule en sept chapitres. Au *chapitre 1* (page 31), nous rappelons la définition des automates temporisés classiques tels qu'ils ont été décrits par Alur et Dill [AD90, AD94].

Au *chapitre 2* (page 49), nous introduisons les automates temporisés avec mises à jour (*updatable timed automata*) [BDFP00a, BDFP00b]. Nous définissons ici, les mises à jours déterministes et les mises à jours non déterministes.

Au *chapitre 3* (page 55), nous abordons la question de l'indécidabilité du problème du vide de certains fragments du modèle que nous venons de définir.

Au *chapitre 4* (page 63), nous nous intéressons à la décidabilité du problème du vide pour les fragments qui n'ont pas été prouvés indécidables dans le chapitre 3.

Au *chapitre 5* (page 83), nous parlerons de l'expressivité du fragment décidable du modèle que nous venons de définir. Comme nous le verrons les mises à jour déterministes et les mises à jour non déterministes y jouent un rôle très différent les uns des autres.

Au *chapitre 6* (page 107), nous relaterons les résultats que nous avons obtenus sur les automates 0/1 ou *stopwatch automata* en étendant notre modèle. Ce fragment s'avère, hélas, indécidable.

Puis, au *chapitre 7* (page 111), nous décrivons des modèles dans lesquels l'utilisation des automates temporisés avec mises à jour s'avèrent utiles.

Enfin, nous conclurons par une synthèse de ce qu'apportent les résultats présentés et par quelques perspectives.

Chapitre 1

Les automates temporisés

The Rabbit say to itself : "Oh dear ! Oh dear ! I shall be too late !"
— Lewis Carroll, *Alice in Wonderland*

Les automates temporisés ont été introduits par Alur et Dill en 1990 [AD90, AD94]. Ce modèle, largement étudié [AHV93, ACH94, Wil94] et étendu depuis [BDGP98, DZ98, CG00], permet de faire de la vérification de propriétés de sûreté sur des systèmes de transitions qui utilisent des variables temporelles à valeurs dans $\mathbb{N}$, $\mathbb{Q}_+$ ou même $\mathbb{R}_+$. Les automates temporisés ont de plus été implantés dans plusieurs logiciels de vérification (Kronos [DOTY96, BDM+98], Uppaal [BLL+96, LPY97, PL00], CMC [LL98, CL00a]) et ont permis de valider ou corriger un certain nombre de protocoles ou algorithmes qui utilisent des variables temporelles dans leur déroulement [DOY94, DOY95, JLS96, HSLL97].

Nous introduisons dans ce chapitre les automates temporisés 'classiques' tels qu'ils sont utilisés dans les outils de vérification. Contrairement au modèle original d'Alur et Dill [AD94], le modèle 'classique' utilise des gardes diagonales ce qui rend le modèle plus complexe. Nous commencerons par définir quelques notions de bases relatives aux automates temporisés (langages temporisés, horloges, gardes, automates de Büchi), puis nous définirons les automates temporisés 'classiques' proprement dit. Enfin, nous nous intéresserons aux propriétés de ce modèle et plus particulièrement au problème du vide.

1.1 Langage temporisé

Un langage temporisé associe à chaque lettre (action) d'un mot, une date. Les mots sont donc formés d'une suite de couples comprenant une lettre choisie dans un alphabet fini Σ et un temps à valeur dans un domaine de temps $\mathbb{T}$, qui sera égal à $\mathbb{N}$, $\mathbb{Q}_+$ ou $\mathbb{R}_+$.

Notation 1. *Soit Z un ensemble quelconque, alors Z^* (resp. Z^ω) est l'ensemble des séquences finies (resp. infinies) d'éléments de Z. On note $Z^\infty = Z^* \cup Z^\omega$.*

Une *séquence temporisée* sur $\mathbb{T}$ est une séquence croissante de $\mathbb{T}^\infty$. Un *mot temporisé* ω est une séquence de couples $(a_i, t_i)_{i>0} \in (\Sigma \times \mathbb{T})^\infty$ telle que $(t_i)_{i>0}$ soit une séquence temporisée. Finalement, un langage temporisé L de Σ^∞ est la donnée d'un sous-ensemble de $(\Sigma \times \mathbb{T})^\infty$.

Définition 1. *Soit L un langage temporisé de $(\Sigma \times \mathbb{T})^\infty$, on appelle $Untimed(L)$ le langage défini par :*

$$Untimed(L) = \{(a_i)_{i>0} \in \Sigma^\infty \mid \exists (t_i)_{i>0} \text{ tel que } (a_i, t_i)_{i>0} \in L\} \tag{1.1}$$

Exemple 1. *Voici quelques exemples de mots temporisés sur l'alphabet $\Sigma = \{a, b, c\}$ et sur le domaine de temps $\mathbb{T} = \mathbb{R}_+$.*

1. $(b, 4)(a, 4.75)(c, 6.3)(a, 12)$,
2. $(a, e)(c, \pi)(a, 2 \times \pi)$,
3. $(a, 2)(a, 4)(a, 6)(a, 8) \ldots$,
4. $(a, 1)(b, 2)(c, 3)(a, 12)(b, 13)(c, 14)(a, 102)(b, 103)(c, 104) \ldots$.

1.2 Horloges et gardes

On appelle *horloges* des variables à valeurs dans le domaine de temps $\mathbb{T}$, qui évoluent de manière synchrone dans le temps. On considère X un ensemble fini d'horloges.

L'ensemble des *gardes* (ou *contraintes*) $\mathcal{C}(X)$ sur l'ensemble des horloges X est défini par la grammaire suivante :

$$\begin{gathered}\varphi ::= x \sim c \mid x - y \sim c \mid \varphi \wedge \varphi \mid true, \\ \text{avec } x, y \in X, c \in \mathbb{Q}_+, \sim \in \{<, \leq, =, \neq, \geq, >\}\end{gathered} \tag{1.2}$$

On notera cet ensemble $\mathcal{C}$, lorsqu'il n'y aura pas d'ambiguïté.

Remarque 1. *Il est à noter que les automates temporisés introduit par Alur et Dill à l'origine [AD90, AD94] ne possédaient pas de garde diagonales ($x - y \sim c$). Cependant, il fut prouvé par la suite que le modèle qui possédait des gardes diagonales et celui qui n'en possédait pas étaient* langage-équivalent *[BDGP98]. Malgré ce résultat, il est important pour la suite de distinguer ici les gardes générales, que nous venons de voir, des gardes non-diagonales.*

On appelle *gardes non diagonales* (*diagonal free guards*), les gardes qui n'incluent pas de contraintes sur des différences d'horloges. Plus formellement, l'ensemble $\mathcal{C}_{df}(X)$ des gardes diagonales est défini par la grammaire :

$$\begin{gathered}\varphi_{df} ::= x \sim c \mid \varphi_{df} \wedge \varphi_{df} \mid true \\ \text{avec } x \in X, c \in \mathbb{Q}_+, \sim \in \{<, \leq, =, \neq, \geq, >\}\end{gathered} \tag{1.3}$$

Exemples 2. *Voici quelques exemples de gardes sur $X = \{x, y, z\}$.*

1. $x = 5$,
2. $x < y$,
3. $x - y < 4$,
4. $(x < 10) \wedge (y - z > 2) \wedge (x - y = 3)$.

Notations 2. *Soit X un ensemble fini d'horloges.*

- *Une fonction $v : X \to \mathbb{T}$ est appelée une* valuation sur les horloges,

– *Si $v \in \mathbb{T}^X$ est une valuation et $t \in \mathbb{T}$, alors $v + t \in \mathbb{T}^X$ est la valuation qui correspond à l'écoulement d'une durée t, et qui est définie par :*

$$(v + t)(x) = v(x) + t, \; \forall x \in X \tag{1.4}$$

– *Si v est une valuation telle que $(v(x))_{x \in X}$ satisfait la garde φ, on dit que v* vérifie *φ et on note $v \models \varphi$,*

– *Soient $Y \subseteq X$ et $f : Y \to \mathbb{T}$, alors la valuation $v[x \leftarrow f(x)/x \in Y]$ est définie par :*

$$v[x \leftarrow f(x)/x \in Y](y) = \begin{cases} v(y), & si\ y \notin Y \\ f(y), & si\ y \in Y \end{cases} \tag{1.5}$$

1.3 Automates de Büchi

Avant de définir des automates permettant de reconnaître des langages temporisés, nous rappelons brièvement la notion classique d'automate de Büchi [Büc62, MN66, Tho90] pour les langages non temporisés.

Un *automate de Büchi* est un système de transitions défini par $\mathcal{B} = (\Sigma, Q, T, I, R)$, avec Σ un alphabet fini, Q un ensemble fini d'états, $T \subseteq Q \times \Sigma \times Q$ un ensemble fini de transitions, $I \subseteq Q$ un ensemble d'états initiaux, $R \subseteq Q$ un ensemble d'états répétés.

Soit un mot infini $\sigma = \sigma_1\sigma_2\ldots$ avec $\sigma_i \in \Sigma$ pour tout $i > 0$. On appelle r une *exécution* (*run*) de $\mathcal{B}$ sur σ une suite :

$$r = q_0 \xrightarrow{\sigma_1} q_1 \xrightarrow{\sigma_2} q_2 \xrightarrow{\sigma_3} q_3 \ldots, \text{ avec } q_0 \in I, \text{ et } \forall i > 0, (q_{i-1}, \sigma_i, q_i) \in T \tag{1.6}$$

On note $rep(r) \subseteq Q$, l'ensemble des états infiniment répétés durant l'exécution r. Une exécution r est *acceptante* si et seulement si $rep(r) \cap R \neq \emptyset$ (*condition de Büchi*). Le mot $\sigma = \sigma_1\sigma_2\sigma_3\ldots$ est *accepté* par $\mathcal{B}$ s'il existe une exécution acceptante de $\mathcal{B}$ sur σ. Le langage des mots acceptés par $\mathcal{B}$ est noté $L(\mathcal{B})$.

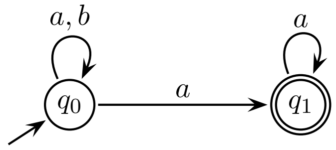

FIG. 1.1 – *Exemple d'automate de Büchi.*

Exemple 3. *Soit $\mathcal{B} = (\Sigma, Q, T, I, R)$ un automate de Büchi avec $\Sigma = \{a, b\}$, $Q = \{q_0, q_1\}$, $T = \{(q_0, a, q_0), (q_0, b, q_0), (q_0, a, q_1), (q_1, a, q_1)\}$, $I = \{q_0\}$ et $R = \{q_1\}$ (voir figure 1.1). Alors, $\mathcal{B}$ accepte le langage $(a + b)^*a^\omega$, constitué des mots infinis ayant un nombre fini de b.*

1.4 Automates temporisés

Un *automate temporisé* est un système de transitions défini par $\mathcal{A} = (\Sigma, Q, T, I, F, R, X)$ avec Σ un alphabet fini d'*actions*, Q un ensemble fini d'*états*, X un ensemble fini d'*horloges*, $I \subseteq Q$ l'ensemble des *états initiaux*, $F \subseteq Q$ l'ensemble des *états finaux*, $R \subseteq Q$ l'ensemble des *états répétés*, et $T \subseteq Q \times [\mathcal{C}(X) \times \Sigma \times \mathcal{P}(X)] \times Q$ un ensemble fini de *transitions* (avec $\mathcal{P}(X)$ l'ensemble des parties de X.).

Ainsi, une transition est définie par un 5-uplet (q, g, a, R, q'), avec :

- $q, q' \in Q$ deux états,
- $g \in \mathcal{C}(X)$ une *garde*,
- $a \in \Sigma$ une *action*,
- $R \in \mathcal{P}(X)$ un sous-ensemble d'horloges *remises à zéro.*

Afin de caractériser les mots temporisés acceptés à partir du système de transitions que nous venons de définir, nous introduisons la notion de *chemin* et d'*exécution* sur un automate temporisé.

On appelle p (*path*) un *chemin* dans $\mathcal{A}$ une suite finie ou infinie de la forme :

$$\begin{gathered} p = q_0 \xrightarrow{g_1,\sigma_1,R_1} q_1 \xrightarrow{g_2,\sigma_2,R_2} q_2 \xrightarrow{g_3,\sigma_3,R_3} q_3 \ldots, \\ \text{avec } q_0 \in I, \text{ et } \forall i > 0, (q_{i-1}, g_i, \sigma_i, R_i, q_i) \in T \end{gathered} \tag{1.7}$$

On note $rep(p) \subseteq Q$ l'ensemble des états qui sont infiniment souvent répétés sur le chemin p. Le chemin fini (resp. infini) p est *acceptant* si l'état final est dans F (resp. si $rep(p) \cap R \neq \emptyset$, *condition de Büchi*).

Une *exécution* r (*run*) sur le chemin p est une suite de la forme :

$$r = \langle q_0, v_0 \rangle \xrightarrow[t_1]{g_1,\sigma_1,R_1} \langle q_1, v_1 \rangle \xrightarrow[t_2]{g_2,\sigma_2,R_2} \langle q_2, v_2 \rangle \ldots, \text{ avec } \forall i \geq 0, v_i \in \mathbb{T}^X \tag{1.8}$$

avec $(t_i)_{i>0}$ une séquence temporisée et $(v_i)_{i\geq 0}$ des valuations d'horloges telles que :

- $v_0(x) = 0, \forall x \in X$,
- $\forall i > 0, v_{i-1} + (t_i - t_{i-1}) \models g_i$,
- $\forall i > 0$ et $\forall x \in X$, $v_i(x) = \begin{cases} 0, & \textit{si } x \in R_i \\ v_{i-1}(x) + (t_i - t_{i-1}), & \textit{sinon} \end{cases}$

On appelle les couples $\langle q_i, v_i \rangle$, des *états étendus*.

L'étiquette de r est le mot temporisé $(\sigma_1, t_1)(\sigma_2, t_2) \ldots$, qui est *accepté* par l'automate temporisé $\mathcal{A}$. L'ensemble des mots qui sont l'étiquette d'une exécution sur un chemin acceptant de $\mathcal{A}$ forment le langage accepté (ou reconnu) par $\mathcal{A}$ et on le note $L(\mathcal{A}, \mathbb{T})$ ou plus simplement $L(\mathcal{A})$ lorsqu'il n'y a pas d'ambiguïté.

Remarque 2. *Dans ce modèle, il est à noter que l'on peut simuler tout automate temporisé utilisant des gardes générales par un automate temporisé utilisant uniquement des gardes non diagonales [BDGP98].*

Comme nous le verrons par la suite, cette simulation n'est plus possible dans le cadre du modèle des automates temporisés avec mises à jour (voir chapitre 2).

Exemple 4. *L'automate de la figure 1.2 page suivante est un exemple d'automate temporisé.*

Pour des raisons de lisibilité, lorsque la garde vaut true ou que l'ensemble des mises à jour vaut $\emptyset$, on n'écrit pas la composante en question.

Voici quelques exemples de mots acceptés par l'automate représenté sur la figure 1.2 :

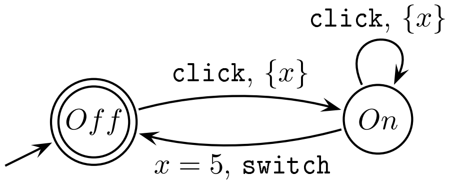

FIG. 1.2 – *Automate temporisé modélisant une minuterie.*

- *Le mot temporisé est :*

$$(\mathtt{click}, 0)(\mathtt{click}, 1)(\mathtt{switch}, 6)\dots(\mathtt{click}, 7n)(\mathtt{click}, 7n{+}1)(\mathtt{switch}, 7n{+}6)\dots$$

  Une exécution acceptante est :

$$\langle Off, 0\rangle \xrightarrow[0]{\mathtt{click},\{x\}} \langle On, 0\rangle \xrightarrow[1]{\mathtt{click},\{x\}} \langle On, 0\rangle \xrightarrow[6]{x=5,\mathtt{switch}} \langle Off, 5\rangle \dots$$

- *Le mot temporisé est :*

$$(\mathtt{click}, 0)(\mathtt{click}, \sqrt{2})\dots(\mathtt{switch}, 5{+}\sqrt{2})(\mathtt{click}, 7n)(\mathtt{click}, 7n{+}\sqrt{2})(\mathtt{switch}, 7n{+}5{+}\sqrt{2})\dots$$

  Une exécution acceptante est :

$$\langle Off, 0\rangle \xrightarrow[0]{\mathtt{click},\{x\}} \langle On, 0\rangle \xrightarrow[\sqrt{2}]{\mathtt{click},\{x\}} \langle On, 0\rangle \xrightarrow[5+\sqrt{2}]{x=5,\mathtt{switch}} \langle Off, 5\rangle \dots$$

Remarque 3. *Une variante du modèle des automates temporisés classiques consiste à introduire des actions silencieuses dans l'alphabet utilisé. Plus précisément, on considère des transitions dans $Q \times \mathcal{C}(X) \times (\Sigma \cup \{\varepsilon\}) \times \mathcal{P}(X) \times Q$. On appelle ce modèle les* automates temporisés avec ε-transitions *[AD90]. Ce modèle est strictement plus expressif que le modèle classique [BDGP98] et diffère sur la complexité de certains problèmes (voir section 1.5).*

Plus formellement, un automate temporisé avec ε-transitions est un automate temporisé classique tel que $\mathcal{A} = (\Sigma \cup \{\varepsilon\}, Q, T, I, F, R, X)$ avec ε une action silencieuse et dont le langage acceptant $L(\mathcal{A})$ est l'ensemble des mots temporisés obtenus en projetant les étiquettes des exécutions acceptantes sur $(\Sigma \times \mathbb{T})^\infty$. Autrement dit, en supprimant toutes les lettres du type (ε, t) avec $t \in \mathbb{T}$, des mots acceptés. On peut remarquer qu'avec cette définition une exécution infinie peut correspondre à un mot temporisé fini dans le langage accepté.

1.5 Propriétés des automates temporisés

Le modèle que nous venons de définir a certaines propriétés intéressantes que nous résumons brièvement ci-dessous. La problème du vide sera repris plus en détails dans la section suivante.

- **Problème du vide** : Pour tout automate temporisé $\mathcal{A}$, le *problème du vide* consiste à décider de la valeur de vérité de l'assertion $L(\mathcal{A}) = \emptyset$.

  Ce problème a été montré *Pspace*-complet pour la classe des automates temporisés classiques [AD94],

- **Universalité** : Ce problème est le dual du problème du vide. Pour tout automate temporisé $\mathcal{A}$, le *problème de l'universalité* consiste à décider la valeur de vérité de l'assertion $L(\mathcal{A}) = (\Sigma \times \mathbb{T})^\infty$.
  Ce problème a été montré *indécidable* pour la classe des automates temporisés classiques [AD94],
- **Inclusion de langage** : Pour tout couple d'automates temporisés $\mathcal{A}$ et $\mathcal{B}$, le *problème de l'inclusion des langages* est de décider la valeur de vérité de l'assertion $L(\mathcal{A}) \subseteq L(\mathcal{B})$.
  Ce problème a été montré *indécidable* pour la classe des automates temporisés classiques [AD94] (on se ramène au problème de l'universalité),
- **Équivalence de langage** : Pour tout couple d'automates temporisés $\mathcal{A}$ et $\mathcal{B}$ et leurs langages associés $L(\mathcal{A})$ et $L(\mathcal{B})$, le *problème de l'équivalence des langages* est de décider la valeur de vérité de l'assertion $L(\mathcal{A}) = L(\mathcal{B})$.
  Ce problème a été montré *indécidable* pour la classe des automates temporisés classiques [AD94] (on se ramène à un problème de double inclusion des langages),
- **Traces non temporisées** : Pour tout automate temporisé $\mathcal{A}$ et un mot σ tel que $\sigma = \sigma_1\sigma_2\sigma_3 \cdots \in \Sigma^*$. Le *problème des traces non temporisées* est de vérifier que l'on a $\sigma \in \mathit{Untimed}(\mathcal{A})$. Autrement dit, qu'il existe une exécution temporisée r sur $\mathcal{A}$ (voir équation page 34) telle que :
  $$r = q_0 \xrightarrow[t_1]{g_1,\sigma_1,R_1} q_1 \xrightarrow[t_2]{g_2,\sigma_2,R_2} q_2 \xrightarrow[t_3]{g_3,\sigma_3,R_3} q_3 \ldots \tag{1.9}$$
  Ce problème a été montré *NP*-complet pour la classe des automates temporisés classiques [AKV98] et *Pspace*-complet pour la classe des automates temporisés avec ε-transitions [AKV98],
- **Traces temporisées** : Pour tout automate temporisé $\mathcal{A}$, un mot $\sigma = \sigma_1\sigma_2 \cdots \in \Sigma^*$ et une séquence temporisée $\tau = t_1t_2 \ldots$ (voir problème page 31). Le *problème des traces temporisées* est de vérifier que l'on a $(\sigma, t) \in L(\mathcal{A})$. Autrement dit, qu'il existe une exécution temporisée r sur $\mathcal{A}$ (voir équation page 34) telle que :
  $$r = q_0 \xrightarrow[t_1]{g_1,\sigma_1,R_1} q_1 \xrightarrow[t_2]{g_2,\sigma_2,R_2} q_2 \xrightarrow[t_3]{g_3,\sigma_3,R_3} q_3 \ldots \tag{1.10}$$
  Ce problème a été montré *NP*-complet pour la classe des automates temporisés classiques [AKV98] et *Pspace*-complet pour la classe des automates temporisés contenant des ε-transitions [AKV98],
- **Génération d'estampillage temporel** : Pour tout automate temporisé $\mathcal{A}$, et un chemin p sur $\mathcal{A}$ (voir équation page 34), tel que :
  $$\begin{gathered} p = q_0 \xrightarrow{g_1,\sigma_1,R_1} q_1 \xrightarrow{g_2,\sigma_2,R_2} q_2 \xrightarrow{g_3,\sigma_3,R_3} q_3 \ldots \xrightarrow{g_n,\sigma_n,R_n} q_n, \\ \text{avec } q_0 \in I, \text{ et } \forall i \leq n,\ (q_{i-1}, g_i, \sigma_i, R_i, q_i) \in T \end{gathered} \tag{1.11}$$
  Le *problème de la génération d'estampillage temporel* est de vérifier s'il existe une exécution temporisée r sur $\mathcal{A}$ (voir équation page 34) telle que :
  $$r = q_0 \xrightarrow[t_1]{g_1,\sigma_1,R_1} q_1 \xrightarrow[t_2]{g_2,\sigma_2,R_2} q_2 \xrightarrow[t_3]{g_3,\sigma_3,R_3} q_3 \ldots \xrightarrow{g_n,\sigma_n,R_n} q_n \tag{1.12}$$
  Ce problème a été montré de complexité $O(n.m^2)$, avec n la longueur du mot et et m le nombre des horloges, pour la classe des automates temporisés classiques et celle des automates temporisés avec ε-transitions [AKV98].

1.6 Le problème du vide

Le problème du vide (voir section 1.5 page 35) est un problème majeur en vérification. Il permet, notamment de vérifier l'accessibilité d'un état sur un automate. Ce test est très utile dans la pratique pour vérifier la présence ou l'absence de certains comportements dans un modèle.

Notre but est de démontrer la décidabilité du problème du vide sur la classe des automates temporisés. Pour ce faire, nous allons montrer que nous pouvons toujours construire, à partir d'un automate temporisé quelconque, un automate de Büchi qui reconnaît le langage *atemporisé* (*untimed*, voir définition 1 page 31) de l'automate temporisé original. Étant donné que le problème du vide est décidable sur les automates de Büchi, on en déduit le théorème attendu. On appellera cet automate de Büchi particulier l'*automate des régions*. La preuve de ce résultat a été proposée par Alur et Dill [AD90, AD94]. Nous présentons ici une généralisation de cette preuve aux automates temporisés avec gardes diagonales par le biais des *régions d'horloges*.

Le but de cette section est de décrire une méthode de construction d'un automate des régions à partir d'un automate temporisé, tout en prouvant qu'il existe *toujours* un tel automate, quelque soit l'automate temporisé initial.

Nous commençons par introduire les *régions d'horloges*, puis le *graphe des régions*, et nous finirons par la construction de l'*automate des régions* proprement dit. Grâce à la remarque suivante, nous pouvons nous restreindre aux automates temporisés dont les gardes n'utilisent que des constantes entières.

Remarque 4. *Nous avons défini les contraintes sur les horloges par des comparaisons avec des constantes dans $\mathbb{Q}$. Pour étudier le problème du vide, on peut se restreindre à des constantes dans $\mathbb{Z}$. En effet, soit $\mathcal{A}$ un automate temporisé et soit d le ppcm (plus petit commun multiple) des dénominateurs des constantes apparaissant dans $\mathcal{A}$. Enfin, soit $\mathcal{A}_d$ l'automate temporisé obtenu à partir de $\mathcal{A}$ en multipliant toutes les constantes de $\mathcal{A}$ par d. Par construction, les constantes de $\mathcal{A}_d$ sont toutes entières. De plus, le mot temporisé $(a_i, t_i)_{i>0}$ est accepté par $\mathcal{A}$ si et seulement si $(a_i, d.t_i)_{i>0}$ est accepté par $\mathcal{A}_d$ [AD94]. Ainsi, $L(\mathcal{A})$ est vide si et seulement si $L(\mathcal{A}_d)$ est vide.*

1.6.1 Les régions d'horloges

Vérifier le vide sur un automate temporisé consiste à tester si *aucune* exécution étendue n'est acceptée. Or, on peut voir facilement qu'il existe un nombre infini de ces exécutions. Pour contourner cette difficulté nous allons abstraire le problème. Nous ne considérerons plus les valuations une à une, mais par groupes de valuations équivalentes. On appellera ces groupes de valuations des *régions*.

Plus précisément, une *région* est un ensemble de valuations qui sont équivalentes du point de vue des gardes de l'automate temporisé que l'on considère. On dit alors que l'ensemble de régions est *compatible* avec l'ensemble des gardes. Dans notre cas, nous allons étudier une partition particulière de $\mathbb{T}^X$ qui forme un ensemble de régions que nous montrerons compatibles avec n'importe quel ensemble de contraintes.

On se donne un ensemble fini d'horloges X et $\lambda = ((max_x)_{x\in X}, (max_{x,y})_{(x,y)\in X^2})$ un ensemble de constantes entières qui représentent respectivement les plus grandes contraintes sur les horloges (max_x) et sur les différences d'horloges ($max_{x,y}$). Nous allons construire une partition $\mathcal{R}_\lambda$ de $\mathbb{T}^X$. On définit deux types de contraintes sur les valuations (les *contraintes simples*, les *contraintes diagonales*) dépendant de λ. Les régions seront ensuite définies par la conjonction de ces contraintes.

Définition 2. *Soient un ensemble fini d'horloges X et un ensemble de constantes entières $\lambda = ((max_x)_{x\in X}, (max_{x,y})_{(x,y)\in X^2})$. Alors on appelle* contrainte simple *associée à λ une formule φ_x telle que :*

$$\begin{array}{rll} \varphi_x ::= & & x = c \\ & | & c < x < c+1 \\ & | & max_x < x \\ & & \\ & & Avec\ c < max_x. \end{array} \tag{1.13}$$

Une contrainte diagonale *associée à λ est une formule $\varphi_{x,y}$ avec $(x,y) \in X^2$ telle que :*

$$\begin{array}{rll} \varphi_{x,y} ::= & & max_{x,y} < x - y \\ & | & c < x - y < c+1 \\ & | & x - y = c \\ & | & x - y < -max_{y,x} \\ & & \\ & & Avec\ \ -max_{y,x} < c < max_{x,y}. \end{array} \tag{1.14}$$

Nous allons introduire quelques ensembles utiles par la suite.

Notations 3. *Soient X un ensemble fini d'horloges, $\lambda = ((max_x)_{x\in X}, (max_{x,y})_{(x,y)\in X^2})$ un ensemble de constantes entières, et $((\varphi_x)_{x\in X}, (\varphi_{x,y})_{(x,y)\in X^2})$ un ensemble de contraintes issues de λ. On pose :*

- $X_0 = \{x \in X \mid \varphi_x = (v(x) = c)\}$ *l'ensemble des horloges dont la partie fractionnaire est* nulle *et dont la valeur est inférieure à max_x,*
- $X_{frac} = \{x \in X \mid \varphi_x = (c < v(x) < c+1)\}$, *l'ensemble des horloges dont la partie fractionnaire est* non nulle *et dont la valeur est inférieure à max_x.*
- $X_\infty = \{x \in X \mid \varphi_x = (v(x) > max_x)\}$, *l'ensemble des horloges dont la valeur est* supérieure à max_x.

Enfin, on définit une région d'horloges comme suit.

Définition 3. *Soient un ensemble fini d'horloges X et un ensemble de constantes entières $\lambda = ((max_x)_{x\in X}, (max_{x,y})_{(x,y)\in X^2})$. Alors, une* région *est la donnée de :*

- $(\varphi_x)_{x\in X}$ *un ensemble de contraintes simples associées à λ,*
- $(\varphi_{x,y})_{(x,y)\in X^2}$ *un ensemble de contraintes diagonales associées à λ,*
- $\succ$ *un préordre total sur l'ensemble des horloges $x \in X_{frac}$.*

La région $((\varphi_x)_{x\in X}, (\varphi_{x,y})_{(x,y)\in X^2}, \succ)$ représente l'ensemble de valuations $\alpha \subset \mathbb{T}^X$, tel que :

$$\alpha = \left\{ v \in \mathbb{T}^X \;\middle|\; \begin{array}{l} \forall x \in X,\ v(x) \models \varphi_x, \\ \forall (x,y) \in X^2,\ avec\ x{\in}X_\infty\ ou\ y{\in}X_\infty,\ v(x){-}v(y) \models \varphi_{x,y}, \\ \forall x,y{\in}\, X_{frac},\ x \succ y \Leftrightarrow frac(v(x)) \geq frac(v(y)) \end{array} \right\} \tag{1.15}$$

Avec, $frac(t) = t - \lfloor t \rfloor$ la partie fractionnaire de t.

Enfin, $\mathcal{R}_\lambda$ est la partition finie formée par toutes les régions engendrées par λ. Par commodité, nous noterons $\alpha = ((\varphi_x)_{x\in X}, (\varphi_{x,y})_{(x,y)\in X^2}, \succ)$.

Exemple 5. *Soit* $X = \{x, y\}$ *un ensemble d'horloges et* $C = \{x < 2, y - x > 1, y = 1\}$ *un ensemble de contraintes sur les horloges. On a* $max_x = 2$ *et* $max_y = 1$*, les deux constantes maximales respectivement sur* x *et sur* y *ainsi que* $max_{x,y} = max_{y,x} = 1$.

La partition $\mathcal{R}_\lambda$ *associée est représentée sur la figure 1.3. On peut y distinguer trois types de régions :*

- *Les régions ponctuelles, qui sont au nombre de neuf,*
- *Les régions linéaires, qui sont au nombre de vingt-trois,*
- *Les régions surfaciques, qui sont au nombre de quatorze.*

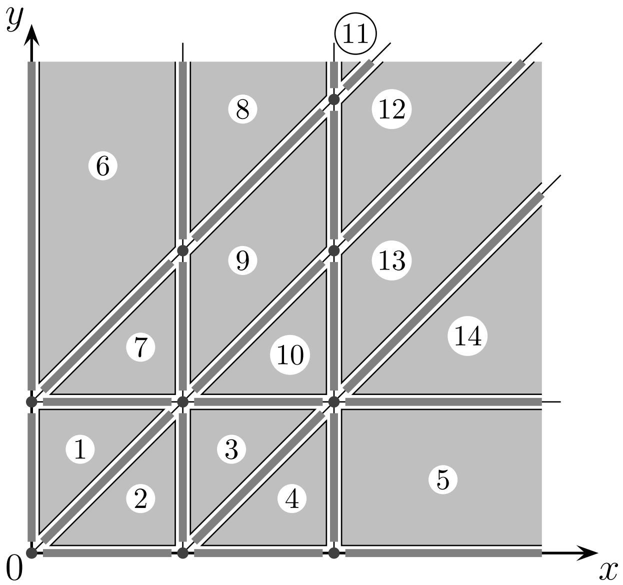

FIG. 1.3 – *Un exemple de régions avec deux horloges* x *et* y.

Voici quelques exemples de régions :

Région ponctuelle : $\alpha_{(2,1)} = \{(x = 2) \wedge (y = 1),\ \emptyset\}$,
Région linéaire bornée : $\alpha_{(1,1),(2,2)} = \{(1 < x < 2) \wedge (y > 1), \{(x \succ y), (y \succ x)\}\}$,
Région linéaire non bornée : $\alpha_{(2,1),(\infty,\infty)} = \{(x > 2) \wedge (y > 1) \wedge (x - y = 1), \{y \succ x\}\}$,
Région surfacique bornée : $\alpha_4 = \{(1 < x < 2) \wedge (0 < y < 1), \{x \succ y\}\}$,
Région surfacique non bornée : $\alpha_8 = \{(1 < x < 2) \wedge (y > 1) \wedge (x - y > 1), \{x \succ y\}\}$.

Nous allons à présent introduire la notion de *compatibilité* entre un ensemble de régions et des contraintes sur les horloges. Le but est de s'assurer que les régions ne regroupent que des valuations qui ont des comportements similaires face aux contraintes d'horloges que l'on se donne.

Définition 4. *Soit* X *un ensemble fini d'horloges,* $C \subseteq \mathcal{C}(X)$ *un ensemble de contraintes et* $\mathcal{R}$ *un ensemble fini de régions sur* $\mathbb{T}^X$*. Alors,* $\mathcal{R}$ *est* compatible *avec* C *si on a :*

$$\forall \psi \in C,\ \forall \alpha \in \mathcal{R},\ (\alpha \models \psi) \text{ ou } (\alpha \models \neg\psi) \tag{1.16}$$

Nous allons nous intéresser maintenant au cas particulier de la partition $\mathcal{R}_\lambda$ de $\mathbb{T}^X$ que nous venons de définir. La proposition suivante est immédiate de par la définition de $\mathcal{C}(X)$ et de $\mathcal{R}_\lambda$.

Proposition 1. *Soient X, un ensemble fini d'horloges, $C \subseteq \mathcal{C}(X)$ un ensemble de contraintes sur les horloges et $\lambda = ((max_x)_{x \in X}, (max_{x,y})_{(x,y) \in X^2})$ un ensemble de constantes entières telles que :*

- *$\forall x \in X$, $max_x = \max\{c \mid (x \sim c) \in C\}$,*
- *$\forall (x,y) \in X^2$, on pose $C_{x,y}$ comme étant l'ensemble des contraintes $(x - y \sim c)$ et des contraintes $(y - x \sim c)$. Plus formellement, $C_{x,y} = \{(x - y \sim c) \in C \text{ et } (y - x \sim c) \in C\}$, et on définit :*

$$max_{x,y} = \begin{cases} \max\{c \mid (x - y \sim c) \in C_{x,y} \text{ et } (y - x \sim c) \in C_{x,y}\}, & \text{si } C_{x,y} \neq \emptyset \\ 0, & \text{si } C_{x,y} = \emptyset \end{cases}$$

Alors, $\mathcal{R}_\lambda$ est compatible avec C.

1.6.2 Le graphe des régions

Soit $\mathcal{R}$ un ensemble fini de régions formant une partition de $\mathbb{T}^X$. Le *graphe des régions* $\mathcal{G}$ associé à $\mathcal{R}$ établit les relations de précédence qu'ont les régions entre elles. Il est défini par $\mathcal{G} = (\mathcal{R}, \rightarrow)$ où la fonction de transition $\rightarrow \subseteq \mathcal{R} \times \mathcal{R}$ est définie à partir des fonctions de transitions suivantes :

$\xrightarrow{time}$: représente l'écoulement du temps :

$$\forall \alpha, \alpha' \in \mathcal{R},\ \alpha \xrightarrow{time} \alpha' \quad \overset{def}{\Longleftrightarrow} \quad \exists v \in \alpha,\ \exists t > 0,\ v + t \in \alpha' \tag{1.17}$$

$\xrightarrow{reset}$: représente les remises à zéro d'horloges de X :

$$\forall \alpha, \alpha' \in \mathcal{R},\ \alpha \xrightarrow{reset} \alpha' \quad \overset{def}{\Longleftrightarrow} \quad \begin{array}{l} \exists v \in \alpha,\ \exists Y \subseteq X,\ \exists v' \in \alpha', \\ \text{tel que } v' = v[x \leftarrow 0 / x \in Y]. \end{array} \tag{1.18}$$

La fonction de transitions $\rightarrow$ du graphe des régions $\mathcal{G} = (\mathcal{R}, \rightarrow)$ est définie par l'union des transitions *time* et *reset* :

$$\rightarrow = \xrightarrow{time} \cup \xrightarrow{reset} \tag{1.19}$$

Exemple 6. *Si l'on prend l'exemple de la figure 1.3 page précédente, le futur de la région surfacique* 3*, c'est à dire les régions accessibles via des transitions de type* time*, est constitué des régions suivantes :*

1. *région* $](1,1),(2,1)[$,
2. *région* 10,
3. *région* $](2,1),(2,2)[$,
4. *région* 13.

De même, les remises à zéro de la région surfacique 3*, c'est à dire les régions accessibles via une transition* reset*, sont les suivantes :*

- *Pour $Y = \{x\}$, on a la région* $](1,0),(2,0)[$,
- *Pour $Y = \{y\}$, on a la région* $](0,0),(0,1)[$,

– *Pour* $Y = \{x, y\}$*, on a la région* $\{(0, 0)\}$

Cette définition du graphe des régions est cependant très générale. Notre but est à présent de contraindre le graphe des régions $\mathcal{G}$ afin de s'assurer que l'automate des régions $\mathcal{A}_R$ que nous allons construire capture l'ensemble de tous les comportements de l'automate original $\mathcal{A}$. À la fois pour les comportements temporisés et pour les "*remise à zéro*". Pour cela nous introduisons les conditions suivantes :

$$\forall \alpha, \alpha' \in \mathcal{R},\ \alpha \xrightarrow{time} \alpha' \Longrightarrow \forall v \in \alpha, \exists t \geq 0, v + t \in \alpha' \tag{$\dagger_1$}$$

$$\forall \alpha, \alpha' \in \mathcal{R},\ \alpha \xrightarrow{reset} \alpha' \Longrightarrow \forall v \in \alpha,\ \exists Y \subseteq X,\ \exists v' \in \alpha', \text{ tel que } v' = v[x \leftarrow 0/x \in Y]. \tag{$\dagger_2$}$$

Il est immédiat que l'ensemble de tous les graphes des régions possibles ne vérifient pas forcément ($\dagger_1$) et ($\dagger_2$). ($\dagger_1$) est une condition assez naturelle, elle exprime le fait que deux valuations d'une région sont équivalentes et indistinguables vis-à-vis de l'écoulement du temps. La condition ($\dagger_2$) exprime, quant à elle, le fait que deux valuations d'une région sont équivalentes et indistinguables vis-à-vis d'une transition *reset*.

Ces deux conditions servent d'hypothèses de bases dans la démonstration du théorème 3 qui montre la décidabilité du problème du vide pour la classe des automates temporisés. Dans le cas des graphes des régions de type $\mathcal{G}_\lambda = (\mathcal{R}_\lambda, \rightarrow)$ où $\mathcal{R}_\lambda$ est la partition définie comme précédemment (voir définition 3 page 38), on peut montrer que ($\dagger_1$) et ($\dagger_2$) sont vraies moyennant une hypothèse sur les contraintes C.

Théorème 2. *Soient X un ensemble fini d'horloges et $\lambda = ((max_x)_{x \in X}, (max_{x,y})_{(x,y) \in X^2})$, un ensemble de constantes entières tel que pour tout couple $(x, y) \in X^2$, on ait $max_{x,y} \leq max_x$. Alors, le graphe des régions $\mathcal{G}_\lambda = (\mathcal{R}_\lambda, \rightarrow)$ vérifie les conditions ($\dagger_1$) et ($\dagger_2$).*

Nous montrons ($\dagger_1$) puis ($\dagger_2$) pour $\mathcal{G}_\lambda = (\mathcal{R}_\lambda, \rightarrow)$ avec $\lambda = ((max_x)_{x \in X}, (max_{x,y})_{(x,y) \in X^2})$ tel que pour $(x, y) \in X^2$, on ait $max_{x,y} \leq max_x$.

Lemme 2.1. *Soient X un ensemble fini d'horloges et $\lambda = ((max_x)_{x \in X}, (max_{x,y})_{(x,y) \in X^2})$, un ensemble de constantes entières. Alors pour toute région $\alpha = ((\varphi_x)_{x \in X}, (\varphi_{x,y})_{(x,y) \in X^2}, \succ) \in \mathcal{R}_\lambda$, il existe un ensemble fini de régions $\alpha_i = ((\varphi^i_x)_{x \in X}, (\varphi^i_{x,y})_{(x,y) \in X^2}, \succ_i)$ avec $i \geq 0$ telles que $\alpha_0 = \alpha$ et que l'on ait :*

1. $\forall \alpha' \in \mathcal{R}_\lambda,\ \alpha \xrightarrow{time} \alpha' \ \Rightarrow \exists i \geq 0,\ \alpha_i = \alpha'$,
2. $\forall i > 0,\ \forall v \in \alpha,\ \exists t > 0,\ v + t \in \alpha_i$.

Avant de démontrer formellement ce lemme, nous allons donner quelques idées intuitives de sa preuve. Tout d'abord, on ordonne les $(\alpha_i)_{i \geq 0}$ de façon chronologique suivant i. On a :

$$\forall i \geq 0,\ \alpha_i \xrightarrow{time} \alpha_{i+1} \tag{1.20}$$

On peut séparer les régions $(\alpha_i)_{i \geq 0}$ en trois catégories :

– **Les régions telles que $\mathbf{X_0}(\alpha_\mathbf{i}) \neq \emptyset$:**
Pour ce type de région, il existe au moins une horloge $x \in X$ telle que $\varphi_x = (v(x) = c)$. Dans ce cas, le temps est fixé par la valeur des horloges de $X_0(\alpha_i)$ et ne peut s'écouler sans que l'on sorte de la région. Par conséquent, si on laisse s'écouler un intervalle de temps, aussi petit soit-il, on aboutit dans une région qui substitue tous les $\varphi_x = (v(x) = c)$ par des $\varphi_x = (c < v(x) < c + 1)$. Le préordre, lui, est enrichi par toutes les horloges qui vérifiaient $\varphi_x = (v(x) = c)$ en tant qu'éléments minimaux.

– **Les régions telles que $\mathbf{X_0}(\alpha_\mathbf{i}) = \emptyset$ et $\mathbf{X_{frac}}(\alpha_\mathbf{i}) \neq \emptyset$:**
Dans ce type de région, le temps peut s'écouler car aucune horloge $x \in X$ n'est telle que $\varphi_x = (v(x) = c)$ et il existe une horloge $x \in X$ telle que $\varphi_x = (c < v(x) < c+1)$. On détermine la région suivante en jouant sur le fait que ce sont les horloges qui correspondent aux éléments maximaux du préordre qui atteindront les premières une valeur entière. La nouvelle région s'obtient en substituant pour toutes les horloges maximales du préordre les $\varphi_x = (c < v(x) < c+1)$ par des $\varphi_x = (v(x) = c+1)$ et en retirant ces horloges du préordre.
– **Les régions telles que $\mathbf{X_0}(\alpha_\mathbf{i}) = \emptyset$ et $\mathbf{X_{frac}}(\alpha_\mathbf{i}) = \emptyset$:**
Cette condition est la condition d'arrêt de l'algorithme qui construit les α_i. Dans ce cas, toutes les horloges $x \in X$ vérifient $\varphi_x = (v(x) > max_x)$. Comme, par définition, toutes les constantes c sont plus petites que max_x, la valeur de vérité de toutes contraintes $x \sim c$ ne changera plus. Par conséquent, la région qui capture ces comportements n'est pas bornée et complète ainsi le futur de α.

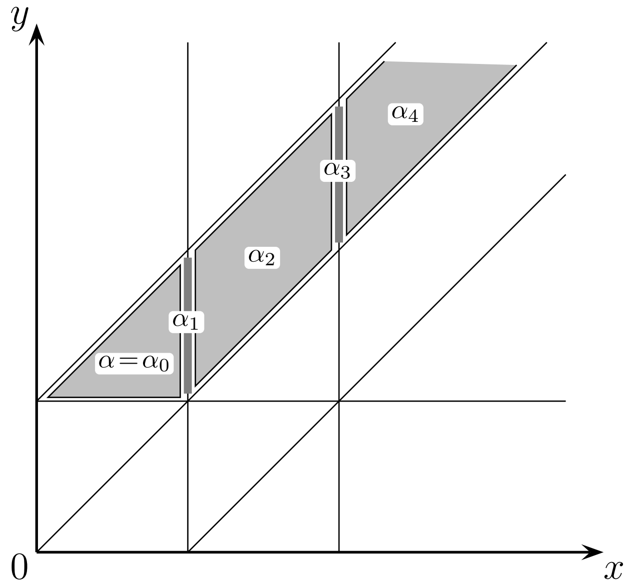

FIG. 1.4 – *Un exemple de régions $(\alpha_i)_{i \geq 0}$ reliées entre elles par la relation $\xrightarrow{time}$.*

Exemple 7. *Afin d'illustrer les différents types de régions, nous détaillons ici chacun des types des régions qui se trouvent sur la figure 1.4 :*

– $X_0(\alpha_0) \neq \emptyset$ *et* $X_{frac}(\alpha_0) \neq \emptyset$,
– $X_0(\alpha_1) = \emptyset$ *et* $X_{frac}(\alpha_1) \neq \emptyset$,
– $X_0(\alpha_2) \neq \emptyset$ *et* $X_{frac}(\alpha_2) \neq \emptyset$,
– $X_0(\alpha_3) = \emptyset$ *et* $X_{frac}(\alpha_3) \neq \emptyset$,
– $X_0(\alpha_4) = \emptyset$ *et* $X_{frac}(\alpha_4) = \emptyset$.

Démonstration. La preuve de ce lemme consiste à caractériser de façon inductive l'ensemble des $(\alpha_i)_{i \geq 0}$ à partir de $\alpha = ((\varphi_x)_{x \in X}, (\varphi_{x,y})_{(x,y) \in X^2}, \succ)$ puis à prouver les propriétés voulues sur cette caractérisation.

Plus précisément, on pose $\alpha_0 = \alpha$ et on construit α_{i+1} à partir de α_i de la façon suivante :

Soit $\alpha_i = ((\varphi_x^i)_{x\in X}, (\varphi_{x,y}^i)_{(x,y)\in X^2}, \succ_i)$, la région $\alpha_{i+1} = ((\varphi_x^{i+1})_{x\in X}, (\varphi_{x,y}^{i+1})_{(x,y)\in Y^2}, \succ_{i+1})$, est définie par :

- Si $X_0(\alpha_i) \neq \emptyset$, alors :
  - $\varphi_x^{i+1} = \begin{cases} \varphi_x^i, & \text{si } x \notin X_0(\alpha_i) \\ (c < v_{i+1}(x) < c+1), & \text{si } \varphi_x^i = (v_i(x) = c) \text{ et } c < max_x \\ (v_{i+1}(x) > max_x), & \text{si } \varphi_x^i = (v_i(x) = max_x) \end{cases}$
  - On ajoute les horloges contenues dans $X_0(\alpha_i)$ en tant qu'éléments minimaux du préordre $\succ_i$. Formellement, on a :

$$\succ_{i+1} = \succ_i \cup \{y \succ_i x \mid x \in X_0(\alpha_i) \text{ et } y \in X_{frac}(\alpha_{i+1})\} \tag{1.21}$$

- Si $X_0(\alpha_i) = \emptyset$ et $X_{frac}(\alpha_i) \neq \emptyset$, alors, on pose $X_{max}(\alpha_i)$ l'ensemble des éléments maximaux pour α_i de $X_{frac}(\alpha_i)$. Plus formellement, on a :

$$X_{max}(\alpha_i) = \{x \in X_{frac}(\alpha_i) \mid \forall y \in X_{frac}(\alpha_i),\ y \not\succ_i x\} \tag{1.22}$$

  Si on note $\lfloor t \rfloor$, la partie entière inférieure de $t \in \mathbb{R}$, on a alors :
  - $\varphi_x^{i+1} = \begin{cases} \varphi_x^i, & \text{si } x \notin X_{max}(\alpha_i) \\ (v_{i+1}(x) = \lfloor v_i(x) \rfloor + 1), & \text{si } x \in X_{max}(\alpha_i) \end{cases}$
  - On retire du préordre toutes les horloges contenues dans $X_{max}(\alpha_i)$. Formellement, on a :

$$\succ_{i+1} = \succ_i \setminus \{x \succ_i y \mid x \in X_{max}(\alpha_i) \text{ et } y \in X\} \tag{1.23}$$

- Si $X_0(\alpha_i) = \emptyset$ et $X_{frac}(\alpha_i) = \emptyset$, alors $\alpha_i = \alpha_{i+1}$.

Maintenant que nous avons défini α_{i+1} en fonction de α_i, et comme l'ensemble $\mathcal{R}_\lambda$ est fini, on peut construire une suite stationnaire $(\alpha_i)_{0\leq i\leq n}$. Il est maintenant immédiat de vérifier que :

1. Par construction, on a :
   - $\alpha_0 = \alpha$,
   - $\forall i \geq 0,\ \alpha_i \xrightarrow{time} \alpha_{i+1}$,
   - $\forall \alpha' \in \mathcal{R}_\lambda,\ \alpha_i \xrightarrow{time} \alpha' \Rightarrow \alpha' = \alpha_i$ ou $\alpha' \xrightarrow{time} \alpha_{i+1}$.

   On peut, donc, en déduire que : $\forall \alpha' \in \mathcal{R}_\lambda,\ \alpha \to \alpha' \Rightarrow \exists i \geq 0,\ \alpha_i = \alpha'$
2. De plus, toujours par construction, on a :
   - $\alpha_0 = \alpha$,
   - $\forall v \in \alpha_i,\ \exists t > 0,\ v + t \in \alpha_{i+1}$.

   On peut, donc, en déduire par induction que : $\forall i \geq 0,\ \forall v \in \alpha,\ \exists t \geq 0,\ v + t \in \alpha_i$

Ce qui permet de conclure que la propriété $(\dagger_1)$ est vraie. □

Nous allons nous intéresser maintenant à la propriété $(\dagger_2)$. C'est dans cette seconde partie de la preuve que nous aurons besoin de l'hypothèse qui assure que pour tout couple $(x, y) \in X^2$, on a $max_{x,y} \leq max_x$.

Lemme 2.2. *Soient X un ensemble fini d'horloges et $\lambda = ((max_x)_{x\in X}, (max_{x,y})_{(x,y)\in Y^2})$, tels que pour tout couple $(x, y) \in X^2$, on ait $max_{x,y} \leq max_x$.*

Alors pour toute région $\alpha = ((\varphi_x)_{x\in X}, (\varphi_{x,y})_{(x,y)\in X^2}, \succ)$ de $\mathcal{R}_\lambda$, il existe un ensemble fini de régions $\alpha_Y = ((\varphi^Y_x)_{x\in X}, (\varphi^Y_{x,y})_{(x,y)\in X^2}, \succ_Y)$ telles que :

1. $\forall \alpha' \in \mathcal{R}_\lambda,\ \alpha \xrightarrow{reset} \alpha' \Rightarrow \exists Y \subseteq X,\ \alpha' = \alpha_Y$,
2. $\forall Y \subseteq X,\ \forall v \in \alpha,\ \exists v_Y \in \alpha_Y,\ v_Y = v[x \leftarrow 0/x \in Y]$.

Démonstration. La preuve de ce lemme consiste à caractériser les ensembles α_Y pour un $Y \subseteq X$ donné. On prouve ensuite les deux propriétés.

Soit $Y \subseteq X$. On définit $\alpha_Y = ((\varphi^Y_x)_{x\in X}, (\varphi^Y_{x,y})_{(x,y)\in X^2}, \succ_Y)$ par :

- $\succ' = \succ \setminus \{(x, y) \in X^2 \mid (x \in Y) \vee (y \in Y)\}$,
- $\varphi'_x = \begin{cases} \varphi_x, & \text{si } x \notin Y \\ (v'(x) = 0), & \text{sinon} \end{cases}$
- $\varphi'_{x,y}$ est tel que :
  - Si $x, y \notin Y$, alors $\varphi'_{x,y} = \varphi_{x,y}$,
  - Si $x, y \in Y$, alors $\varphi'_{x,y} = (v'(x) - v'(y) = 0)$,
  - Si $x \in Y$ et $y \notin Y$, alors :
    - Soit $\varphi_y = (v(y) = c)$ et on a : $\varphi'_{x,y} = (v'(x) - v'(y) = -c)$,
    - Soit $\varphi_y = (c < v(y) < c + 1)$ et on a $\varphi'_{x,y} = (-(c + 1) < v'(x) - v'(y) < -c)$,
    - Soit $\varphi_y = (max_y < v(y))$ et comme par hypothèse on a $max_{y,x} \leq max_y$, on en déduit que $-v(y) < -max_y \leq -max_{y,x}$ et donc que $0 - v(y) < -max_{y,x}$. Dans ce cas, on peut conclure que $\varphi'_{x,y} = (v'(x) - v'(y) < -max_{y,x})$.
  - Si $x \notin Y$ et $y \in Y$, alors, de façon symétrique, on a :
    - Soit $\varphi_x = (v(x) = c)$ et on a : $\varphi'_{x,y} = (v'(x) - v'(y) = c)$,
    - Soit $\varphi_x = (c < v(x) < c + 1)$ et on a $\varphi'_{x,y} = (c < v'(x) - v'(y) < c + 1)$,
    - Soit $\varphi_x = (max_x < v(x))$ et comme par hypothèse on a $max_{x,y} \leq max_x$, on en déduit que $max_{x,y} \leq max_x < v(x)$ et donc que $max_{x,y} < v(x) - 0$. Dans ce cas, on peut conclure que $\varphi'_{x,y} = (max_{x,y} < v'(x) - v'(y))$.

Ainsi, par construction, les $(\alpha_Y)_{Y\subseteq X}$ ainsi définis vérifient bien :

1. $\forall \alpha' \in \mathcal{R}_\lambda,\ \alpha \xrightarrow{reset} \alpha' \Rightarrow \exists Y \subseteq X,\ \alpha' = \alpha_Y$,
2. $\forall Y \subseteq X,\ \forall v \in \alpha,\ \exists v_Y \in \alpha_Y,\ v_Y = v[x \leftarrow 0/x \in Y]$.

□

À partir des lemmes 2.1 et 2.2 et des constructions présentées dans les démonstrations, on définit la relation :

$$\rightarrow = \xrightarrow{time} \cup \xrightarrow{reset} \tag{1.24}$$

Pour finir, $\mathcal{G}_\lambda = (\mathcal{R}_\lambda, \rightarrow)$ ainsi défini vérifie bien le théorème 2 grâce aux lemmes 2.1 et 2.2 que nous avons établis.

1.6.3 L'automate des régions

L'automate des régions est un automate de Büchi classique $\mathcal{A}_R$ qui accepte le langage $Untimed(L(\mathcal{A}))$. Vérifier le vide sur $\mathcal{A}_R$ est clairement équivalent à vérifier le vide sur $\mathcal{A}$. Sommairement, $\mathcal{A}_R$ est obtenu en faisant le produit cartésien de $\mathcal{A}$ et de $\mathcal{G}_\lambda$, en effaçant les contraintes et les remises à zéro.

Définition 5. *Soient $\mathcal{A} = (\Sigma, Q, T, I, F, R, X)$ un automate temporisé, $C \subseteq \mathcal{C}$ l'ensemble des contraintes sur les horloges de $\mathcal{A}$ et $\lambda = ((max_x)_{x \in X}, (max_{x,y})_{(x,y) \in X^2})$ un ensemble des constantes entières tel que $\mathcal{R}_\lambda$ soit compatible avec C. Enfin, soit $\mathcal{G}_\lambda = (\mathcal{R}_\lambda, \rightarrow)$ le graphe des régions associé à λ.*
*Alors l'*automate des régions $\mathcal{A}_R = (\Sigma', Q', T', I', R')$ *est l'automate de Büchi tel que :*

- $\Sigma' = \Sigma$,
- $Q' = Q \times \mathcal{R}_\lambda$,
- $I' = I \times [v_0]$ *avec* $v_0(x) = 0$ *pour tout* $x \in X$,
- $R' = R \times \mathcal{R}_\lambda$,
- $T' \subseteq \langle Q \times \mathcal{R}_\lambda \rangle \times \Sigma \times \langle Q \times \mathcal{R}_\lambda \rangle$ *avec* $(\langle q, \alpha \rangle, \sigma, \langle q', \alpha' \rangle) \in T'$ *si et seulement si on a l'un des deux cas suivants :*
  1. $\exists (q, g, \sigma, r, q') \in T$ *telle que :*
     - $\alpha \subseteq g$,
     - $\exists \xrightarrow{reset} \in \rightarrow$, $\alpha \xrightarrow{reset} \alpha'$ *avec* $\alpha' = \alpha[r \leftarrow 0]$.
  2. $\exists \xrightarrow{time} \in \rightarrow$, $\alpha \xrightarrow{time} \alpha'$ *pour tout* $q = q'$ *et avec* $\sigma = \varepsilon$.

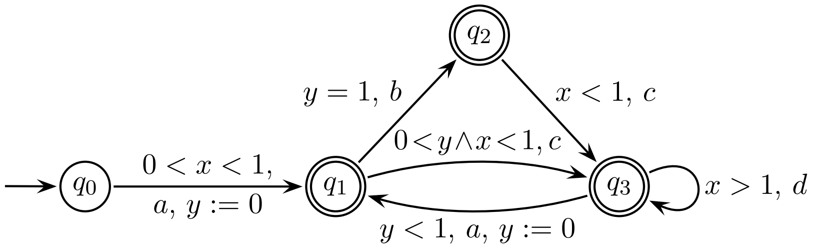

FIG. 1.5 – *Automate temporisé $\mathcal{A}_0$.*

Exemple 8. *Pour illustrer l'automate des régions, nous reprenons l'exemple de [AD94][1]. Soit $\mathcal{A}_0$ l'automate temporisé représenté figure 1.5. L'automate des régions qui correspond à $\mathcal{A}_0$ est représenté sur la figure 1.6 page suivante.*

L'état initial est $\langle q_0, (x = y = 0) \rangle$. Les états répétés sont indiqués par une double ligne autour de l'état.

L'automate des régions $\mathcal{A}_R$ une fois défini, en utilisant $\dagger_1$ et $\dagger_2$, on peut montrer le résultat suivant dû à Alur et Dill :

[1] Avec quelques corrections.

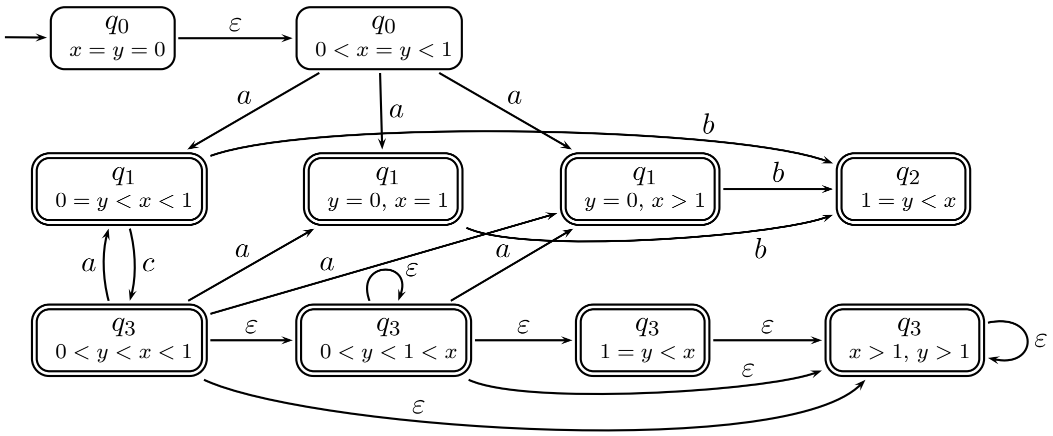

FIG. 1.6 – *Automate des régions associé à $\mathcal{A}_0$.*

Théorème 3 ([AD94]). *Soient un automate temporisé $\mathcal{A} = (\Sigma, Q, T, I, F, R, X)$ et son automate des régions $\mathcal{A}_R = (\Sigma', Q', T', I', R')$. Alors, $\mathcal{A}_R$ accepte le langage Untimed$(L(A))$.*

Démonstration. Nous allons montrer que pour toute exécution de $\mathcal{A}$:

$$\langle q_0, v_0\rangle \xrightarrow[t_1]{g_1,\sigma_1,R_1} \langle q_1, v_1\rangle \xrightarrow[t_2]{g_2,\sigma_2,R_2} \langle q_2, v_2\rangle \ldots$$

Il existe une exécution de $\mathcal{A}_R$:

$$\langle q_0, \alpha_0\rangle \xrightarrow{\varepsilon}^{*} \xrightarrow{\sigma_1} \langle q_1, \alpha_1\rangle \xrightarrow{\varepsilon}^{*} \xrightarrow{\sigma_2} \langle q_2, \alpha_2\rangle \ldots$$

Telle que les régions α_i contiennent les valuations v_i. Puis nous montrerons la réciproque.

1. Soit une exécution de $\mathcal{A}$:

$$\langle q_0, v_0\rangle \xrightarrow[t_1]{g_1,\sigma_1,R_1} \langle q_1, v_1\rangle \xrightarrow[t_2]{g_2,\sigma_2,R_2} \langle q_2, v_2\rangle \ldots$$

On suppose que l'on a déjà construit les i premières étapes de l'exécution de $\mathcal{A}_R$:

$$\langle q_0, \alpha_0\rangle \xrightarrow{\varepsilon}^{*} \xrightarrow{\sigma_1} \langle q_1, \alpha_1\rangle \xrightarrow{\varepsilon}^{*} \xrightarrow{\sigma_2} \langle q_2, \alpha_2\rangle \ldots \xrightarrow{\varepsilon}^{*} \xrightarrow{\sigma_{i-1}} \langle q_{i-1}, \alpha_{i-1}\rangle$$

Tel que pour tout $0 \leq j < i$, on ait $v_j \in \alpha_j$.

Soit la transition $(q_{i-1}, g_i, \sigma_i, R_i, q_i)$ de $\mathcal{A}$ et un temps $t \in \mathbb{T}$ tel que $(v_{i-1} + t) \models g_i$ et $v_i = (v_{i-1} + t)[0 \to x \mid x \in R_i]$. Alors, d'après la propriété ($\dagger_1$) (p. 41), on peut, à partir de l'état $\langle q_{i-1}, \alpha_{i-1}\rangle$ de $\mathcal{A}_R$, atteindre l'état $\langle q_{i-1}, \alpha'_{i-1}\rangle$ avec $(v_{i-1} + t) \in \alpha'_{i-1}$ en prenant une succession de transitions temporelles ($\xrightarrow{\varepsilon}^{*}$). De plus, on sait par hypothèse que $(v_{i-1} + t) \models g_i$, donc $\alpha'_{i-1} \models g_i$ car le graphe des régions de $\mathcal{A}_R$ est compatible avec les contraintes de $\mathcal{A}$ et on suppose donc que $\alpha \models g$ ou $\alpha \models \neg g$. Donc, par définition de $\mathcal{A}_R$, on a :

$$\langle q_{i-1}, \alpha_{i-1}\rangle \xrightarrow{\varepsilon}^{*} \langle q_{i-1}, \alpha'_{i-1}\rangle \xrightarrow{\sigma_i} \langle q_i, \alpha_i\rangle = \langle q_{i-1}, \alpha_{i-1}\rangle \xrightarrow{\varepsilon}^{*} \xrightarrow{\sigma_i} \langle q_i, \alpha_i\rangle$$

Ce qui achève la première partie de la preuve.

2. Considérons à présent une exécution de $\mathcal{A}_R$:

$$\langle q_0, \alpha_0 \rangle \xrightarrow{\varepsilon}^* \xrightarrow{\sigma_1} \langle q_1, \alpha_1 \rangle \xrightarrow{\varepsilon}^* \xrightarrow{\sigma_2} \langle q_2, \alpha_2 \rangle \ldots$$

   On suppose que l'on a déjà construit les i premières étapes de l'exécution de $\mathcal{A}$:

$$\langle q_0, v_0 \rangle \xrightarrow[t_1]{g_1, \sigma_1, R_1} \langle q_1, v_1 \rangle \xrightarrow[t_2]{g_2, \sigma_2, R_2} \langle q_2, v_2 \rangle \ldots \xrightarrow[t_{i-1}]{g_{i-1}, \sigma_{i-1}, R_1} \langle q_{i-1}, v_{i-1} \rangle$$

   Tel que pour tout $0 \leq j < i$, on ait $v_j \in \alpha_j$.

   On suppose que l'exécution de $\mathcal{A}_R$ contient : $\langle q_{i-1}, \alpha_{i-1} \rangle \xrightarrow{\varepsilon}^* \langle q_{i-1}, \alpha'_{i-1} \rangle \xrightarrow{\sigma_i} \langle q_i, \alpha_i \rangle$. Nous avons donc la relation $\alpha_{i-1} \xrightarrow{time} \alpha'_{i-1}$. Or, par construction de $\mathcal{A}_R$, on en déduit qu'il existe dans l'automate temporisé $\mathcal{A}$ une transition $(q_{i-1}, g_i.\sigma_i, R_i, q_i)$ telle que $\alpha'_{i-1} \models g_i$ et $\alpha'_{i-1} \xrightarrow{reset} \alpha_i$.

   Par $(\dagger_1)$, on peut en conclure aussi qu'il existe un $t \in \mathbb{T}$ tel que $(v_{i-1} + t) \in \alpha'_{i-1}$ qui vérifie $(v_{i-1} + t) \models g_i$.
   Enfin, grâce à $(\dagger_2)$, on en déduit qu'il existe $v_i = (v_{i-1} + t)[0 \leftarrow x \mid x \in R_i]$ avec $v_i \in \alpha_i$. Ce qui conclu la deuxième et dernière partie de cette preuve.

□

Comme le problème du vide sur un automate fini (ou de Büchi) est décidable [HU79] et comme, de plus, le vide de $L(A)$ est équivalent à celui de $Untimed(L(A))$, on en déduit le corollaire suivant :

Corollaire 3.1. *Le problème du vide est décidable pour les automates temporisés classiques.*

Alur et Dill ont aussi établi la complexité du problème du vide sur un automate temporisé :

Théorème 4 ([AD94]). *Le problème de décider du vide sur un automate temporisé est* Pspace-*complet.*

Idée de la preuve. Voici une intuition de la preuve de la *Pspace*-complétude du problème du vide pour les automates temporisés.

- **Pspace-easy** : Soit une entrée de taille n qui représente un automate temporisé $\mathcal{A}$. Comme l'automate des régions est de taille exponentielle en nombre d'horloges, on ne peut écrire la représentation de $\mathcal{A}_R$ sur un ruban sans perdre l'aspect PSPACE. On construit donc, à la volée, les états de $\mathcal{A}_R$ dont on a besoin. Mais la représentation des régions sur le ruban est de taille polynômiale en n sur le ruban. On en déduit que le problème est au moins en espace polynômial. Comme la vérification du vide sur un automate de Büchi est en NLOGSPACE [Var96], on en déduit que le problème est en PSPACE.
- **Pspace-hard** : Comme dans [AD94], on réduit le problème du vide sur les automates temporisés avec mises à jour au problème de l'arrêt sur une machine de Turing dont la tête de lecture ne peut dépasser le marqueur de fin du mot d'entrée. Ce problème étant Pspace-complet, on en déduit la Pspace-complétude du problème du vide.

□

Conclusion

Nous avons présenté dans ce chapitre les automates temporisés 'classiques' qui peuvent être vu comme une extension des automates temporisés originaux d'Alur et Dill [AD90, AD94] auxquels on a ajouté des gardes diagonales. Le modèle des automates temporisés 'classiques' est le modèle utilisé dans les logiciels de vérification Kronos [DOTY96, BDM+98], CMC [LL98, CL00a] et Uppaal [BLL+96, LPY97, PL00]. Nous avons définis le cadre de ce modèle, puis nous avons introduit la notion d'automate temporisé. Nous avons ensuite fait un rapide tour d'horizon de la complexité de différents problèmes dans ce modèle. Enfin, nous avons exhibé une preuve détaillée de la décidabilité du problème du vide.

Dans le chapitre suivant nous allons proposer une extension de ce modèle qui permet une plus grande variété de mises à jour. Comme nous le verrons dans le reste de ce document, les propriétés de cette extension sont des plus intéressantes.

Chapitre 2

Les automates temporisés avec mises à jour

There is a theory which states that if ever anyone discovers exactly what the Universe is for and why it is here, it will instantly disappear and be replaced by something even more bizarre and inexplicable.

There is another which states that this has already happened.

— Douglas Adams, *The Hitchhiker's Guide To The Galaxy*

Dans le modèle classique des automates temporisés on autorise la remise à zéro d'un sous-ensemble d'horloges. Nous proposons dans ce chapitre un modèle qui élargit cette possibilité en autorisant des *mises à jour* des valeurs des horloges. On peut lors d'une transition donner, par exemple, des valeurs constantes aux horloges, ou leur attribuer la valeur d'une autre horloge, ou encore leur assigner des valeurs de manière non-déterministe. Nous appellerons ce modèle les *automates temporisés avec mise à jour* (*updatable timed automata*) [BDFP00a, BDFP00b].

2.1 Les mises à jour

Nous étendons le cadre des automates temporisés classiques décrit dans le chapitre 1 en permettant des transformations plus variées sur la valeur des horloges.

Les mises à jour déterministes rassemblent toutes les mises à jours qui affectent à une horloge donnée une constante $x := c$ ou la somme d'une horloge et d'une constante $x := y + c$. Ces affectations sont dites *déterministes* dans le sens où elles déterminent exactement la valeur de l'horloge sans aucune alternative. À l'inverse, les mises à jour non-déterministes associent à une horloge un intervalle d'affectations possibles dont les bornes sont constituées soit d'une constante (c), soit de la somme d'une horloge et d'une constante ($y + c$). Lors d'une mise à jour de ce type, l'horloge considérée peut prendre, de façon *non-déterministe*, n'importe quelle valeur de cet ensemble d'intervalles.

Nous commençons par définir la syntaxe de ces différentes classes de mises à jour, puis, nous définissons plus formellement leur sémantique.

2.1.1 Syntaxe des mises à jour déterministes

Une *mise à jour déterministe* consiste à affecter à une horloge donnée la valeur d'une constante ou la somme d'une horloge et d'une constante. Plus précisément, les mises à jour déterministes sont engendrées par la grammaire suivante :

$$\varphi ::= x := c \,|\, x := y + c, \text{ avec } c \in \mathbb{Q}, \text{ et } x, y \in X \tag{2.1}$$

2.1.2 Syntaxe des mises à jour non-déterministes

Une *mise à jour non-déterministe* est un ensemble de contraintes définissant un intervalle sur lequel l'horloge mise à jour doit prendre sa valeur. Plus précisément, les mises à jour non-déterministes sont engendrées par la grammaire suivante :

$$\varphi ::= x :\sim c \,|\, x :\sim y + c \,|\, \varphi \wedge \varphi \text{ avec } c \in \mathbb{Q},\ x, y \in X \text{ et } \sim \in \{<, \leq, \geq, >\} \tag{2.2}$$

2.1.3 Syntaxe des mises à jour générales

Une *mise à jour générale* φ, est la conjonction de mises à jour déterministes et/ou non-déterministes. Plus précisément, les mises à jour générales sont engendrées par la grammaire suivante :

$$\begin{array}{l} \varphi ::= x :\sim c \,|\, x :\sim y + c \,|\, \varphi \wedge \varphi \\ \qquad\quad \text{avec } c \in \mathbb{Q},\ x, y \in X \text{ et } \sim \in \{<, \leq, =, \geq, >\} \end{array} \tag{2.3}$$

2.1.4 Sémantique des mises à jour

Nous allons à présent introduire la sémantique des mises à jour. D'une manière générale, une *mise à jour* peut être définie comme une fonction $up : \mathbb{T}^X \mapsto \mathcal{P}(\mathbb{T}^X)$ qui assigne à une valuation donnée un ensemble de valuations.

Nous appellerons up_x une *mise à jour locale* à x, une conjonction de mises à jours simples sur x dont la syntaxe est donnée par :

$$\begin{array}{l} up_x ::= x :\sim c \,|\, x :\sim y + c \,|\, up_x \wedge up_x, \\ \qquad\quad \text{avec } c \in \mathbb{Q},\ y \in X \text{ et } \sim \{<, \leq, =, \geq, >\} \end{array} \tag{2.4}$$

On considère une mise à jour locale comme une fonction $[\![up_x]\!] : \mathbb{T}^X \mapsto \mathcal{P}(\mathbb{T})$ dont la sémantique est la suivante :

$$\begin{array}{lcl} [\![\, up_x^1 \wedge up_x^2 \,]\!]\,(v) & = & [\![\, up_x^1 \,]\!] \cap [\![\, up_x^2 \,]\!] \\ [\![\, x :\sim c \,]\!]\,(v) & = & \{w \in \mathbb{T} \mid w \sim c\} \\ [\![\, x :\sim y + c \,]\!]\,(v) & = & \{w \in \mathbb{T} \mid w \sim v(y) + c\} \end{array} \tag{2.5}$$

Remarque 5. *Si v est une valuation, on peut avoir $[\![\, up_x \,]\!]\,(v) = \emptyset$.*

Par abus de langage, nous écrirons par la suite up_x à la place de $[\![\, up_x \,]\!]$.

Soient $Y \subseteq X$ un sous-ensemble d'horloges et $(up_x)_{x \in Y}$ un ensemble de mises à jour locales sur Y. Alors, la *mise à jour up* est la fonction $up : \mathbb{T}^X \mapsto \mathcal{P}(\mathbb{T}^X)$ associée à $(up_x)_{x \in Y}$ dont la sémantique est définie pour toute valuation $v \in \mathbb{T}$, par :

$$up(v) = \left\{ v' \in \mathbb{T}^X \mid \begin{array}{l} v'(x) \in up_x(v), \text{ si } x \in Y \\ v'(x) = v(x), \text{ si } x \notin Y \end{array} \right\} \tag{2.6}$$

La mise à jour est dite *valide* si $up(v) \neq \emptyset$. Nous noterons aussi $\mathcal{U}(X)$ l'ensemble de toutes les mises à jour possibles sur X. Enfin, par extension, si on définit $\mathcal{R}$ un ensemble de régions qui forment une partition de $\mathbb{T}^X$ et α une région, on pose :

$$up(\alpha) = \{\alpha' \in \mathcal{R} \mid \exists v \in \alpha,\ up(v) \cap \alpha' \neq \emptyset\} \tag{2.7}$$

Exemple 9. *Soient* $X = \{x, y, z\}$, $Y = \{x, y\}$, $up_x = (x := 3)$ *et* $up_y = (y :> 2) \wedge (y :< 10)$, *alors on aura :*

$$up(v) = \left\{ v' \in \mathbb{T}^X \;\middle|\; \begin{array}{l} v'(x) = 3 \\ 2 < v'(y) < 10 \\ v'(z) = v(z) \end{array} \right\} \tag{2.8}$$

2.2 Les automates temporisés avec mise à jour

De la même façon que les automates temporisés classiques, un *automate temporisé avec mise à jour* est un 7-uplet $\mathcal{A} = (\Sigma, Q, T, I, F, R, X)$ avec Σ un alphabet fini d'*actions*, Q un ensemble fini d'*états*, X un ensemble fini d'*horloges*, $I \subseteq Q$ l'ensemble des *états initiaux*, $F \subseteq Q$ l'ensemble des *états finaux*, $R \subseteq Q$ l'ensemble des *états répétés*, et un ensemble fini de *transitions* $T \subseteq Q \times [\mathcal{C}(X) \times \Sigma \times \mathcal{U}(X)] \times Q$. Ainsi, une transition est définie par un quintuplet (q, g, a, u, q'), tel que :

- $q, q' \in Q$, deux états,
- $g \in \mathcal{C}(X)$, une *garde*,
- $a \in \Sigma$, une *action*,
- $u \in \mathcal{U}(X)$, une *mise à jour.*

Les notions de *chemins* et d'*exécution* s'étendent sans difficulté aux automates temporisés avec mises à jour.

Un *chemin* (*path*) p dans $\mathcal{A} = (\Sigma, Q, T, I, F, R, X)$ est une suite finie ou infinie de la forme :

$$\begin{gathered} p = q_0 \xrightarrow{g_1, \sigma_1, up_1} q_1 \xrightarrow{g_2, \sigma_2, up_2} q_2 \xrightarrow{g_3, \sigma_3, up_3} q_3 \ldots, \\ \text{avec } q_0 \in I, \text{ et } \forall i > 0,\ (q_{i-1}, g_i, \sigma_i, up_i, q_i) \in T \end{gathered} \tag{2.9}$$

Nous noterons $rep(p) \subset Q$ l'ensemble des états qui sont infiniment souvent répétés sur le chemin p. Le chemin fini (resp. infini) p est *acceptant* si l'état final est dans F (resp. si $rep(p) \cap R \neq \emptyset$, *condition de Büchi*).

Une *exécution* (*run*) r sur le chemin p est une suite de la forme :

$$r = \langle q_0, v_0 \rangle \xrightarrow[t_1]{g_1, \sigma_1, up_1} \langle q_1, v_1 \rangle \xrightarrow[t_2]{g_2, \sigma_2, up_2} \langle q_2, v_2 \rangle \ldots, \text{ avec } \forall i \geq 0,\ v_i \in \mathbb{T}^X \tag{2.10}$$

avec $(t_i)_{i>0}$ une séquence temporisée et $(v_i)_{i\geq 0}$ des valuations d'horloges telles que :

- $v_0(x) = 0$, $\forall x \in X$,
- $\forall i > 0$, $v_{i-1} + (t_i - t_{i-1}) \models g_i$,
- $\forall i > 0$, $v_i \in up_i(v_{i-1} + (t_i - t_{i-1}))$.

Les couples $\langle q_i, v_i \rangle$ sont encore appelés des *états étendus*.

L'étiquette de r est le mot temporisé $(\sigma_1, t_1)(\sigma_2, t_2)\ldots$, qui est dit *accepté* par l'automate temporisé $\mathcal{A}$. L'ensemble des mots qui sont l'étiquette d'une exécution acceptante de $\mathcal{A}$ forment le langage accepté (ou reconnu) par $\mathcal{A}$ et on le note $L(\mathcal{A}, \mathbb{T})$ ou plus simplement $L(\mathcal{A})$.

Exemple 10. *L'automate de la figure 2.1 est un exemple d'automate temporisé avec mises à jour comprenant, entre autres, des mises à jour de type $x := y$, $x := c$ et $x :> c$.*

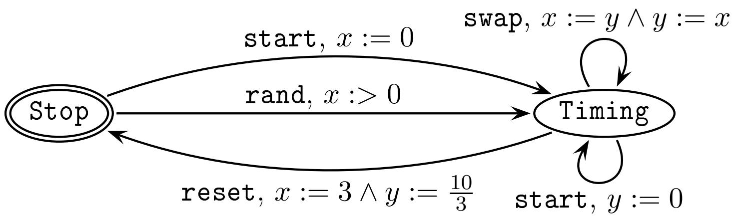

FIG. 2.1 – *Exemple d'automate temporisé avec mise à jour.*

Nous noterons $Aut(C(X), U(X))$ l'ensemble de tous les automates temporisés avec mises à jour construits avec des contraintes sur les horloges contenues dans $C(X)$ et des mises à jour dans $U(X)$. Nous écrirons simplement $Aut(C, U)$ lorsqu'il n'y aura pas d'ambiguïté.

Par la suite, nous aurons besoin de distinguer plusieurs sous-classes d'automates temporisés avec mises à jour. À cet effet, on note

- $\mathcal{C}$: L'ensemble de toutes les contraintes (voir la formule 1.2 page 32),
- $\mathcal{C}_{df}$: L'ensemble des contraintes non diagonales (voir la formule 1.3 page 32),
- $\mathcal{U}$: L'ensemble de toutes les mises à jour (voir la formule 2.3 page 50),
- $\mathcal{U}_{det}$: L'ensemble des mises à jour déterministes (voir équation 2.1 page 50).
- $\mathcal{U}_{det}^{+}$: L'ensemble des mises à jour déterministes avec des constantes sur $\mathbb{Q}_+$ (voir équation 2.1 page 50).
- $\mathcal{U}_{ndet}$: L'ensemble des mises à jour non-déterministes (voir équation 2.2 page 50).
- $\mathcal{U}_{ndet}^{+}$: L'ensemble des mises à jour non-déterministes avec des constantes sur $\mathbb{Q}_+$ (voir équation 2.2 page 50).
- $\mathcal{U}_{local}$: L'ensemble de toutes les mises à jours locales (up_x, avec $x \in X$),
- $\mathcal{U}_0$: L'ensemble des mises à jour du type $x := 0$ (remises à zéro),
- $\mathcal{U}_{cst}$: L'ensemble des mises à jour du type $x := c$.

On peut remarquer que $Aut(\mathcal{C},\mathcal{U}_0)$ correspond à l'ensemble des automates temporisés classiques.

2.3 Conclusion

Le modèle que nous proposons ici, étend les automates temporisés classiques en permettant des mises à jours en lieu et place des remises à zéro habituelles. Ce nouveau modèle soulève deux questions importantes :

– Apporte-t-il un pouvoir expressif plus important que le modèle original ?
– Le problème du vide est-il toujours décidable ? Et si oui, avec quelle complexité ?

Nous pourrions a priori nous intéresser à l'expressivité et à la décidabilité des modèles $Aut(C, U)$ constitués par les classes que nous avons définies ci-dessus. Cependant, il s'avère que le découpage est trop grossier pour permettre d'appréhender correctement ce modèle. Afin de mieux cerner les mécanismes qui régissent la décidabilité ou l'expressivité de ce modèle, nous allons nous intéresser à des classes de mises à jours plus précises présentées sur le tableau 2.1. Chaque case de ce tableau présente un couple *mises à jour* et *contraintes* d'horloges qu'il nous a semblé pertinent de considérer. Il est toutefois à noter que les remises à zéro des horloges sont possibles partout (c'est le cas classique). Les chapitres qui vont suivre utiliseront ce tableau comme un guide dans l'exploration des sous-classes du modèle que nous avons introduit ici.

	Mises à jour	Contraintes non diagonales	Contraintes générales
Déterministes	$x := 0$	PSPACE/TA	PSPACE/TA
	$x := c,\ c \in \mathbb{Q}_+$	?	?
	$x := y + c,\ c \in \mathbb{Q}_+$	?	?
	$x := y - 1$	?	?
	$x := y + c,\ c \in \mathbb{Q}$	?	?
Non-déterministes	$x :< c,\ c \in \mathbb{Q}_+$	?	?
	$x :> c,\ c \in \mathbb{Q}_+$	?	?
	$x :< y$	?	?
	$x :> y$	?	?
	$x :< y + c,\ c \in \mathbb{Q}_+$	?	?
	$x :> y + c,\ c \in \mathbb{Q}_+$	?	?
	$y + c <: x :< y + d,\ c, d \in \mathbb{Q}_+$	?	?
	$y + c <: x :< z + d,\ c, d \in \mathbb{Q}_+,\ y \neq z$	?	?

TAB. 2.1 – *Décidabilité et expressivité des différentes sous-classes du modèle.*

On peut remarquer dès à présent que nous avons fait le choix de séparer les différentes sous-classes suivant deux critères orthogonaux. Le premier critère est de séparer les *contraintes non-diagonales* des *contraintes générales*. Comme nous le verrons par la suite, la présence de contraintes du type $x - y \sim c$ induit d'importants changements dans la décidabilité du modèle. Cela peut paraître surprenant au premier abord lorsqu'on sait que pour la classe particulière des automates temporisés classiques, les contraintes $x - y \sim c$ étaient simulables par des contraintes de type $x \sim c$ [BDGP98]. Le deuxième critère de séparation concerne les mises à jour elles-mêmes. On distingue les contraintes déterministes des contraintes non-déterministes.

Nous pouvons compléter la ligne qui comporte la mise à jour de type ($x := 0$) car elle correspond aux automates temporisés classiques dont nous avons rappelé au chapitre 1 la PSPACE-complétude. Nous noterons TA, l'ensemble des langages qui peuvent être générés par un automate temporisé classique. PSPACE/TA signifie que le problème du vide est PSPACE et que l'expressivité de ce modèle est égale à celle des automates temporisés classiques.

Dans les chapitres suivants, nous commencerons par étudier les aspects d'indécidabilité (chapitre 3) et de décidabilité (chapitre 4) du problème du vide sur le modèle des automates temporisés avec mises à jour. Nous présenterons, ensuite, nos résultats sur l'expressivité de ce modèle comparé aux automates temporisés classiques (chapitre 5).

Chapitre 3

Indécidabilité

A child of five could understand this...
Fetch me a child of five.

— Groucho Marx

Le présent chapitre s'attache à exposer les différents résultats d'indécidabilité que nous avons établis. En premier lieu, le modèle dans son ensemble est indécidable. On affinera cependant ce résultat en exhibant les sous-classes qui en sont à l'origine. Toutes les preuves d'indécidabilité reposent sur des réductions d'une machine de Minsky [Min67] en un automate temporisé avec des mises à jour. Pour conclure ce chapitre, nous compléterons le tableau 2.1 page 53 que nous avons proposé au chapitre 2 et nous discuterons des résultats.

3.1 La machine à deux compteurs

Nous allons rapidement rappeler ce qu'est une machine de Minsky et les propriétés qui sont utiles pour nos démonstrations (indécidabilité du problème du vide).

Définition 6. *Une* machine à deux compteurs *ou* machine de Minsky $\mathcal{M}$ *est un ensemble fini d'instructions* $(p_k)_{0 \leq k \leq f}$ *sur deux compteurs entiers* x *et* y *dont la valeur initiale est* 0. *On distingue une instruction initiale* (p_0) *et une instruction finale* (p_f) *:*

- *Initialisation :* p_0 : `INIT; goto` p_1
- *Arret :* p_f : `HALT`

Et des instructions d'incrémentation et de décrémentation sur les compteurs :

- *Incrémentation* $(+1_i)$ *:* p: `i:=i+1 ; goto q`
- *Décrementation* (-1_i) *:* p:
```
if i>0
  then i:=i-1 ; goto q
  else goto q'
```

Avec `p`, `q` *et* `q'` *des instructions de* $\mathcal{M}$ *et* $i \in \{x, y\}$.

Exemple 11. *Voici un exemple de machine de Minsky.*

p_0 : `INIT`; `goto` p_1
p_1 : $x := x + 1$; `goto` p_3
p_2 : $y := y + 1$; `goto` p_2
p_3 : `if` $y > 0$
  `then` $y := y - 1$; `goto` p_1
  `else goto` p_2
p_4 : `HALT`

`INIT` correspond à la remise à zéro des deux compteurs x et y. Les instructions p_1 et p_2 réalisent chacune une incrémentation, et p_3 un décrémentation. Comme, on le voit facilement, cette machine de Minsky ne termine pas (i.e. elle n'atteint jamais l'état d'arrêt p_4).

Malgré son apparente simplicité, la machine de Minsky possède le même pouvoir d'expression qu'une machine de Turing. Minsky a d'ailleurs prouvé le théorème suivant :

Théorème 5 ([Min67]). *Le problème de l'arrêt est indécidable sur une machine de Minsky.*

L'intérêt de ce modèle est qu'il est extrêmement facile à simuler. Pour cela, il suffit d'exprimer l'opération d'incrémentation et de décrémentation avec test à zéro pour obtenir l'indécidabilité du problème de l'arrêt pour le modèle considéré. C'est donc ce modèle que nous allons simuler avec certaines sous-classes des automates temporisés avec mises à jour.

3.2 Indécidabilité des automates temporisés avec mises à jour

La première évidence est que le modèle des automates temporisés avec mises à jour dans son ensemble est indécidable. Il est en effet aisé de simuler une machine de Minsky avec des mises à jour $x := x + 1$ et $x := x - 1$ et des tests sur les horloges ($x = 0$ et $x > 0$).

Théorème 6. *Soit X un ensemble fini d'horloges contenant au moins 3 éléments. Alors, le problème du vide sur $Aut(\mathcal{C},\mathcal{U})$ est indécidable.*

Démonstration. La preuve revient à simuler une machine de Minsky avec des automates temporisés avec mises à jour. On considère un automate temporisé avec $X = \{x, y, z\}$. Les compteurs de la machine de Minsky sont représentés par les horloges x et y. On simule l'incrémentation en utilisant une mise à jour $x := x + 1$ (figure 3.1) et la décrémentation avec une mise à jour $x := x - 1$ (figure 3.2 page suivante). L'horloge z sert à figer le temps et à ne faire prendre aux compteurs que des valeurs entières. □

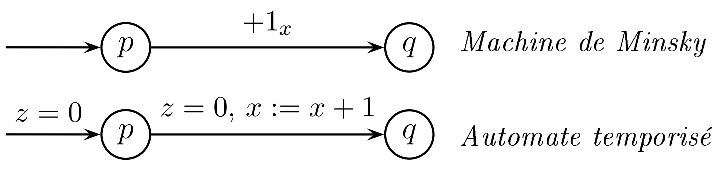

FIG. 3.1 – *Simulation d'une opération "$+1_x$".*

L'indécidabilité de l'ensemble du modèle étant établie, nous allons nous intéresser à certaines sous-classes strictes de $Aut(\mathcal{C},\mathcal{U})$ afin de mieux cerner l'origine de cette indécidabilité et de trouver quelques îlots de décidabilité.

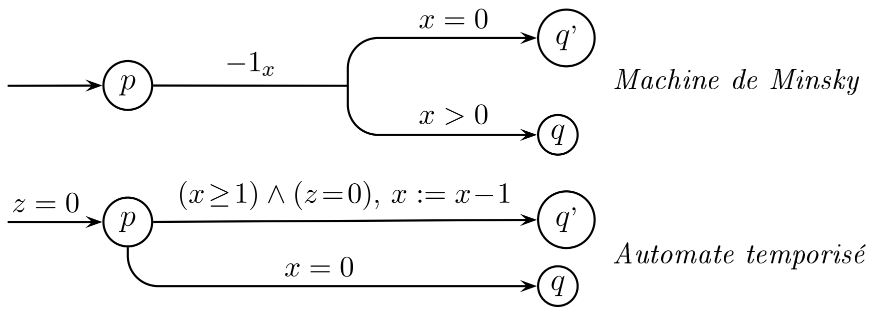

FIG. 3.2 – *Simulation d'une opération "-1_x".*

3.3 Les automates avec mises à jour $x := x - 1$

Nous allons commencer par nous intéresser aux mises à jour de type $x := x - 1$ avec des gardes non diagonales. Pour ce fragment précis, simuler l'opération de décrémentation est direct. L'incrémentation par contre est plus subtile.

Théorème 7. *Soit X un ensemble fini d'horloges tel que $\{x, y, z\} \subseteq X$ et un ensemble de mises à jour $U = \{x{:=}x-1,\ y{:=}y-1,\ z{:=}0\}$. Alors le problème du vide sur les sous-classes $Aut(\mathcal{C}, U)$ et $Aut(\mathcal{C}_{df}, U)$ est indécidable.*

Démonstration. La preuve consiste à simuler une machine de Minsky avec un automate temporisé avec mises à jour qui contiennent des gardes classiques et des mises à jour du type $x := x - 1$. On simule l'opération d'incrémentation ($+1_x$) en laissant écouler une unité de temps et en retranchant 1 à l'autre compteur (figure 3.3). L'opération de décrémentation (-1_x) est aisément simulable par les mises à jour $x := x - 1$ (figure 3.4 page suivante). Enfin, les tests à zéro sont obtenus par de simples gardes $x = 0$. $\square$

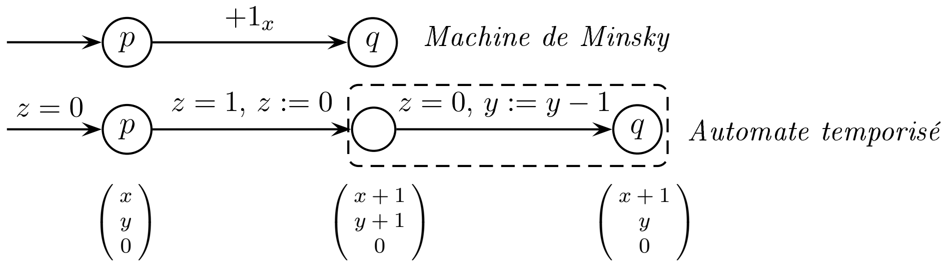

FIG. 3.3 – *Simulation d'une opération "$+1_x$".*

Remarque 6. *Il est intéressant de noter que dans notre réduction, seule une transition (encadrée en pointillé sur les figures 3.3 et 3.4) fait appel à des mises à jour autres que des remises à zéro. On peut donc en conclure que simuler cette transition (figure 3.5) (et non la machine de Minsky complète) suffit à prouver l'indécidabilité. C'est ce que nous allons faire par la suite.*

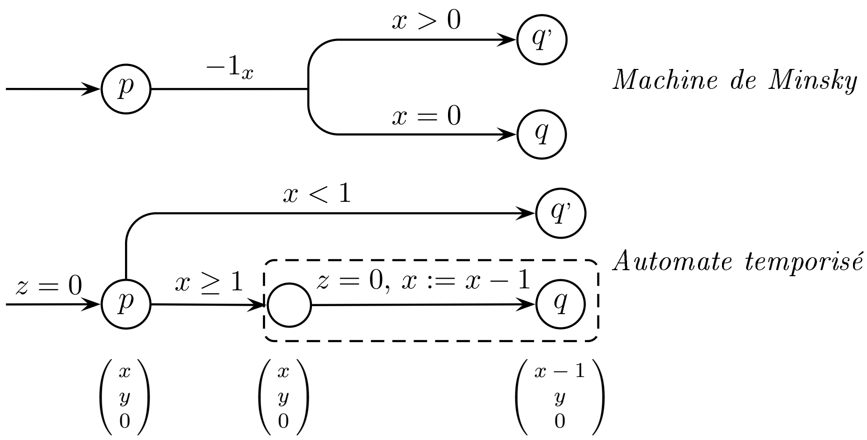

FIG. 3.4 – *Simulation d'une opération "-1_x".*

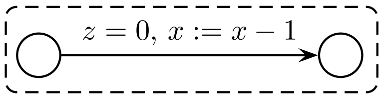

FIG. 3.5 – *Transition utilisant les mises à jour dans la démonstration.*

3.4 Les automates temporisés avec mises à jour $x := x + 1$

Nous abordons ici les mises à jour $x := x + 1$ avec des gardes diagonales. À nouveau pour ce fragment, l'une des deux opérations est aisée. Il s'agit ici de l'incrémentation. La décrémentation telle que nous l'avons codée fait intervenir une horloge supplémentaire qui sert de référence aux deux autres.

Théorème 8. *Soit X un ensemble fini d'horloges tel que $\{w, x, y, z\} \subseteq X$ et un ensemble de mises à jour $U = \{w{:=}0,\ x{:=}0,\ y{:=}0,\ x{:=}x+1,\ y{:=}y+1,\ w{:=}w+1\}$. Alors le problème du vide sur la sous-classe $Aut(\mathcal{C}, U)$ est indécidable.*

Démonstration. On se sert d'une méthode non déterministe pour simuler une décrémentation des compteurs (figure 3.6). Pour ce faire, on utilise une horloge intermédiaire w qui est incrémentée de façon aléatoire en un temps nul (les contraintes sur z arrêtent l'écoulement du temps). Puis on teste le fait que $x - w = 1$ en sortie d'état. Enfin, on incrémente le compteur x de façon aléatoire et, comme précédemment, on teste $x = w$. $\square$

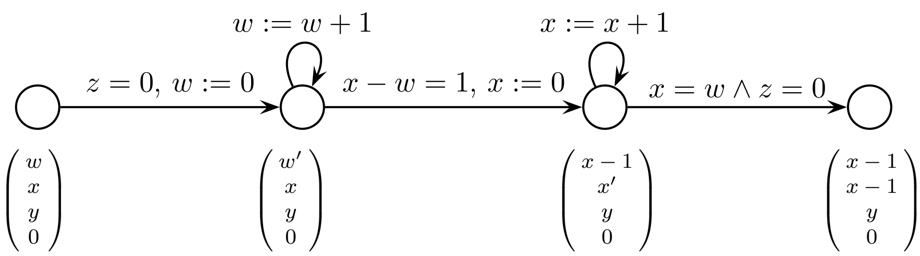

FIG. 3.6 – *Simulation d'une mise à jour "$x := x - 1$" avec "$x := x + 1$".*

Remarque 7. *Il est important de remarquer que l'on utilise des contraintes $x-y=1$. Comme on le verra par la suite, l'absence de ce type de contraintes rend la sous-classe correspondante décidable.*

3.5 Les automates temporisés avec mises à jour $x :> 0$

Nous passons ici aux mises à jours non déterministes avec les mises à jour du type $x :> c$. Ici, nous ne considérons que les mises à jour de type $x :> 0$, mais la preuve peut facilement s'étendre à $x :> c$ par un corollaire assez simple. La technique que nous introduisons ici joue sur la condition d'acceptation du mot temporisé et sur le fait que l'on peut faire un choix non déterministe. Ce méthode de preuve sera reprise pour presque toutes les preuves d'indécidabilité des fragments qui incluent des mises à jour non déterministes.

Théorème 9. *Soit X un ensemble fini d'horloges tel que $\{w,x,y,z\} \subseteq X$ et un ensemble de mises à jour tel que $U = \{x{:>}0,\ y{:>}0,\ w{:>}0\}$. Alors le problème du vide sur la sous-classe $Aut(\mathcal{C},U)$ est indécidable.*

Démonstration. On procède exactement comme pour les $x := x+1$ (incrémentation aléatoire), excepté que l'on remplace les boucles successives par une unique transition avec une mise à jour non déterministe de type $x :> 0$ (figure 3.7). □

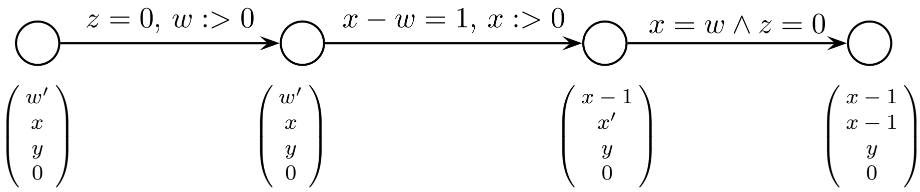

FIG. 3.7 – *Simulation d'une mise à jour "$x := x-1$" avec "$x :> 0$".*

3.6 Les automates temporisés avec mises à jour $x :> y$

Nous nous intéressons ici aux mises à jour de type $x :> y$ avec gardes diagonales. La principale astuce, ici, consiste à utiliser des mises à jour de type $w :> w$ alors que l'on a précédemment remis le contenu de l'horloge w à zéro. L'horloge peut prendre virtuellement n'importe quelle valeur, mais une seule nous intéresse : $x-1$. Dès lors il suffit d'ajouter une garde $w = x-1$ sur la seule transition sortante de l'état suivant la mise à jour $w :> w$.

Théorème 10. *Soit X un ensemble fini d'horloges tel que $\{w,x,y,z\} \subseteq X$ et un ensemble de mises à jour tel que $U = \{w{:=}0,\ x{:=}0\,y{:=}0,\ x{:>}x,\ y{:>}y,\ w{:>}w\}$. Alors le problème du vide sur la sous-classe $Aut(\mathcal{C},U)$ est indécidable.*

Démonstration. Analogue aux deux cas précédents en remplaçant les incrémentations aléatoires par une remise à zéro de l'horloge w, suivie d'une assignation non déterministe (similaire à celle de $x :> 0$). □

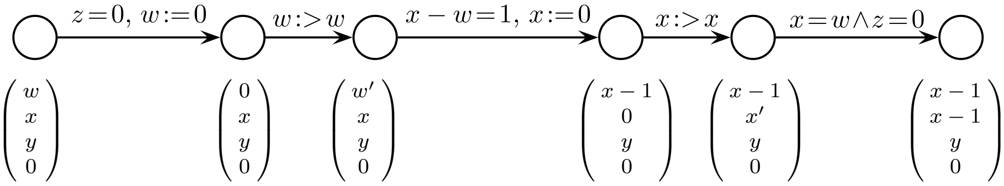

FIG. 3.8 – *Simulation d'une mise à jour "$x := x-1$" avec "$x :> y$".*

3.7 Les automates temporisés avec mises à jour $x :< y$

Nous traitons ici les mises à jour de type $x :< y$ avec des gardes diagonales. Comme nous voulons trouver de façon non déterministe la valeur $x-1$, il suffit de faire prendre à w n'importe quelle valeur inférieure à x puis de vérifier qu'elle est bien égale à $x-1$.

Théorème 11. *Soit X un ensemble fini d'horloges tel que $\{w,x,y,z\} \subseteq X$ et un ensemble de mises à jour tel que $U = \{x :< x,\ y :< y,\ w :< x,\ w :< y\}$. Alors le problème du vide sur la sous-classe $Aut(\mathcal{C},U)$ est indécidable.*

Démonstration. On utilise la mise à jour $w :< x$ pour deviner la valeur de $x-1$. Puis, on teste si la condition est bien réalisée ($x-w=1$) et on fait la même chose avec $x :< x$ en le comparant à w en sortie. □

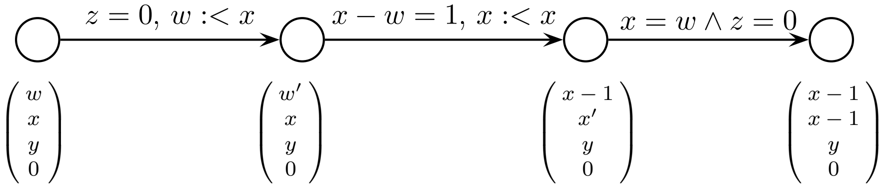

FIG. 3.9 – *Simulation d'une mise à jour "$x := x-1$" avec "$x :< y$".*

3.8 Les automates temporisés avec mises à jour $y+c <: x :< z+d$

On va maintenant montrer que l'ajout d'une mise à jour du type $y+c <: x :< z+d$ permet de simuler les comparaisons d'horloges et rend indécidable des sous classes de $Aut(\mathcal{C}_{df},\mathcal{U})$ de la même manière que $Aut(\mathcal{C},\mathcal{U})$.

Théorème 12. *Soit X un ensemble fini d'horloges tel que $\{x,y,z\} \subseteq X$ et U un ensemble de mises à jour. Si $Aut(\mathcal{C},U)$ est indécidable.*

Alors la classe $Aut(\mathcal{C}_{df}, U \cup \{y+c <: x :< z+d \mid x,y,z \in X,\ c,d \in \mathbb{Q}_+\})$ l'est aussi.

Démonstration. Afin de prouver ce théorème, nous allons montrer que nous pouvons simuler la contrainte diagonale $x-y<0$ par une mise à jour $x <: z :< y$ (figure 3.10). Étendre ce résultat pour des contraintes diagonales $x-y<d$ et des mises à jour $x+c <: z :< y+d$ est ensuite immédiat.

La transition qui contient la mise à jour $x <: z :< y$ ne peut être prise que si l'ensemble des valuations possibles n'est pas vide. Or, si $x-y \geq 0$ cet ensemble est vide. Par conséquent,

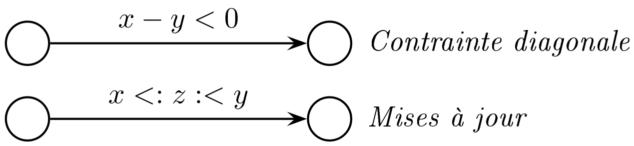

FIG. 3.10 – *Simulation d'une contrainte "$x - y \sim 0$" par "$y {<:} x {:<} z$".*

cette transition ne peut être tirée que si $x - y < 0$. On procède de façon similaire pour $>$, $=$, et $\neq$. On peut donc simuler n'importe quelle contrainte sur une différence d'horloge par une mise à jour de type $y \sim: x :\sim z$.

Le résultat pour la contrainte $x - y < d$ avec des mises à jour de type $x + c <: z :< y + d$ s'obtient de la même façon. $\square$

On peut alors en déduire le corollaire suivant qui établit que considérer des gardes diagonales ou des mises à jour de type $x + c <: z :< y + d$ est équivalent.

Corollaire 12.1. *La décidabilité du problème du vide pour des sous-classes de $Aut(\mathcal{C}_{df}, \mathcal{U})$ qui contiennent des mises à jour du type $y + c <: x :< z + d$ est la même que pour les sous-classes de $Aut(\mathcal{C}, U')$ avec $U' = U \setminus \{y + c <: x :< z + d \mid x, y, z \in X,\ c, d \in \mathbb{Q}_+\}$.*

3.9 Conclusion

Les résultats que nous venons d'établir nous permettent de compléter plus avant le tableau 3.1 que nous avions introduit au chapitre 2. Comme on peut le voir, l'indécidabilité du modèle qui autorise les contraintes diagonales a été largement établie.

	Mises à jours	Contraintes non diagonales	Contraintes générales
Déterministes	$x := 0$	Pspace	Pspace
	$x := c,\ c \in \mathbb{Q}_+$	?	?
	$x := y + c,\ c \in \mathbb{Q}_+$	?	Indécidable
	$x := y - 1$	Indécidable	
	$x := y + c,\ c \in \mathbb{Q}$		
Non déterministes	$x :< c,\ c \in \mathbb{Q}_+$	?	?
	$x :> c,\ c \in \mathbb{Q}_+$	?	Indécidable
	$x :< y$	?	
	$x :> y$	?	
	$x :< y + c,\ c \in \mathbb{Q}_+$	?	
	$x :> y + c,\ c \in \mathbb{Q}_+$	?	
	$y + c <: x :< y + d,\ c, d \in \mathbb{Q}_+$	?	
	$y + c <: x :< z + d,\ c, d \in \mathbb{Q}_+,\ y \neq z$	Indécidable	

TAB. 3.1 – *Décidabilité des différentes sous-classes d'automates temporisés.*

On peut d'ailleurs remarquer dès à présent que les différences d'horloges ont été très employées à travers toutes les preuves d'indécidabilité que nous venons d'établir. Une question pertinente est de se demander si l'absence de ces contraintes rend le problème décidable sur le modèle ainsi défini. Comme nous allons le voir par la suite, c'est effectivement le cas.

Chapitre 4

Décidabilité

As a math atheist, I should be excused from this.
— Calvin & Hobbes

Nous allons à présent nous attacher à prouver la décidabilité du problème du vide pour les sous-classes qui restent encore à préciser sur le tableau 3.1 page 61. Afin de ne pas trop faire durer le suspense, je dirais que nous allons, évidemment, découvrir qu'elles sont décidables, mais sous certaines conditions[1]. Ce sont ces conditions que nous allons fixer dans ce chapitre.

Comme dans le chapitre 1, la preuve de décidabilité repose en grande partie sur la notion de régions d'horloges. Cependant, il nous faut tenir compte de l'introduction des mises à jour au niveau du graphe des régions et de l'automate des régions. Ces résultats ont été publiés dans [BDFP00a].

Nous commencerons par montrer la décidabilité des sous-classes qui n'utilisent que des contraintes non diagonales, puis nous considérerons ensuite la sous-classe des contraintes générales. Enfin, nous établirons la complexité de ces problèmes.

4.1 Mises à jour sans contraintes diagonales

Nous nous intéresserons tout d'abord au fragment des automates temporisés avec mises à jour et sans contraintes diagonales sur les gardes dont la décidabilité n'a pas encore été établie (voir tableau 3.1).

Pour ce faire, nous allons étudier les classes définies par l'ensemble des automates $Aut(\mathcal{C}_{df}, \mathcal{U}_{ndet})$ avec $up \in \mathcal{U}_{ndet}$ défini par la grammaire suivante :

$$up = \bigwedge_{x \in Y} up_x, \text{ avec } Y \subseteq X \text{ et } up_x ::= det_x \,|\, inf_x \,|\, sup_x \,|\, int_x, \text{ avec :}$$

[1]Ne considérer que des constantes c positives dans les mises à jour ($x :\sim c$ et $x :\sim y + c$) est une condition *suffisante* (mais pas *nécessaire*) pour assurer la décidabilité de la classe considérée.

- $det_x ::= x := c \,|\, x := y + d$, avec $c \in \mathbb{N}, d \in \mathbb{Z}$ et $y \in X$
- $inf_x ::= x : \lhd\ c \,|\, x : \lhd\ y + d \,|\, inf_x \wedge inf_x$, avec $\lhd \in \{<, \leq\}, c \in \mathbb{N}, d \in \mathbb{Z}$ et $y \in X$,
- $sup_x ::= x : \rhd\ c \,|\, x : \rhd\ y + d \,|\, sup_x \wedge sup_x$, avec $\rhd \in \{<, \leq\}, c \in \mathbb{N}, d \in \mathbb{Z}$ et $y \in X$,
- $int_x ::= x :\in (c, d) \,|\, x :\in (c, y + d) \,|\, x :\in (y + c, d) \,|\, x :\in (y + c, y + d)$,
  avec ')', '(' $\in$ {']', '['} , $c \in \mathbb{N}, d \in \mathbb{Z}$ et $y \in X$.

Autrement dit, on définit une mise à jour up comme la conjonction de ses mises à jour locales (det_x, inf_x, sup_x, int_x). Et on note une mise à jour générale $up = (up_x)_{x \in Y}$, avec $Y \subseteq X$ et up_x une mise à jour locale à x, pour tout $x \in Y$. Comme noté dans la remarque 4 page 37, nous pouvons, ici aussi, nous restreindre aux seules constantes entières dans les gardes et les mises à jour et étendre le résultat aux constantes rationnelles.

Comme dans la section 1.6 page 37, on va associer à un automate temporisé quelconque de $Aut(\mathcal{C}_{df}, \mathcal{U}_{ndet})$, un ensemble de régions d'horloges $\mathcal{R}$. On définit ensuite, à partir de $\mathcal{R}$, le graphe des régions $\mathcal{G}$ qui va être sensiblement différent du cas classique. Puis par un produit cartésien entre l'automate original et le graphe des régions $\mathcal{G}$, on construit l'automate des régions $\mathcal{A}_R$ qui reconnaît $Untimed(L(A))$. Ce qui nous permet de déduire la décidabilité du problème du vide.

Cependant, comme nous l'avons déjà dit plus haut, le graphe des régions que nous devons construire diffère du cas classique. Essentiellement à cause des mises à jour qui sont plus générales que de simples remises à zéro. Nous allons donc devoir redéfinir l'ensemble des régions d'horloges $\mathcal{R}$ et le graphe des régions $\mathcal{G}$ pour le cas qui nous intéresse.

4.1.1 Les régions d'horloges

Le fait de ne considérer ici que des contraintes non-diagonales sur les gardes va simplifier de façon conséquente la définition des régions. Ainsi, on se donne X un ensemble fini d'horloges et $\lambda = (max_x)_{x \in X}$ un ensemble de constantes entières. On va construire un ensemble de régions $\mathcal{R}_\lambda$ qui forme une partition de $\mathbb{T}^X$. Chacune de ces régions de $\mathcal{R}_\lambda$ étant définie par le produit d'un ensemble d'*intervalles élémentaires* sur les valuations et qui dépendent de λ.

Définition 7. *Soient X un ensemble fini d'horloges et $\lambda = (max_x)_{x \in X}$ un ensemble de constantes entières. Alors, on appelle I_x un* intervalle élémentaire *de x sur λ si on a $I_x \in \mathcal{I}_x$ avec :*

$$\begin{aligned} \mathcal{I}_x = &\{\{c\} \mid 0 \leq c \leq max_x\} \cup \\ &\{\,]c, c+1[\mid 0 \leq c < max_x\} \cup \\ &\{\,]max_x, +\infty[\ \} \end{aligned} \tag{4.1}$$

On se donne aussi X_0, X_{frac} et X_∞ définis comme suit :

Notations 4. *Soient X un ensemble fini d'horloges, $\lambda = (max_x)_{x \in X}$ un ensemble de constantes entières et $(I_x)_{x \in X}$ un ensemble d'intervalles élémentaires. Alors, on pose :*

- $X_0 = \{x \in X \mid I_x = \{c\}\}$ *l'ensemble des horloges dont la partie fractionnaire est* nulle *et dont la valeur est inférieure à max_x,*
- $X_{frac} = \{x \in X \mid I_x =]c, c+1[\}$, *l'ensemble des horloges dont la partie fractionnaire est* non nulle *et dont la valeur est inférieure à max_x.*
- $X_\infty = \{x \in X \mid I_x =]max_x, +\infty[\}$, *l'ensemble des horloges* supérieure à max_x.

On introduit maintenant la notion de *régions*.

Définition 8. *Soient X un ensemble fini d'horloges, $\lambda = (max_x)_{x \in X}$ un ensemble de constantes entières et $(\mathcal{I}_x)_{x \in X}$ un ensemble d'intervalles élémentaires issus de λ. Une* région *est la donnée de :*

- *$(I_x)_{x \in X}$, un ensemble d'intervalles élémentaires de $(\mathcal{I}_x)_{x \in X}$,*
- *$\succ$, un préordre sur les horloges de X_{frac}.*

La région $((I_x)_{x \in X}, \succ)$ représente l'ensemble des valuations $\alpha \subset \mathbb{T}^X$, tel que :

$$\alpha = \left\{ v \in \mathbb{T}^X \;\middle|\; \begin{array}{l} \forall x \in X,\ v(x) \in I_x, \\ \forall x, y \in X_{frac},\ x \succ y \Leftrightarrow frac(v(x)) \geq frac(v(y)) \end{array} \right\} \tag{4.2}$$

Avec $frac(t) = t - \lfloor t \rfloor$ la partie fractionnaire de t. Enfin, $\mathcal{R}_\lambda$ est l'ensemble fini de toutes les régions engendrées par λ. Par commodité, nous noterons $\alpha = ((I_x)_{x \in X}, \succ)$.

Il est à noter que les régions définies ainsi correspondent exactement avec les régions d'horloges telles qu'elles ont été introduites par Alur et Dill à l'origine des automates temporisés [AD90, AD94].

La notion de compatibilité est identique à celle de la définition 4 page 39 :

Définition 9. *Soit X un ensemble fini d'horloges, $C \subseteq \mathcal{C}$ un ensemble de contraintes et $\mathcal{R}$ un ensemble fini de régions sur $\mathbb{T}^X$. Alors, C est* compatible *avec $\mathcal{R}$ si on a :*

$$\forall \psi \in C,\ \forall \alpha \in \mathcal{R},\ (\alpha \models \psi)\ ou\ (\alpha \models \neg\psi) \tag{4.3}$$

Comme dans le chapitre 1, on vérifie par construction qu'un ensemble des contraintes quelconques issues de $\mathcal{C}$ est compatible avec l'ensemble des régions $\mathcal{R}_\lambda$.

Proposition 2. *Soient X un ensemble fini d'horloges, $C \subseteq \mathcal{C}_{df}(X)$ un ensemble fini de contraintes d'horloges et $\lambda = (max_x)_{x \in X}$ un ensemble de constantes entières tel que pour toute horloge x de X, on ait $max_x = \max\{c \mid (x \sim c) \in C\}$. Alors, $\mathcal{R}_\lambda$ est compatible avec C.*

Nous avons donc défini les régions pour des automates sans garde diagonale et nous avons montré que les régions $\mathcal{R}_\lambda$ sont toujours compatibles avec n'importe quel sous-ensemble de $\mathcal{C}_{df}$. Nous allons, à présent, construire le graphe des régions qui correspond aux automates temporisés avec les mises à jour que nous avons défini au début de cette section.

4.1.2 Le graphe des régions

Le graphe des régions va être modifié par la présence de mises à jour plus générales que celles introduites par Alur et Dill, même si le principe reste le même que dans le chapitre 1.

Soit $\mathcal{R}$ un ensemble fini de régions formant une partition de $\mathbb{T}^X$. Le *graphe des régions* $\mathcal{G}$ associé à $\mathcal{R}$ établit les relations de précédence qu'ont les régions entre elles. Il est défini par $\mathcal{G} = (\mathcal{R}, \rightarrow)$ où la fonction de transition $\rightarrow \subset \mathcal{R} \times \mathcal{R}$ est définie à partir des fonctions de transitions suivantes :

- $\xrightarrow{time}$: représente l'écoulement du temps :

$$\forall \alpha, \alpha' \in \mathcal{R},\ \alpha \xrightarrow{time} \alpha' \ \overset{def}{\Longleftrightarrow}\ \exists v \in \alpha,\ \exists t > 0,\ v + t \in \alpha' \tag{4.4}$$

– $\xrightarrow{update}$: représente la mise à jour d'horloges de X :

$$\forall \alpha, \alpha' \in \mathcal{R},\ \alpha \xrightarrow{update} \alpha' \overset{def}{\iff} \exists v \in \alpha,\ \exists up \in \mathcal{U}_{ndet},\ \exists v' \in \alpha', v' \in up(v) \tag{4.5}$$

La fonction de transitions $\rightarrow$ du graphe des régions $\mathcal{G} = (\mathcal{R}, \rightarrow)$ est définie par l'union des transitions *time* et *update* :

$$\rightarrow = \xrightarrow{time} \cup \xrightarrow{update} \tag{4.6}$$

Comme dans le chapitre 1, on contraint le graphe des régions afin de s'assurer que l'automate des régions $\mathcal{A}_R$ capture l'ensemble de tous les comportements de l'automate $\mathcal{A}$. À la fois pour les comportements temporisés et pour les "*mises à jour*". Ces conditions copient les conditions ($\dagger_1$) et ($\dagger_2$) qui sont utilisées pour démontrer la décidabilité du problème du vide dans le théorème 3 page 46. Les conditions ($\ddagger_1$) et ($\ddagger_2$), que nous allons définir ici sont des adaptations des conditions ($\dagger_1$) et ($\dagger_2$) au modèle contenant des mises à jour :

$$\forall \alpha, \alpha' \in \mathcal{R},\ \text{tel que } \alpha \xrightarrow{time} \alpha' \Longrightarrow \forall v \in \alpha, \exists t \geq 0, v + t \in \alpha' \tag{$\ddagger_1$}$$

$$\forall \alpha, \alpha' \in \mathcal{R},\ \text{tel que } \alpha \xrightarrow{update} \alpha' \Longrightarrow \forall v \in \alpha,\ \exists up \in \mathcal{U}_{ndet},\ \exists v' \in \alpha' \cap up(v) \tag{$\ddagger_2$}$$

Ces conditions ne sont pas toujours vérifiées pour un graphe des régions choisi de façon quelconque. Par contre, nous allons montrer que dans le cas particulier d'un graphe des régions de type $\mathcal{G}_\lambda = (\mathcal{R}_\lambda, \rightarrow)$, où $\mathcal{R}_\lambda$ est défini à partir de λ comme précédemment, ($\ddagger_1$) et ($\ddagger_2$) sont vraies si certaines conditions sont vérifiées.

Théorème 13. *Soient X un ensemble fini d'horloges, $C \subseteq \mathcal{C}_{df}(X)$ un ensemble de gardes non diagonales, $U \subseteq \mathcal{U}_{ndet}(X)$ un ensemble fini de mises à jour et $\lambda = (max_x)_{x \in X}$ un ensemble de constantes entières tel que :*

$$\forall (x \sim c) \in C,\ max_x \geq c \tag{4.7}$$
$$\forall (x :\sim c) \in U,\ max_x \geq c \tag{4.8}$$
$$\forall (x :\sim y + c) \in U,\ max_y + c \geq max_x \tag{4.9}$$
$$\forall (x :\in (c, d)) \in U,\ max_x \geq c, d \tag{4.10}$$
$$\forall (x :\in (y + c, d)) \in U,\ max_x \geq d \text{ et } max_y + c \geq max_x \tag{4.11}$$
$$\forall (x :\in (c, y + d)) \in U,\ max_x \geq c \text{ et } max_y + d \geq max_x \tag{4.12}$$
$$\forall (x :\in (y + c, y + d)) \in U,\ max_y + c \geq max_x \text{ et } max_y + d \geq max_x \tag{4.13}$$

Alors, la partition $\mathcal{R}_\lambda$ est compatible avec l'ensemble de contraintes C et le graphe des régions $\mathcal{G}_\lambda = (\mathcal{R}_\lambda, \rightarrow)$ vérifie les conditions ($\ddagger_1$) et ($\ddagger_2$).

Nous allons scinder la preuve de ce théorème en trois parties. Nous montrerons d'abord que $\mathcal{R}_\lambda$ est compatible avec C. Puis nous montrerons successivement que ($\ddagger_1$) puis ($\ddagger_2$) sont vraies pour $\mathcal{G}_\lambda = (\mathcal{R}_\lambda, \rightarrow)$.

La compatibilité de $\mathcal{R}_\lambda$ avec C s'obtient de façon immédiate en utilisant l'hypothèse (4.7) et la proposition 2. Ce qui nous permet de passer à la preuve de la propriété ($\ddagger_1$).

Lemme 13.1. *Soient X un ensemble fini d'horloges, et $\lambda = (max_x)_{x \in X}$ un ensemble de constantes entières. Alors, la condition ($\ddagger_1$) est vraie sur $\mathcal{R}_\lambda$.*

Démonstration. La preuve suit exactement le même schéma que celui du lemme 2.1 page 41. Nous prouvons que pour toute région $\alpha = ((I_x)_{x\in X}, \succ)$ de $\mathcal{R}_\lambda$, il existe un nombre fini de régions de $\mathcal{R}_\lambda$, $\alpha_i = ((I^i_x)_{x\in X}, \succ_i)$ avec $i \geq 0$ telles que $\alpha_0 = \alpha$ et que l'on ait :

1. $\forall \alpha' \in \mathcal{R}_\lambda,$ tel que $\alpha \xrightarrow{time} \alpha' \Rightarrow \exists i \geq 0,$ tel que $\alpha_i = \alpha'$,
2. $\forall i > 0,\ \forall v \in \alpha,\ \exists t > 0,$ tel que $v + t \in \alpha_i$.

Il suffit de ne pas tenir compte des gardes diagonales. Puis, on en déduit la propriété ($\ddagger_1$) à partir de *1.* et *2.* □

La dernière étape de cette preuve est de montrer la propriété ($\ddagger_2$).

Lemme 13.2. *Soient X un ensemble fini d'horloges, $U \subset \mathcal{U}_{ndet}$ un ensemble fini de mises à jour sur X et $\lambda = (max_x)_{x\in X}$ un ensemble de constantes entières telles que :*

$$\forall (x :\sim c) \in U,\ max_x \geq c \tag{4.8}$$
$$\forall (x :\sim y + c) \in U,\ max_y + c \geq max_x \tag{4.9}$$
$$\forall (x :\in (c, d)) \in U,\ max_x \geq c, d \tag{4.10}$$
$$\forall (x :\in (y + c, d)) \in U,\ max_x \geq d \text{ et } max_y + c \geq max_x \tag{4.11}$$
$$\forall (x :\in (c, y + d)) \in U,\ max_x \geq c \text{ et } max_y + d \geq max_x \tag{4.12}$$
$$\forall (x :\in (y + c, y + d)) \in U,\ max_y + c \geq max_x \text{ et } max_y + d \geq max_x \tag{4.13}$$

Alors, la propriété ($\ddagger_2$) est vraie sur $\mathcal{G}_\lambda = (\mathcal{R}_\lambda, \rightarrow)$.

La preuve de ce lemme s'effectue en deux parties. Nous commencerons par prouver la propriété ($\ddagger_2$) pour des *mises à jours simples* que nous définissons plus bas, puis nous étendrons le résultat aux mises à jours $\mathcal{U}_{ndet}$.

Dans cette première partie de la preuve nous présumerons que la mise à jour *up* est une mise à jour simple sur l'horloge $x \in X$. On appelle une *mise à jour simple*, une mise à jour de la forme :

$$up ::= x :\sim c \,|\, x :\sim y + d, \text{ avec } y \in X,\ c \in \mathbb{N}, d \in \mathbb{Z}, \text{ et } \sim \in \{<, \leq, =, \geq, >\} \tag{4.14}$$

Nous allons d'abord caractériser $up(\alpha)$, l'ensemble des régions accessibles à partir de la région α par *up*. Puis, nous montrerons que $up(\alpha)$ est inclus dans un ensemble de régions de $\mathcal{R}_\lambda$. Enfin, nous en déduirons que la propriété ($\ddagger_2$) est vraie.

Par la suite, pour des raisons de concision, nous noterons $\succ_{|Y}$ la restriction de la relation $\succ \subset X \times X$ à $Y \subset X$. Ou plus formellement :

$$\succ_{|Y} = \succ \cap (Y \times Y) \text{ avec } Y \subseteq X \tag{4.15}$$

Nous allons donc prouver la propriété ($\ddagger_2$). Pour ce faire, nous allons caractériser toutes les régions α' qui sont accessibles à partir d'une mise à jour de la région α.

Soient une région $\alpha = ((I_x)_{x\in X}, \succ) \in \mathcal{R}_\lambda$ et une mise à jour simple *up* sur l'horloge $z \in X$. Soit aussi une région $\alpha' = ((I'_x)_{x\in X}, \succ') \in \mathcal{R}_\lambda$ telle que $\alpha' \in up(\alpha)$. Nécessairement, par définition de *up*, on a $I'_x = I_x$ pour $x \neq z$. Reste à définir I'_z et $\succ'$:

Si $\mathbf{up} = (\mathbf{z} :\sim \mathbf{c})$: I'_z peut être n'importe quel intervalle de $\mathcal{I}_z$ qui intersecte l'ensemble $\{v \in \mathbb{T} \mid v \sim c\}$. Les I'_z peuvent prendre l'une des trois formes suivantes :

- Soit $I'_z =]max_z, +\infty[$, dans ce cas on a le préordre $\succ' = \succ_{|X_{frac} \setminus \{z\}}$.
  Nous avons donc définit un $\alpha' \in \mathcal{R}_\lambda$, montrons à présent que si l'on choisit un $v \in \alpha$, quelconque, alors il existe $v' \in \alpha' \cap up(v)$. Par définition de up, $v'(x) = v(x)$ pour toutes les horloges $x \in X \setminus \{z\}$. Il nous suffit de définir un $v'(z)$ tel que $v' \in \alpha' \cap up(v)$. Comme $I'_z =]max_z, +\infty[$, on pose $v'(z) = max_z + 1$. Par hypothèse, on sait que $c \leq max_z$ (4.8), ce qui veut dire que $I'_z \subseteq \{v \in \mathbb{T} \mid v \sim c\}$. On en déduit que $v' \in \alpha' \cap up(v)$ puisque $v'(z) \in I'_z$.

- Soit $I'_z = \{d\}$, dans ce cas on a le préordre $\succ' = \succ_{|X_{frac} \setminus \{z\}}$.
  Nous avons donc définit un $\alpha' \in \mathcal{R}_\lambda$, montrons à présent que si l'on choisit un $v \in \alpha$, quelconque, alors il existe $v' \in \alpha' \cap up(v)$. Par définition de up, $v'(x) = v(x)$ pour toutes les horloges $x \in X \setminus \{z\}$. Il nous suffit de définir un $v'(z)$ tel que $v' \in \alpha' \cap up(v)$. Comme $I'_z = \{d\}$, on pose $v'(z) = d$, et on voit que $v' \in \alpha' \cap up(v)$ puisque $v'(z) \in I'_z$.

- Soit $I'_z =]d, d+1[$, et on a $X'_{frac} = X_{frac} \cup \{z\}$ et $\succ'$ est n'importe quel préordre qui vérifie $\succ'_{|X_{frac} \setminus \{z\}} = \succ_{|X_{frac} \setminus \{z\}}$.
  Nous avons donc définit un $\alpha' \in \mathcal{R}_\lambda$, montrons à présent que si l'on choisit un $v \in \alpha$, quelconque, alors il existe $v' \in \alpha' \cap up(v)$. Par définition de up, $v'(x) = v(x)$ pour toutes les horloges $x \in X \setminus \{z\}$. Il nous suffit de définir un $v'(z)$ tel que $v' \in \alpha' \cap up(v)$. Si on prend, $\succ'$, n'importe quel préordre total sur $X'_{frac} = X_{frac} \cup \{z\}$, alors on pose $\varepsilon > 0$ tel que si on a $x \succ' z$ (resp. $z \succ' x$) avec $x \in X_{frac} \setminus \{z\}$ on ait $frac(v(x)) \geq \varepsilon$ (resp. $\varepsilon \geq frac(v(x))$). Alors, on a $v'(z) = d + \varepsilon$, et on voit que $v' \in \alpha' \cap up(v)$ pour n'importe quel α' de la collection définie précédemment.

Si $\mathbf{up} = (\mathbf{z} :\sim \mathbf{y} + \mathbf{c})$: I'_z peut être n'importe quel intervalle de $\mathcal{I}_z$ qui intersecte l'ensemble $\{v_z \in \mathbb{T} \mid v_z \sim v_y + c \text{ avec } v_y \in I_y\}$. Les I'_z peuvent prendre l'une des trois formes suivantes :

- Soit $I'_z =]max_z, +\infty[$, dans ce cas on a le préordre $\succ' = \succ_{|X_{frac} \setminus \{z\}}$.
  Nous avons donc définit un $\alpha' \in \mathcal{R}_\lambda$, montrons à présent que si l'on choisit un $v \in \alpha$, quelconque, alors il existe $v' \in \alpha' \cap up(v)$. Par définition de up, $v'(x) = v(x)$ pour toutes les horloges $x \in X \setminus \{z\}$. Il nous suffit de définir un $v'(z)$ tel que $v' \in \alpha' \cap up(v)$. Comme $I'_z =]max_z, +\infty[$, on pose $v'(z) = max_y + c + 1$. Par hypothèse, on sait que $max_y + c \leq max_z$ (4.9), ce qui veut dire que $max_y + c + 1 \geq max_z + 1 > max_z$. On en déduit que $v' \in \alpha' \cap up(v)$ puisque $v'(z) \in I'_z$.

- Soit $I'_z = \{d\}$, dans ce cas on a $X'_{frac} = X_{frac} \setminus \{z\}$ et $\succ' = \succ_{|X_{frac} \setminus \{z\}}$.
  Nous avons donc définit un $\alpha' \in \mathcal{R}_\lambda$, montrons à présent que si l'on choisit un $v \in \alpha$, quelconque, alors il existe $v' \in \alpha' \cap up(v)$. Par définition de up, $v'(x) = v(x)$ pour toutes les horloges $x \in X \setminus \{z\}$. Il nous suffit de définir un $v'(z)$ tel que $v' \in \alpha' \cap up(v)$. Comme $I'_z = \{d\}$, on pose $v'(z) = d$, et on voit que $v' \in \alpha' \cap up(v)$ puisque $v'(z) \in I'_z$.

- Soit $I'_z =]d, d+1[$, auquel cas on a $X'_{frac} = X_{frac} \cup \{z\}$ et on a deux cas :
  - Soit $y \notin X_{frac}$, et $\succ'$ est n'importe quel préordre qui vérifie $\succ'_{|X_{frac} \setminus \{z\}} = \succ_{|X_{frac} \setminus \{z\}}$,
  - Soit $y \in X_{frac}$, et alors on a :

- Soit $I_y+c \neq I'_z$, et $\succ'$ est n'importe quel préordre tel que $\succ'_{|X_{frac}\setminus\{z\}} = \succ_{|X_{frac}\setminus\{z\}}$,
- Soit $I_y+c = I'_z$, et $\succ'$ est n'importe quel préordre tel que $\succ'_{|X_{frac}\setminus\{z\}} = \succ_{|X_{frac}\setminus\{z\}}$ et qui vérifie :
  - $z \succ' y$, si $\{\sim\} = \{\geq\}$,
  - $y \succ' z$, si $\{\sim\} = \{\leq\}$,
  - $z \succ' y$ et $y \not\succ' z$, si $\{\sim\} = \{>\}$,
  - $z \not\succ' y$ et $y \succ' z$, si $\{\sim\} = \{<\}$,
  - $z \succ' y$ et $y \succ' z$, si $\{\sim\} = \{=\}$,
  - $(z \succ' y$ et $y \not\succ' z)$ ou $(z \not\succ' y$ et $y \succ' z)$, si $\{\sim\} = \{\neq\}$.

Nous avons donc définit un $\alpha' \in \mathcal{R}_\lambda$, montrons à présent que si l'on choisit un $v \in \alpha$, quelconque, alors il existe $v' \in \alpha' \cap up(v)$. Par définition de up, $v'(x) = v(x)$ pour toutes les horloges $x \in X \setminus \{z\}$. Il nous suffit de définir un $v'(z)$ tel que $v' \in \alpha' \cap up(v)$. On distingue deux cas suivant le préordre $\succ'$ choisi :

- S'il existe une horloge x dont la partie fractionnaire est la même que celle de z (*i.e.* il existe $x \in X \setminus \{z\}$ telle que $x \succ' z$ et $z \succ' x$). Alors, on pose $v'(z) = d + frac(v'(x))$ et on voit que $v' \in \alpha' \cap up(v)$ puisque $v'(z) \in I'_z$.
- Si aucune horloge n'a une partie fractionnaire égale à celle de z (*i.e.* si pour toute horloge $x \in X \setminus \{z\}$ on a $x \not\succ' z$ ou $z \not\succ' x$). Alors, on pose $v'(z) = d + \varepsilon$, avec :

$$\max_{x \in X\setminus\{z\}} \{frac(v'(x)) \mid z \succ' x\} < \varepsilon < \min_{x \in X\setminus\{z\}} \{frac(v'(x)) \mid x \succ' z\}$$

  L'existence d'un tel ε est justifiée par le fait que d'une part, comme aucune horloge n'a une partie fractionnaire égale à celle de z, on peut en déduire que :

$$\max_{x \in X\setminus\{z\}} \{frac(v'(x)) \mid z \succ' x\} < \min_{x \in X\setminus\{z\}} \{frac(v'(x)) \mid x \succ' z\}$$

  D'autre part, comme on se trouve sur un domaine dense, on sait qu'il existe toujours un élément entre deux éléments de ce domaine (par définition). On peut donc en déduire qu'il existe un ensemble infini de tels ε. On en déduit, finalement, que $v' \in \alpha' \cap up(v)$ puisque $v'(z) \in I'_z$.

Nous avons donc montré qu'étant donné deux régions α et α' de $\mathcal{R}$ telles que $\alpha' \cap up(\alpha) \neq \emptyset$, si $v \in \alpha$, alors il existe un $v' \in \alpha' \cap up(v)$.

Il nous reste toutefois à prouver que nous avons bien décris de façon exhaustive toutes les régions de $up(\alpha)$. Afin de s'en convaincre, on peut remarquer que l'on a décrit toutes les régions qui intersectent l'ensemble $\{v_z \in \mathbb{T} \mid v_z \sim c\}$ (*resp.* $\{v_z \in \mathbb{T} \mid v_z \sim v_y + c$ avec $v_y \in I_y\}$) pour une mise à jour du type $(x :\sim c)$ (*resp.* $(x :\sim y + c)$). Si l'on suppose, par l'absurde, qu'il existe une région α', telle que $\alpha' \in up(v)$ et qui ne rencontre pas $\{v_z \in \mathbb{T} \mid v_z \sim c\}$ (*resp.* $\{v_z \in \mathbb{T} \mid v_z \sim v_y + c$ avec $v_y \in I_y\}$), alors les valuations de cette région ne vérifient pas les conditions de la mise à jour. Ce qui nous permet de déduire que la région ne peut faire partie de la mise à jour up.

Ce qui nous permet de conclure que $(\ddagger_2)$ est vraie pour les mises à jour simples.

Remarque 8. *On peut remarquer que les mises à jour de type $z := z - 1$ ne peuvent être utilisées dans le cadre de ce théorème car elles enfreignent la condition $max_x \leq max_y + c$. En effet, dans le cas particulier de cette mise à jour on aurait : $max_z \leq max_z - 1$. Ce qui est impossible.*

Avant de considérer des mises à jour quelconques de $\mathcal{U}_{ndet}$ (voir p. 63), nous devons nous assurer que le résultat que nous venons d'établir pour les mises à jour simples s'étend aux mises à jour locales.

Soit α une région et up_x une mise à jour locale telle qu'elles sont définies au début de cette section. La première étape va être de caractériser les régions $\alpha' \in up_x(\alpha)$. Les caractérisations des mises à jour du type $(x :\sim c)$ et $(x :\sim y + c)$ sont identiques à celles de la preuve précédente. Cependant, le cas des mises à jour du type $(x :\in I)$ diffèrent sensiblement. En effet, nous devons toujours être capable de décider si la mise à jour est valide (*i.e.* savoir si $I \neq \emptyset$) à partir des informations contenues dans α.

- **Si** $up = (x :\in (c, d))$:
  Le test $(I \neq \emptyset)$ est équivalent à $c < d$. Le résultat de ce test est indépendant de α.
- **Si** $up = (x :\in (y + c, d))$:
  Le test $(I \neq \emptyset)$ est équivalent à $y < d - c$. Le résultat de ce test est donnée par le test $[0, d - c[\cap I_y \neq \emptyset$.
- **Si** $up = (x :\in (c, y + d))$:
  Le test $(I \neq \emptyset)$ est équivalent à $y > c - d$. Le résultat de ce test est donnée par le test $]c - d, +\infty[\cap I_y \neq \emptyset$.
- **Si** $up = (x :\in (y + c, y + d))$:
  Le test $(I \neq \emptyset)$ est équivalent à $c < d$. Le résultat de ce test est indépendant de α.

Une fois que nous avons établi que l'intervalle I est différent de l'ensemble vide, caractériser les régions α' et prouver que $\alpha' \in up_x(\alpha)$ suit le même schéma que dans la preuve précédente.

Nous avons donc prouvé que $(\ddagger_2)$ est vraie pour les mises à jour locales en étendant les résultats que nous avions obtenus pour les mises à jour simples.

Remarque 9. *Il est intéressant de remarquer que la classe décidable que nous venons d'exhiber n'autorise pas les mises à jour du type $(x :\in (y, z))$. En effet, le test $I \neq \emptyset$ revient à répondre à la question $(y - z < 0)$. Or, comme nous ne considérons pas les gardes diagonales dans ce modèle, α ne contient pas cette information lorsque soit y, soit z est plus grande que leur constante maximum.*

Il se trouve, par ailleurs, que ce fait est corroboré par le chapitre 3 dans lequel nous avons montré qu'il était possible de coder une machine de Minsky avec des mises à jour du type $(x :\in (y, z))$ (voir p.60).

Nous allons maintenant considérer les mises à jour $up \in \mathcal{U}_{ndet}$ telles que nous les avons définies au début de cette section. Il est facile de voir que toute mise à jour de $\mathcal{U}_{ndet}$ peut être obtenue par la composition de mises à jour locales. Mais, nous pouvons aussi mettre à jour plusieurs horloges en même temps. Il nous faut donc vérifier que cela ne risque pas de remettre en cause la propriété $(\ddagger_2)$.

Soient $\alpha = ((I_x)_{x \in X}, \succ)$ et $\alpha' = ((I'_x)_{x \in X}, \succ')$ deux régions de $\mathcal{R}_\lambda$ et $up \in \mathcal{U}_{ndet}$ une mise à jour de U telle que $\alpha' \in up(\alpha)$(*i.e.* $\exists v \in \alpha$ et $v' \in \alpha'$ telles que $v' \in up(v)$). Alors, pour toutes les horloges $x \in X$, on définit la valuation v_z comme suit :

$$v_z(x) = \begin{cases} v(x), & \text{si } x \neq z \\ v'(z), & \text{si } x = z \end{cases} \tag{4.16}$$

Et soit $\alpha_z = ((I_x^{(z)})_{x \in X}, \succ^{(z)})$ l'unique région de $\mathcal{R}_\lambda$ contenant v_z.

Soit maintenant w une valuation dans α. Comme nous venons de le montrer, la propriété ($\ddagger_2$) est vraie pour un graphe des régions R_λ et des mises à jour locales $(up_x)_{x \in X}$. Alors, on en déduit que pour chaque horloge x, il existe une valuation $w_x \in up_x(w)$. Si de plus on pose $w'(x) = w_x(x)$ pour toute horloge x, on en déduit par définition que $w' \in up(w)$. Il est ensuite facile de déduire que $w' \in R'$, car on a $w'(x) = w_x(x) \in I_x^{(x)} = I'_y$.

Reste à montrer que la suite $(frac(w'(x)))_{x \in X}$ vérifie les conditions données par le pré-ordre $\succ'$. Nous allons procéder en exhibant une méthode de construction de $\succ'$ en fonction de $\succ$ et des pré-ordres $(\succ^{(x)})_{x \in X}$.

Soit X' une copie disjointe de l'ensemble d'horloges X. On définit une suite $(\overline{\succ}^{(x)})_{x \in X}$ de pré-ordres sur l'ensemble d'horloges $X \cup X'$. Intuitivement, $\overline{\succ}^{(x)}$ est obtenu à partir de $\succ^{(x)}$ en remplaçant simplement l'horloge x par sa copie x'. Formellement, on a :

$$\forall y, z \in X \setminus \{x\},\ y \overline{\succ}^{(x)} z, \text{ si } y \succ^{(x)} z$$
$$\forall y \in X \setminus \{x\},\ y \overline{\succ}^{(x)} x', \text{ si } y \succ^{(x)} x$$
$$\forall y, z \in X \setminus \{x\},\ x' \overline{\succ}^{(x)} y, \text{ si } x \succ^{(x)} y$$

On définit $\overline{\succ}$ comme l'union de tous les $\overline{\succ}^{(x)}$ et on vérifie facilement que $\overline{\succ}$ est encore un pré-ordre sur $X \cup X'$. Ensuite, on obtient $\succ'$ à partir de $\overline{\succ}$ en le restreignant à $X' \times X'$, puis en substituant chaque horloge x' par sa copie x. Le pré-ordre $\succ'$ peut donc être calculé à partir des pré-ordres $\succ$ et $(\succ^{(x)})_{x \in X}$. Comme $(frac(w(x)))_{x \in X}$ vérifie le pré-ordre $\succ$ et que chaque pré-ordre $(frac(w_x(y)))_{yx \in X}$ vérifie le pré-ordre $\succ^{(x)}$, on obtient le fait que $(frac(w'(x)))_{x \in X}$ vérifie le pré-ordre $\succ'$. Nous pouvons en conclure que la valuation w' est dans α'

Finalement, nous avons montré que si $\alpha' \in up(\alpha)$, alors pour toute valuation w de α, il existe une valuation w' dans $\alpha' \in up(\alpha)$. Ce qui prouve la propriété $\ddagger_2$.

Remarque 10. *Ainsi que nous venons de le voir dans le cas des mises à jour du type $x :\sim y+c$, la condition $max_x \leq max_y + c$ est suffisante pour garantir les conditions ($\ddagger_1$) et ($\ddagger_2$). Il est par contre difficile de voir en quoi cette hypothèse est utile. Nous allons présenter ici un contre exemple qui permettra de mieux appréhender les mécanismes qui sous-tendent le choix de cette condition.*

Considérons l'ensemble des régions d'horloges $\mathcal{R}_{(2,1)}$ donné par $max_x = 2$ et $max_y = 1$. Nous voulons trouver l'ensemble des régions accessibles par une mise à jour $up = (x :> y)$ à partir de la région $\alpha = (]1, 2[\times]1, +\infty[, \emptyset)$. La figure 4.1 présente l'ensemble des régions $\mathcal{R}_{(2,1)}$.

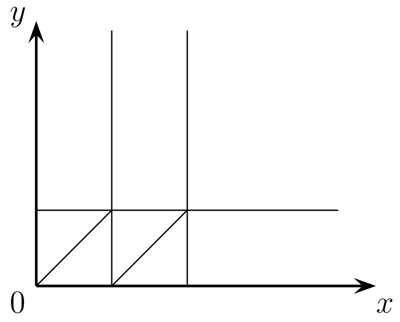

FIG. 4.1 – *Régions engendrées par $max_x = 2$ et $max_y = 1$ ($\mathcal{R}_{(2,1)}$).*

On peut immédiatement remarquer que la mise à jour $x :> y$ sur $\mathcal{R}_{(2,1)}$ ne vérifie pas la condition $max_x \leq max_y + c$ ($2 \leq 1 + 0$).

Comme le montre la figure 4.2, l'image de α par $x :> y$ dépend de la position de la valuation à laquelle on applique la mise à jour. On peut ainsi découper la région α en trois zones (A), (1) et (2). Et on a :

- *L'image de (A) par $x :> y$ est (B),*
- *L'image de (1) par $x :> y$ est (2), (3) ou (4),*
- *L'image de (2) par $x :> y$ est (2), (3) ou (4),*

Cependant, aucune de ces régions ne fait partie de $\mathcal{R}_{(2,1)}$. On ne peut donc pas « décider » de l'image de α par la mise à jour $x :> y$.

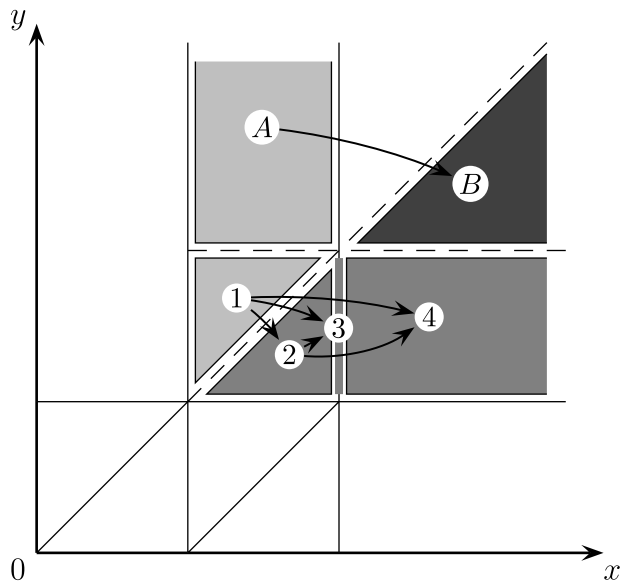

FIG. 4.2 – *Contre exemple pour $x :> y$ et $max_x = 2$ et $max_y = 1$.*

Comme nous venons de la montrer la décidabilité d'un automate temporisé avec mises à jour non-déterministes est liée à l'existence ou non d'une solution à un ensemble d'équations Diophantiennes de la forme $max_x \leq c$ et $max_x \leq max_y + c$. Lorsque l'on considère des constantes c toujours positives, l'existence d'une solution est assurée [Dom91]. Cela implique le fait que si l'on se restreint à des constantes positives, alors le classe des automates temporisés avec mises à jour est toujours décidable pour le problème du vide. À contrario, si l'on considère des constantes éventuellement négatives, l'existence d'une solution au système d'inéquations n'est plus assurée et l'on ne peut trancher sur la décidabilité de cette classe de façon globale. Il faut alors considérer les automates au cas par cas. Finalement, nous pouvons à présent conclure avec le corollaire suivant :

Corollaire 13.1. *Soit $\mathcal{C} \subset \mathcal{C}_{df}(X)$ un ensemble de contraintes non-diagonales et $\mathcal{U} \subset \mathcal{U}_{ndet}(X)$ un ensemble de mises à jour. Alors, l'existence d'une solution au système linéaire d'inéquations Diophantiennes sur les variables $(max_x)_{x \in X}$ définit par :*

$$\{c \leq max_x \mid (x \sim c) \in \mathcal{C}\} \cup \{max_x \leq max_y + c \mid x :\sim y + c \in \mathcal{U}\} \tag{4.17}$$

Implique que le problème du vide sur classe des $UTA(\mathcal{C},\mathcal{U})$ soit décidable.

En conclusion, nous avons donc montré que sous des conditions idoines le graphe des régions vérifie bien ($\ddagger_1$) et ($\ddagger_2$) et possède un nombre fini de régions et de transitions. Comme dans la section 1.6 du chapitre 1, on peut construire l'automate des régions de la même manière que suggérée par Alur et Dill [AD94] et en déduire la décidabilité du problème du vide comme présenté dans le chapitre 1.

4.1.3 Complexité

Pour finir, nous allons rapidement établir la complexité du problème du vide pour un automate temporisé sans garde diagonales et avec mises à jour quelconques. La preuve suivante est inspirée de [AL02].

On considère que l'entrée correspond au codage d'un automate temporisé $\mathcal{A}$. Le problème du vide sur un automate peut être réduit au problème de l'accessibilité pour ce même automate. Or, le problème de décider si un état r de $\mathcal{A}$ est accessible à partir de l'état initial q_0 de $\mathcal{A}$ peut être réalisé par un algorithme non-déterministe qui construit étape par étape un chemin dans le graphe des régions qui mène à un état de contrôle $\langle r, v\rangle$ avec $v \in \mathbb{T}^X$. L'algorithme est le suivant :

Algorithme 1. *Si l'état de contrôle courant est tel que $\langle q, v\rangle = \langle r, v\rangle$, alors la réponse est* «oui»*. Sinon l'algorithme devine l'état de contrôle suivant $\langle q', v'\rangle$ qui est sur le chemin qui conduit à r. Cette étape se fait en temps polynômial en se basant sur l'automate temporisé original.*

Chaque étape de cet algorithme nécessite uniquement de se rappeler que de deux états de contrôle. De plus, chaque état de contrôle peut être codé en espace polynômial (un entier pour coder l'état et deux entiers par horloge pour symboliser les bornes de l'intervalle où se trouve l'horloge et, enfin, l'ordre relatif des horloges). Enfin, d'après le théorème de Savitch [Sav70], le fait que l'algorithme soit non-déterministe n'est pas essentiel. Ce qui nous permet de conclure que le problème est P-space facile.

Comme, de plus, la classe des automates temporisés sans garde diagonale et avec mises à jour contient l'ensemble des automates temporisés d'Alur et Dill, on en déduit que le problème du vide est *P-space dur* pour cette classe [AD94]. Ce qui permet de conclure que le problème du vide pour les automates temporisés avec mises à jour est *P-space complet.*

Nous allons à présent passer à la démonstration de la décidabilité du problème du vide sur les automates temporisés avec mises à jour et gardes générales. Comme on va le voir la technique de preuve que nous allons utiliser est similaire à celle que nous venons de décrire. Cependant, certaines conditions diffèrent drastiquement du cas que nous venons d'étudier.

4.2 Mises à jour avec contraintes quelconques

Nous nous intéressons ici aux automates temporisés avec mises à jour avec des gardes générales. En fait, comme pour le cas précédent, nous nous restreindrons au fragment dont la décidabilité n'a pas encore été établie dans le tableau 3.1 page 61. Formellement, on se focalise sur la classe d'automates $Aut(\mathcal{C},\mathcal{U}_{det})$, avec $up \in \mathcal{U}_{det}$ si et seulement si up est défini par la grammaire suivante :

$$up = \bigwedge_{x\in Z} up_x, \text{ avec } Z \subseteq X \text{ et } up_x ::= det_x \,|\, inf_x \,|\, int_x, \text{ avec :}$$

- $det_x ::= x := c \,|\, x := y$, avec $c \in \mathbb{N}$ et $y \in X$,
- $inf_x ::= x : \lhd\, c \,|\, inf_x \wedge inf_x$, avec $\lhd \in \{<, \leq\}$, et $c \in \mathbb{N}$,
- $int_x ::= x :\in (c, d)$, avec ')', '(' $\in$ {']', '['} , et $c, d \in \mathbb{N}$.

Autrement dit, on définit une mise à jour up comme la conjonction de ses mises à jour locales (det_x, inf_x, int_x). Et on note $up = (up_x)_{x \in Z}$ avec $Z \subset X$ et up_x une mise à jour locale à x, pour tout $x \in Z$.

La notion de région est ici exactement identique à celle introduite au chapitre 1. Néanmoins, nous allons reformuler rapidement cette définition afin de représenter les contraintes non plus sous la forme de formules logiques mais d'intervalles. Cette dernière représentation étant plus simple à manipuler lorsqu'il s'agit d'automates temporisés avec mises à jour.

4.2.1 Les régions d'horloges

On se donne donc X un ensemble fini d'horloges et $\lambda = ((max_x)_{x \in X}, (max_{x,y})_{(x,y) \in X^2})$ un ensemble de constantes entières. Comme précédemment, on définit $\mathcal{I}_x$ pour tout $x \in X$ comme étant l'ensemble des *intervalles élémentaires* de x (voir 7 page 64). Et on se donne X_0, X_{frac} et X_∞ comme définit dans la notation 4 page 64. Enfin, on se donne $\mathcal{J}_{x,y}$ l'ensemble des *intervalles élémentaires diagonaux*.

Définition 10. *Soient X un ensemble fini d'horloges et $\lambda = ((max_x)_{x \in X}, (max_{x,y})_{(x,y) \in X^2})$ un ensemble de constantes entières. Alors, on appelle $\mathcal{J}_{x,y}$ l'ensemble des* intervalles élémentaires diagonaux *de (x, y) issus de λ avec :*

$$\begin{aligned} \mathcal{J}_{x,y} = & \{\ ]-\infty, -max_{y,x}[\ \} \cup \{\ ]max_{x,y}, +\infty[\ \} \cup \\ & \{\{d\} \mid max_{x,y} \geq d \geq -max_{y,x}\} \cup \\ & \{\ ]d, d+1[\ \mid\ max_{x,y} > d \geq -max_{y,x}\} \end{aligned} \tag{4.18}$$

Finalement, une région se définit donc naturellement comme suit :

Définition 11. *Soient X un ensemble fini d'horloges et $\lambda = ((max_x)_{x \in X}, (max_{x,y})_{(x,y) \in X^2})$ un ensemble de constantes entières et avec $(\mathcal{I}_x)_{x \in X}$ et $(\mathcal{J}_{x,y})_{(x,y) \in X^2}$ l'ensemble, respectivement, des intervalles élémentaires et des intervalles élémentaires diagonaux qui sont associés à λ. Soient aussi :*

- *$(I_x)_{x \in X}$ un ensemble d'intervalles élémentaires tels que pour tout $x \in X$ on ait $I_x \in \mathcal{I}_x$,*
- *$(J_{x,y})_{(x,y) \in X^2}$ un ensemble d'intervalles élémentaires diagonaux tels que pour tous les couples $(x, y) \in X^2$ on ait $J_{x,y} \in \mathcal{J}_{x,y}$,*
- *$\succ$ un préordre sur les horloges $x \in X_{frac}$.*

La région *$((I_x)_{x \in X}, (J_{x,y})_{(x,y) \in X^2}, \succ)$ est définie comme l'ensemble de valuations α tel que :*

$$\alpha = \left\{ \; v \in \mathbb{T}^X \;\middle|\; \begin{array}{l} \forall x \in X,\ v(x) \in I_x, \\ \forall (x,y) \in X^2,\ \textit{avec } x \in X_\infty \textit{ ou } y \in X_\infty,\ v(x) - v(y) \in J_{x,y}, \\ \forall x, y \in X_{frac},\ x \succ y \Leftrightarrow frac(v(x)) \geq frac(v(y)) \end{array} \right\} \tag{4.19}$$

Avec , $\mathcal{R}_\lambda$ l'ensemble fini de toutes les régions engendrées par λ. Par commodité, nous noterons $\alpha = ((I_x)_{x \in X}, (J_{x,y})_{(x,y) \in X^2}, \succ)$.

À nouveau, la notion de compatibilité est identique à celle de la définition 4 page 39. Exactement comme dans le chapitre 1, on vérifie par construction qu'un ensemble de contraintes quelconques issues de $\mathcal{C}$ est compatible avec l'ensemble des régions $\mathcal{R}_\lambda$.

Proposition 3. *Soient X un ensemble fini d'horloges, $C \subseteq \mathcal{C}(X)$ un ensemble de contraintes d'horloges et $\lambda = ((max_x)_{x\in X}, (max_{x,y})_{(x,y)\in X^2})$ un ensemble de constantes entières tel que :*

- $\forall x \in X,\ max_x = \max\{c \mid (x \sim c) \in C\}$,
- $\forall (x, y) \in X^2$, *si on pose* $C_{x,y} = \{(x - y \sim c) \in C \text{ et } (y - x \sim c) \in C\}$, *on définit :*

$$max_{x,y} = \begin{cases} \max(c \mid (x-y\sim c) \in C_{x,y} \text{ et } (y-x\sim -c) \in C_{x,y}), & si\ C_{x,y} \neq \emptyset \\ 0, & si\ C_{x,y} = \emptyset \end{cases}$$

Alors, C et $\mathcal{R}_\lambda$ sont compatibles.

Nous avons donc rappelé la définition des régions pour des automates temporisés classiques. En outre, nous avons montré que l'ensemble de régions $\mathcal{R}_\lambda$ est toujours compatible avec n'importe quel sous-ensemble de contraintes de $\mathcal{C}$. Dans la section suivante, nous allons étendre ce graphe des régions aux mises à jour.

4.2.2 Le graphe des régions

Le principe de la démonstration est similaire à celui du chapitre 1 mais la présence des mises à jour qui étendent les remises à zéro modifie le graphe des régions.

Soit $\mathcal{R}$ un ensemble fini de régions formant une partition de $\mathbb{T}^X$. Le *graphe des régions* $\mathcal{G}$ associé à $\mathcal{R}$ établit les relations de précédence qu'ont les régions entre elles. Il est défini par $\mathcal{G} = (\mathcal{R}, \rightarrow)$ où la fonction de transition $\rightarrow \subset \mathcal{R}\times\mathcal{R}$ est définie à partir des fonctions de transitions $\xrightarrow{time}$ et $\xrightarrow{update}$ (voir 4.4 et 4.5 page 66) :

- $\xrightarrow{time}$: représente l'écoulement du temps :

$$\forall \alpha, \alpha' \in \mathcal{R},\ \alpha \xrightarrow{time} \alpha' \quad \stackrel{def}{\Longleftrightarrow} \quad \exists v \in \alpha,\ \exists t > 0,\ v + t \in \alpha'$$

- $\xrightarrow{update}$: représente la mise à jour d'horloges de X :

$$\forall \alpha, \alpha' \in \mathcal{R},\ \alpha \xrightarrow{update} \alpha' \quad \stackrel{def}{\Longleftrightarrow} \quad \exists v \in \alpha,\ \exists up \in \mathcal{U}_{ndet},\ \exists v' \in \alpha', v' \in up(v)$$

La fonction de transitions $\rightarrow$ du graphe des régions $\mathcal{G} = (\mathcal{R}, \rightarrow)$ est définie par l'union des transitions *time* et *update* :

$$\rightarrow\ =\ \xrightarrow{time} \cup \xrightarrow{update}$$

Comme dans la section 4.1.2 page 65, on définit les conditions suivantes :

$$\forall \alpha, \alpha' \in \mathcal{R},\ \text{tel que } \alpha \xrightarrow{time} \alpha' \Longrightarrow \forall v \in \alpha, \exists t \geq 0, v + t \in \alpha' \qquad (\ddagger_1)$$

$$\forall \alpha, \alpha' \in \mathcal{R},\ \text{tel que } \alpha \xrightarrow{update} \alpha' \Longrightarrow \forall v \in \alpha,\ \exists up \in \mathcal{U}_{ndet},\ \exists v' \in \alpha',\ v' \in up(v) \qquad (\ddagger_2)$$

Le but étant de montrer que les conditions $(\ddagger_1)$ et $(\ddagger_2)$ sont vraies si l'on a un ensemble de mises à jour qui vérifient certaines conditions.

Théorème 14. *Soient X un ensemble fini d'horloges, $C \subseteq \mathcal{C}(X)$ un ensemble fini de contraintes, $U \subseteq \mathcal{U}_{det}(X)$ un ensemble fini de mises à jour et $\lambda = ((max_x)_{x\in X}, (max_{x,y})_{(x,y)\in X^2})$ un ensemble de constantes entières tel que :*

$$\forall (x \sim c) \in C, \text{ avec } max_x \geq c, \tag{4.20}$$
$$\forall (x - y \sim c) \in C, \text{ avec } max_{x,y} \geq c \geq -max_{y,x} \tag{4.21}$$
$$\forall (x :\sim c) \in U, \text{ avec } \sim \in \{<, \leq, =\}, \text{ on vérifie :}$$
$$\bullet\ max_x \geq c, \tag{4.22}$$
$$\bullet\ \forall y \in X,\ max_y \geq max_{y,x} + c, \tag{4.23}$$
$$\bullet\ \forall y \in X,\ max_y \geq max_{x,y} + c, \tag{4.24}$$
$$\forall (x := y) \in U, \text{ on vérifie :}$$
$$\bullet\ max_y \geq max_x, \tag{4.25}$$
$$\bullet\ \forall z \in X,\ max_{y,z} \geq max_{x,z} \tag{4.26}$$
$$\bullet\ \forall z \in X,\ max_{z,x} \geq max_{z,y} \tag{4.27}$$
$$\forall (x :\in (c, d)) \in U, \text{ avec `)', `(' } \in \{\text{`]', `['}\}, \text{ on vérifie :}$$
$$\bullet\ max_x \geq d, \tag{4.28}$$
$$\bullet\ \forall y \in X,\ max_y \geq max_{y,x} + d. \tag{4.29}$$
$$\bullet\ \forall y \in X,\ max_y \geq max_{x,y} + d. \tag{4.30}$$

Alors, la partition $\mathcal{R}_\lambda$ est compatible avec l'ensemble des contraintes C et le graphe des régions $\mathcal{G}_\lambda = (\mathcal{R}_\lambda, \rightarrow)$ vérifie les conditions ($\ddagger_1$) et ($\ddagger_2$).

Il est facile de déduire que l'ensemble des contraintes C est compatible avec $\mathcal{R}_\lambda$ grâce aux conditions (4.20) et (4.21) et à la proposition 3. Nous allons ensuite successivement démontrer que ($\ddagger_1$) puis ($\ddagger_2$) sont vraies pour $\mathcal{G}_\lambda = (\mathcal{R}_\lambda, \rightarrow)$.

Lemme 14.1. *Soient X un ensemble fini d'horloges et $\lambda = ((max_x)_{x\in X}, (max_{x,y})_{(x,y)\in X^2})$ un ensemble de constantes entières. Alors, la condition ($\ddagger_1$) est vraie sur $\mathcal{R}_\lambda$.*

Démonstration. La preuve suit exactement le même schéma que celui du lemme 2.1 page 41. □

La dernière étape de cette preuve est de montrer la propriété ($\ddagger_2$).

Lemme 14.2. *Soient X un ensemble fini d'horloges, $U \subseteq \mathcal{U}_{det}$ un ensemble fini de mises à* *jour sur X et $\lambda = ((max_x)_{x\in X}, (max_{x,y})_{(x,y)\in X^2})$ un ensemble de constantes entières tels que :*

$$\forall (x :\sim c) \in U, \text{ avec } \sim\in \{<, \leq, =\}, \text{ on vérifie :}$$

$$\bullet\ max_x \geq c, \tag{4.22}$$

$$\bullet\ \forall y \in X,\ max_y \geq max_{y,x} + c, \tag{4.23}$$

$$\bullet\ \forall y \in X,\ max_y \geq max_{x,y} + c, \tag{4.24}$$

$$\forall (x := y) \in U, \text{ on vérifie :}$$

$$\bullet\ max_y \geq max_x, \tag{4.25}$$

$$\bullet\ \forall z \in X,\ max_{y,z} \geq max_{x,z} \tag{4.26}$$

$$\bullet\ \forall z \in X,\ max_{z,x} \geq max_{z,y} \tag{4.27}$$

$$\forall (x :\in (c, d)) \in U, \text{ avec } `)', `(' \in \{`]', `['\},\ \text{ on vérifie :}$$

$$\bullet\ max_x \geq d, \tag{4.28}$$

$$\bullet\ \forall y \in X,\ max_y \geq max_{y,x} + d. \tag{4.29}$$

$$\bullet\ \forall y \in X,\ max_y \geq max_{x,y} + d. \tag{4.30}$$

Alors, la propriété ($\ddagger_2$) est vraie sur $\mathcal{R}_\lambda$.

$$\forall \alpha, \alpha' \in \mathcal{R},\ \text{tel que } \alpha \xrightarrow{update} \alpha' \Longrightarrow \forall v \in \alpha,\ \exists up \in \mathcal{U}_{ndet},\ \exists v' \in \alpha',\ v' \in up(v) \tag{$\ddagger_2$}$$

Démonstration. Comme dans la première preuve du théorème 13, nous commencerons par prouver la propriété pour des mises à jour simples puis nous étendrons le résultat à la conjonction de mises à jour locales (det_x, inf_x, int_x).

Dans cette première partie de la preuve nous présumerons que la mise à jour up est connue et que c'est une mise à jour simple sur l'horloge $z \in X$ telle que :

$$up ::= z :\sim c \,|\, z := y \,|\, z :\in (c, d), \text{ avec } y \in X, c \in \mathbb{N}, \text{ et } \sim\in \{<, \leq, =\}.$$

On notera $\succ_{|Y}$ la restriction de $\succ\subset X \times X$ à $Y \subset X$. Ou plus formellement :

$$\succ_{|Y}=\succ \cap (Y \times Y) \text{ avec } Y \subseteq X \tag{4.31}$$

Soit une région $\alpha = ((I_x)_{x\in X}, (J_{x,y})_{(x,y)\in X^2}, \succ) \in \mathcal{R}_\lambda$ et une mise à jour simple up sur l'horloge $z \in X$. Alors, la région $\alpha' = ((I'_x)_{x\in X}), (J'_{x,y})_{(x,y)\in X^2}, \succ') \in \mathcal{R}_\lambda$ est dans $up(\alpha)$ si et seulement si :

- $I'_x = I_x$, pour toute horloge $x \neq z$,
- $J'_{x,y} = J_{x,y}$, pour toutes les horloges $x, y \neq z$.

Et si on a :

Si $\mathbf{up} = (\mathbf{z} :\sim \mathbf{c})$ (resp. $\mathbf{up} = (\mathbf{z} :\in (\mathbf{c}, \mathbf{d}))$) : I'_z peut être n'importe quel intervalle de $\mathcal{I}_z$ qui intersecte l'intervalle $\{v \in \mathbb{T} \mid v \sim c\}$ (resp. $\{v \in \mathbb{T} \mid v \in (c, d)\}$), on distingue les cas suivants :

- Soit $I'_z = \{d\}$ et on a :
  - $X'_{frac} = X_{frac} \setminus \{z\}$,
  - $\succ'=\succ_{|X'_{frac}}$,

- $J'_{x,z} =]max_{x,z}, +\infty[$, si $x \in X_\infty$.

  En effet, si v est une valuation telle que $v(x) > max_x$ et comme $v'(z) \sim c$, alors nous en déduisons que $v(x) - v'(z) > max_x - c$. Or, par hypothèse, on sait que $max_x \geq max_{x,z} + c$ (hypothèse 4.23), on obtient $v(x) - v'(z) > max_{x,z}$. Ce qui nous permet de conclure que $J'_{x,z} =]max_{x,z}, +\infty[$.
- $J'_{z,x} =]-\infty, -max_{z,x}[$, si $x \in X_\infty$.

  En effet, si v est une valuation telle que $v(x) > max_x$ et comme $v'(z) \sim c$, alors nous en déduisons que $v'(z) - v(x) < c - max_x$. Or, par hypothèse, on sait que $max_x \geq max_{z,x} + c$ (hypothèse 4.24), on obtient $v'(z) - v(x) < -max_{z,x}$. Ce qui nous permet de conclure que $J'_{z,x} =]-\infty, -max_{z,x}[$.

- Soit $I'_z =]d, d+1[$ et on a :
  - $X'_{frac} = X_{frac} \cup \{z\}$,
  - $\succ'$ est un préordre total quelconque sur X'_{frac} qui coïncide avec $\succ$ sur $X'_{frac} \setminus \{z\}$,
  - $J'_{x,z} =]max_{x,z}, +\infty[$, si $x \in X_\infty$ et $J'_{z,x} =]-\infty, -max_{z,x}[$, si $x \in X_\infty$. Pour les mêmes raisons que précédemment.

Si up $= (\mathbf{z} := \mathbf{y})$: I'_z est défini par :
- Si $I_y = \{d\}$, alors $I'_z = \{d\}$ si $d \leq max_z$ et $I'_z =]max_z, +\infty[$ sinon,
- Si $I_y =]d, d+1[$, alors $I'_z =]d, d+1[$ si $d \leq max_z$ et $I'_z =]max_z, +\infty[$ sinon,
- Si $I_y =]max_y, +\infty[$, alors $I'_z =]max_z, +\infty[$ car $max_z \leq max_y$ (hypothèse 4.25).

À présent que I'_z est défini, on distingue plusieurs cas :
- Soit $I'_z = \{d\}$ et on a :
  - $X'_{frac} = X_{frac} \cup \{z\}$,
  - $\succ' = \succ_{|X'_{frac}}$,
  - $J'_{z,x} =]max_{z,x}, +\infty[$ (resp. $]-\infty, -max_{x,z}[$) si on a $J_{y,x} \subset]max_{z,x}, +\infty[$ (resp. $]-\infty, -max_{x,z}[$) et $J'_{z,x} = J_{y,x}$ sinon. En effet, comme $max_{y,x} \geq max_{z,x}$ (hypothèse 4.26), on a $]max_{y,x}, +\infty[\subset]max_{z,x}, +\infty[$ et, réciproquement, comme on a $max_{x,y} \geq max_{x,z}$ (hypothèse 4.27), on a $]-\infty, -max_{x,y}[\subset]-\infty, -max_{x,z}[$.
  - $J'_{x,z} =]max_{x,z}, +\infty[$ (resp. $]-\infty, -max_{z,x}[$) si on a $J_{x,y} \subset]max_{x,z}, +\infty[$ (resp. $]-\infty, -max_{z,x}[$) et $J'_{x,z} = J_{x,y}$ sinon. L'argument est le même que celui développé précédemment.
- Soit $I'_z =]d, d+1[$ et on a :
  - $X'_{frac} = X_{frac} \cup \{z\}$,
  - $\succ'$ est un préordre total quelconque sur X'_{frac} qui coïncide avec $\succ$ sur $X'_{frac} \setminus \{z\}$,
  - $J'_{z,x}$ et $J'_{x,z}$ sont définis comme précédemment.
- Soit $I'_z =]max_z, +\infty[$ et on a :
  - $X'_{frac} = X_{frac} \cup \{z\}$,
  - $\succ' = \succ_{|X'_{frac}}$,
  - Le calcul de $J'_{z,x}$ (et $J'_{x,z}$) nécessite de distinguer plusieurs cas qui dépendent de la forme de I_x et I_y :

1. Si $I_x = \{e\}$ et $I_y = \{f\}$, alors on a :

$$J'_{z,x} = \begin{cases} \{f-e\}, & \text{si } max_{z,x} \geq f-e \geq -max_{x,z} \\ ]max_{z,x}, +\infty[, & \text{si } f-e > max_{z,x} \\ ]-\infty, -max_{x,z}[, & \text{si } -max_{x,z} > f-e \end{cases}$$

2. Si $I_x = \{e\}$ et $I_y =]f, f+1[$, alors on a :

$$J'_{z,x} = \begin{cases}]f-e-1, f-e[, & \text{si } max_{z,x} > f-e-1 \geq -max_{x,z} \\ ]max_{z,x}, +\infty[, & \text{si } f-e-1 \geq max_{z,x} \\ ]-\infty, -max_{x,z}[, & \text{si } -max_{x,z} > f-e-1 \end{cases}$$

3. Si $I_x = \{e\}$ et $I_y =]max_y, +\infty[$, alors $J'_{z,x}$ est l'unique intervalle de $\mathcal{J}_{z,x}$ qui contient $J_{y,x}$. L'argument est le même que dans le cas $I'_z = \{d\}$ et repose sur les hypothèses 4.26 et 4.27.
4. Si $I_x =]e, e+1[$ et $I_y = \{f\}$, alors ce cas est identique au cas 2. ci-dessus.
5. Si $I_x =]e, e+1[$ et $I_y =]f, f+1[$, alors on a :
   (a) Si $z \succ x$ et $x \succ y$, alors :

$$J'_{z,x} = \begin{cases} \{f-e\}, & \text{si } max_{z,x} \geq f-e \geq -max_{x,z} \\ ]max_{z,x}, +\infty[, & \text{si } f-e > max_{z,x} \\ ]-\infty, -max_{x,z}[, & \text{si } -max_{x,z} > f-e \end{cases}$$

   (b) Si $z \succ x$ et $x \not\succ z$, alors :

$$J'_{z,x} = \begin{cases}]f-e, f-e+1[, & \text{si } max_{z,x} > f-e \geq -max_{x,z} \\ ]max_{z,x}, +\infty[, & \text{si } f-e \geq max_{z,x} \\ ]-\infty, -max_{x,z}[, & \text{si } -max_{x,z} > f-e \end{cases}$$

   (c) Si $z \not\succ x$ et $x \succ y$, alors :

$$J'_{z,x} = \begin{cases}]f-e-1, f-e[, & \text{si } max_{z,x} > f-e-1 \geq -max_{x,z} \\ ]max_{z,x}, +\infty[, & \text{si } f-e-1 \geq max_{z,x} \\ ]-\infty, -max_{x,z}[, & \text{si } -max_{x,z} > f-e-1 \end{cases}$$

6. Si $I_x =]e, e+1[$ et $I_y =]max_y, +\infty[$, alors ce cas est identique au cas 3. ci-dessus.
7. Si $I_x =]max_x, +\infty[$, alors ce cas est identique aux cas 3. ci-dessus.

Nous avons donc caractérisé la relation $\xrightarrow{update}$ plus précisément. La preuve de la propriété ($\ddagger_2$) suit exactement le même schéma que dans la démonstration du lemme 13.2 page 67. C'est à dire que l'on montre que pour n'importe que couple de régions α et $\alpha' \in up(\alpha)$, on a $\forall v \in \alpha$, $\exists v' \in \alpha'$ tel que $v' = up(v)$.

Puis, on étend le résultat des mises à jour simples aux mises à jour locales en vérifiant que l'on peut décider du vide des mises à jour.

Et enfin, on montre que la composition de mises à jour locales ne remet pas en cause la propriété ($\ddagger_2$). □

À partir de ces deux lemmes nous pouvons donc déduire le théorème 14 page 76, énoncé au début de la section.

Un corollaire crucial, lié à ce théorème, est que quelque soit $C \subseteq \mathcal{C}$ un ensemble de contraintes et $U \subseteq \mathcal{U}_{det}$ un ensemble de mises à jour générales, alors le problème du vide est décidable pour un automate utilisant C et U.

Ce résultat découle directement du théorème 14 et du fait que le système d'inéquations induit par l'ensemble des contraintes et des mises à jour a toujours une solution. En effet, le système suivant admet toujours une solution :

$$\begin{aligned}
&\{c \leq max_x \mid z \sim c \in C\} \\
\cup&\{c \leq max_{x,y} \mid x - y \sim c \in C\} \\
\cup&\{c \leq max_x, c + max_{x,y} \leq max_y \mid (x :\sim c) \in U \text{ avec } \sim \in \{<, \leq, =\}\} \\
\cup&\{d \leq max_x, d + max_{x,y} \leq max_y \mid (x :\in (c, d)) \in U \text{ avec `)', `(' } \in \{\text{`]', `['}\}\} \\
\cup&\{max_x \leq max_y,\ max_{x,z} \leq max_{y,z},\ max_{z,y} \leq max_{z,x} \mid (x := y) \in U\}
\end{aligned}$$

Autrement dit, en utilisant le théorème 14, tout automate temporisé défini avec des contraintes de $\mathcal{C}$ et des mises à jours de $\mathcal{U}_{det}$ est décidable.

Corollaire 14.1. *Soit $C \subseteq \mathcal{C}$ un ensemble de contraintes et $U \subseteq \mathcal{U}_{det}$ un ensemble de mises à jour générales. Alors, le problème du vide sur la classe $Aut(C, U)$ est décidable.*

Finalement, nous avons donc montré que pour tout automate temporisé avec des mises à jour prises dans $\mathcal{U}_{det}$, on peut décider du vide.

Ainsi, nous avons des résultats de décidabilité général du modèle des automates temporisés avec mises à jours avec gardes diagonales ou générales. La résolution du système d'inéquations que nous avons décrit permet de construire un ensemble de régions qui abstrait l'automate initial pour le problème de l'accessibilité.

4.2.3 Complexité

Comme pour le cas des automates temporisés avec gardes non diagonales, le problème du vide est P-space complet. Par une argumentation similaire, on en déduit le résultat de complexité (voir section 4.1.3 page 73).

Conclusion

Ce chapitre termine notre étude de la décidabilité des sous-classes des automates temporisés avec mises à jour. Nous avons donc établis que le fragment $Aut(\mathcal{C}, \mathcal{U}_{det})$ est décidable sans condition. Et que le fragment $Aut(\mathcal{C}_{df}, \mathcal{U}_{ndet})$ est décidable s'il existe une solution à un système d'inéquations Diophantiennes.

Théorème 15. *Soit $Aut(C, U)$ une classe d'automate temporisé avec mises à jour. Le problème du vide sur cette classe $Aut(C, U)$ est Pspace-complet, si on vérifie l'un des deux conditions suivantes :*

- *$Aut(C, U) \subseteq Aut(\mathcal{C}, \mathcal{U}_{det})$,*
- *$Aut(C, U) \subseteq Aut(\mathcal{C}_{df}, \mathcal{U}_{ndet})$, et de plus :*
  - *Pour toute mise à jour $(x :\sim c) \in U$, on a $c \leq max_x$,*

– *Pour toute mise à jour* $(x :\sim y + c) \in U$, *on a* $max_x \leq max_y + c$.

Ces résultats nous permettent de compléter notre tableau (4.1).

<table>
<tr><th></th><th>Mises à jours</th><th>Contraintes non diagonales</th><th>Contraintes générales</th></tr>
<tr><td rowspan="5">Déterministes</td><td>$x := 0$</td><td rowspan="3">Pspace</td><td rowspan="2">Pspace</td></tr>
<tr><td>$x := c,\ c \in \mathbb{Q}_+$</td></tr>
<tr><td>$x := y + c,\ c \in \mathbb{Q}_+$</td><td rowspan="3">Indécidable</td></tr>
<tr><td>$x := y - 1$</td><td rowspan="2">Indécidable</td></tr>
<tr><td>$x := y + c,\ c \in \mathbb{Q}$</td></tr>
<tr><td rowspan="8">non-déterministes</td><td>$x :< c,\ c \in \mathbb{Q}_+$</td><td rowspan="7">Pspace</td><td>Pspace</td></tr>
<tr><td>$x :> c,\ c \in \mathbb{Q}_+$</td><td rowspan="7">Indécidable</td></tr>
<tr><td>$x :< y$</td></tr>
<tr><td>$x :> y$</td></tr>
<tr><td>$x :< y + c,\ c \in \mathbb{Q}_+$</td></tr>
<tr><td>$x :> y + c,\ c \in \mathbb{Q}_+$</td></tr>
<tr><td>$y + c <: x :< y + d,$
$c, d \in \mathbb{Q}_+$</td></tr>
<tr><td>$y + c <: x :< z + d,$
$c, d \in \mathbb{Q}_+,\ y \neq z$</td><td>Indécidable</td></tr>
</table>

TAB. 4.1 – *Décidabilité des différentes sous-classes d'automates temporisés.*

Comme on peut le constater, la présence ou non de contraintes diagonales dans les gardes influence grandement la décidabilité du modèle. Ce fait pourrait paraître assez surprenant étant donné que les automates temporisés classiques étaient considérés comme équivalents (les deux modèles sont simulables l'un par l'autre). Cependant, on retrouve cette distinction lorsque l'on cherche à étendre les propriétés des régions sur des zones [Bou02b].

On peut aussi noter que le fragment décidable de la partie sans contrainte diagonale est assez important. Comme nous le verrons sur les exemples au chapitre 7 page 111, les mises à jour non déterministes qu'il autorise permettent de concevoir des modèles plus compacts et plus intuitifs que ceux que permettent les automates temporisés classiques. On peut notamment modéliser des protocoles de communication réseau réels sans grande difficulté. De plus, l'absence des gardes diagonales ne nuit pas outre mesure à la réalisation du modèle. En outre, à propos de l'absence de gardes diagonales, une récente découverte [Bou02b] a remis en question leur utilisation dans la plupart des logiciels de vérification basés sur les automates temporisés.

La question qui se pose à présent est de savoir si l'ajout des mises à jour augmente ou non le pouvoir d'expression du modèle des automates temporisés ? C'est de cette question que traite le chapitre 5 page 83.

Chapitre 5

Expressivité des automates avec mises à jour

"I don't know what you mean by 'glory'," Alice said.

Humpty Dumpty smiled contemptuously. "Of course you don't – till I tell you. I meant 'there's a nice knock-down argument for you!'"

"But glory doesn't mean 'a nice knock-down argument'," Alice objected.

"When I use a word," Humpty Dumpty said, in a rather scornful tone, "it means just what I choose it to mean – neither more nor less."

"The question is," said Alice, "whether you can make words mean so many different things."

— Lewis Carrol, *Through the Looking Glass*

Dans les chapitres 3 et 4, nous nous sommes focalisés sur la décidabilité et la complexité du modèle des automates temporisés avec mises à jour. Nous allons à présent nous intéresser à l'expressivité de ce modèle. Il s'agit de déterminer si les sous-classes décidables que nous avons identifiées ont un pouvoir d'expression strictement plus important ou seulement équivalent au modèle original. Ces résultats ont été publiés dans [BDFP00b].

Comme nous allons le voir, les mises à jour déterministes n'ajoutent rien au pouvoir expressif du modèle original. Par contre les mises à jour non-déterministes donnent au modèle un pouvoir d'expression strictement plus fort que celui du modèle original. Cependant, le modèle comprenant les mises à jour non-déterministes reste inclus ou égal au modèle contenant des actions silencieuses (ε-transitions).

Nous commencerons par introduire les notions d'équivalence de langage et de bisimilarité (forte et faible) [Par81, Mil89] pour les automates temporisés. Puis, nous prouverons la bisimilarité forte entre les automates temporisés classiques et les automates temporisés avec mises à jour déterministes. Puis, nous prouverons l'inclusion stricte de la classe des langages engendrés par les automates temporisés classiques dans la classe des langages engendrés par les automates temporisés avec mises à jour non-déterministes. Enfin, nous prouverons l'inclusion de la classe des langages engendrés par les automates temporisés avec mises à jour non-déterministes dans la classe des langages engendrés par les automates temporisés avec ε-transitions.

5.1 Notions d'équivalences entre automates temporisés

Nous présentons ici des notions permettant de comparer l'expressivité des différents modèles que nous avons présentés. Outre la notion classique d'équivalence de langages, nous allons utiliser la notion de *bisimilarité* qui permet de mettre en évidence une relation étroite entre deux systèmes de transitions.

Cependant, la notion de bisimilarité n'est définie que pour des systèmes de transitions non temporisés, nous allons donc définir deux transformations qui permettent d'appliquer la notion de bisimilarité sur les automates temporisés. Celles-ci permettront d'introduire une bisimulation forte (correspondance exacte transition à transition) et une bisimulation faible (correspondance exacte à une action silencieuse près).

Définition 12 (Équivalence de langages). *Soient $\mathcal{A}$ et $\mathcal{B}$ deux automates temporisés, $\mathcal{A}$ et $\mathcal{B}$ sont* équivalents du point de vue du langage *si $L(\mathcal{A}) = L(\mathcal{B})$. Et on note $\mathcal{A} \equiv_\ell \mathcal{B}$.*

Par extension, si on considère deux familles d'automates temporisés Aut_1 et Aut_2. On dit que Aut_1 et Aut_2 sont équivalents du point de vue du langage *si :*

1. $\forall \mathcal{A}_1 \in Aut_1,\ \exists \mathcal{A}_2 \in Aut_2$ *tel que* $L(\mathcal{A}_1) = L(\mathcal{A}_2)$,
2. $\forall \mathcal{A}_2 \in Aut_2,\ \exists \mathcal{A}_1 \in Aut_1$ *tel que* $L(\mathcal{A}_2) = L(\mathcal{A}_1)$.

Et on note $Aut_1 \equiv_\ell Aut_2$.

Introduisons à présent la notion de bisimilarité définie pour un système de transitions non temporisé $\mathcal{T}$ quelconque. La bisimulation est une relation qui met en correspondance transition à transition deux systèmes de transitions. On a une relation d'équivalence plus forte que la simple équivalence de langages.

Pour cela, nous rappelons la définition générale d'un *système de transitions* non temporisé, puis la définition de la relation de *similarité*. Enfin, nous définissons la bisimilarité classique.

Définition 13 (Système de transitions). *On appelle $\mathcal{T} = (S, Init, A, \rightarrow)$ un* système de transitions, *avec S un ensemble d'états, $Init \subseteq S$ un ensemble d'états initiaux, A un alphabet fini d'actions et $\rightarrow \subset S \times A \times S$ un ensemble de transitions étiquetées.*

Définition 14 (Similarité). *Soient $\mathcal{T}_1 = (S_1, Init_1, A, \rightarrow_1)$ et $\mathcal{T}_2 = (S_2, Init_2, A, \rightarrow_2)$ deux systèmes de transitions. On dit que $\mathcal{T}_1$* simule *$\mathcal{T}_2$, et on le note $\mathcal{T}_1 \sim \mathcal{T}_2$, s'il existe une relation $\rhd \subseteq S_1 \times S_2$ qui vérifie les conditions suivantes :*

- ***Initialisation*** : $\forall s_1 \in Init_1,\ \exists s_2 \in Init_2,\ s_1 \rhd s_2$
- ***Propagation*** : $(s_1 \rhd s_2) \wedge (s_1 \xrightarrow{e}_1 s_1') \Rightarrow \exists s_2' \in S_2,\ (s_2 \xrightarrow{e}_2 s_2') \wedge (s_1' \rhd s_2')$

Définition 15 (Bisimilarité). *Soient $\mathcal{T}_1$ et $\mathcal{T}_2$ deux systèmes de transitions. On dit que $\mathcal{T}_1$ et $\mathcal{T}_2$ sont* bisimilaires, *et on le note $\mathcal{T}_1 \approx \mathcal{T}_2$, s'il existe une relation $\rhd \subseteq S_1 \times S_2$ qui vérifie les conditions suivantes :*

- ***Initialisation*** : $\left\{ \begin{array}{ll} \forall s_1 \in Init_1, \exists s_2 \in Init_2, & s_1 \rhd s_2 \\ \forall s_2 \in Init_2, \exists s_1 \in Init_1, & s_1 \rhd s_2 \end{array} \right.$
- ***Propagation*** : : $\left\{ \begin{array}{l} (s_1 \rhd s_2) \wedge (s_1 \xrightarrow{e}_1 s_1') \Rightarrow \exists s_2' \in S_2, (s_2 \xrightarrow{e}_2 s_2') \wedge (s_1' \rhd s_2') \\ (s_1 \rhd s_2) \wedge (s_2 \xrightarrow{e}_2 s_2') \Rightarrow \exists s_1' \in S_1, (s_1 \xrightarrow{e}_1 s_1') \wedge (s_1' \rhd s_2') \end{array} \right.$

La bisimilarité porte sur des systèmes de transitions non temporisés et l'appliquer aux automates temporisés peut se faire par le biais d'une transformation d'un automate temporisé en un système de transitions non temporisé. Nous allons présenter ici deux transformations. La première impose une correspondance exacte entre deux automates temporisés, on parle de *bisimulation forte*. La deuxième, plus laxiste, permet de mettre en correspondance deux systèmes de transitions de la même façon mais en ignorant les transitions marquées par une action silencieuse définie au préalable, on parle de *bisimulation faible*.

Afin de considérer le cas le plus général possible, nous adjoindrons le mot vide ε à l'alphabet afin de représenter l'action silencieuse. Néanmoins cette action silencieuse n'est réellement utile que dans le cadre de la bisimulation faible.

Commençons donc par définir la bisimulation forte.

Définition 16. *Soient $\mathcal{A} = (\Sigma, Q, T, I, F, R, X) \in Aut(\mathcal{C},\mathcal{U})$ un automate temporisé avec mises à jour et $\mathcal{T}_s(\mathcal{A}) = (S, Init, A, \rightarrow)$ le système de transitions défini par :*

- *$S = Q \times \mathbb{T}^X$ est l'ensemble d'états,*
- *$Init = \{\langle q, v\rangle \mid q \in I \text{ et } \forall x \in X,\ v(x) = 0\}$ est l'ensemble des états initiaux,*
- *$A = \Sigma \cup \{\varepsilon\} \cup \mathbb{Q}_+$,*
- *$\rightarrow \subset S \times A \times S$ est l'ensemble de transitions telles que :*
  - *$\forall a \in \Sigma \cup \{\varepsilon\}$, $\langle q, v\rangle \xrightarrow{a} \langle q', v'\rangle$ ssi $\exists (q, g, a, up, q') \in T$ avec $v \models g$ et $v' \in up(v)$,*
  - *$\forall d \in \mathbb{Q}_+$, $\langle q, v\rangle \xrightarrow{d} \langle q', v'\rangle$ ssi $q = q'$ et $v' = v + d$.*

Une fois la définition de $\mathcal{T}_s(\mathcal{A})$ établie, la bisimilarité forte sur les automates temporisés se définit comme suit :

Définition 17. *Soient $\mathcal{A}$, $\mathcal{B} \in Aut(\mathcal{C},\mathcal{U})$ deux automates temporisés avec mises à jour. $\mathcal{A}$ et $\mathcal{B}$ sont dits* fortement bisimilaires *si $\mathcal{T}_s(\mathcal{A}) \approx \mathcal{T}_s(\mathcal{B})$. Et on note $\mathcal{A} \equiv_s \mathcal{B}$.*

Nous allons à présent nous intéresser à la bisimulation faible. Comme nous l'avons dit plus haut, le but de cette relation est d'affaiblir la relation de bisimulation forte en introduisant une action silencieuse ε qui ne sera pas prise en compte dans la relation d'équivalence. On doit donc supprimer toute référence à cette action ε lors de la transformation de l'automate temporisé en un système de transitions non temporisé.

Définition 18. *Soient $\mathcal{A} = (\Sigma, Q, T, I, F, R, X) \in Aut(\mathcal{C},\mathcal{U})$ un automate temporisé avec mises à jour et $\mathcal{T}_w(\mathcal{A}) = (S, Init, A, \Rightarrow)$ un système de transitions tel que :*

- *$S = Q \times \mathbb{T}^X$ est l'ensemble d'états,*
- *$Init = \{\langle q, v\rangle \mid q \in I \text{ et } \forall x \in X,\ v(x) = 0\}$ est l'ensemble d'états initiaux,*
- *$A = \Sigma \cup \mathbb{Q}_+$,*
- *$\overset{a}{\Rightarrow} \subseteq S \times A \times S$ est un ensemble de transitions telles que :*
  - *$\forall a \in \Sigma$, $\langle q, v\rangle \overset{a}{\Rightarrow} \langle q', v'\rangle$ ssi $\langle q, v\rangle \xrightarrow{\varepsilon^*}\xrightarrow{a}\xrightarrow{\varepsilon^*} \langle q', v'\rangle$,*
  - *$\forall d \in \mathbb{Q}$, $\langle q, v\rangle \overset{d}{\Rightarrow} \langle q', v'\rangle$ ssi $\langle q, v\rangle \xrightarrow{\varepsilon^*}\xrightarrow{d_1}\xrightarrow{\varepsilon^*} \dots \xrightarrow{d_k}\xrightarrow{\varepsilon^*} \langle q', v'\rangle$ et $d = \sum_{i=1}^{k} d_i$.*

Comme précédemment, une fois la définition de $\mathcal{T}_w(\mathcal{A})$ établie, la définition de la bisimilarité faible sur les automates temporisés est facile.

Définition 19. *Soient $\mathcal{A}$, $\mathcal{B} \in Aut(\mathcal{C},\mathcal{U})$ deux automates temporisés avec mises à jour. $\mathcal{A}$ et $\mathcal{B}$ sont dit* faiblement bisimilaires *si $\mathcal{T}_w(\mathcal{A}) \approx \mathcal{T}_w(\mathcal{B})$. Et on note $\mathcal{A} \equiv_w \mathcal{B}$.*

Remarque 11. *On peut noter que lorsqu'on compare deux automates temporisés via une relation de bisimulation faible ou de bisimulation forte, on peut se contenter de considérer des variables entières (voir remarque 4 page 37) car si $\mathcal{A} \equiv_s \mathcal{B}$ (resp. $\mathcal{A} \equiv_w \mathcal{B}$) alors $\lambda\mathcal{A} \equiv_s \lambda\mathcal{B}$ (resp. $\lambda\mathcal{A} \equiv_w \lambda\mathcal{B}$) pour tout $\lambda \in \mathbb{R}_+^*$. Comme le signale la remarque 4 page 37, nous pouvons, ici aussi, nous restreindre aux seules constantes entières dans les gardes et les mises à jour et étendre le résultat aux constantes rationnelles.*

5.2 Pouvoir d'expression des mises à jour déterministes

Nous allons commencer par nous intéresser aux automates temporisés avec mises à jour déterministes. Comme nous l'avons notifié au début de ce chapitre, ceux-ci vérifient une relation de bisimulation forte avec les automates temporisés classiques. Ces résultats font partie du "folklore" des automates temporisés, sans que l'on puisse trouver une preuve finalisée dans la littérature. Nous présentons ici une preuve formelle de ce résultat étendu au fragment décidable des automates temporisés avec mises à jour déterministes tels qu'ils sont définis page 73.

Théorème 16. *Soit X un ensemble fini d'horloges, $\mathcal{U}_{det}(X)$ le fragment décidable de l'ensemble des mises à jour déterministes et $\mathcal{C}_{df}(X)$ l'ensemble des contraintes non-diagonales sur les horloges. Alors, on a :*

$$Aut(\mathcal{C}_{df},\mathcal{U}_{det}) \equiv_s Aut(\mathcal{C}_{df},\mathcal{U}_0) \tag{5.1}$$

Démonstration. On procède en trois étapes. La première consiste à montrer la bisimilarité forte entre les automates temporisés classiques et les automates temporisés avec des mises à jour du type $x:=y$ (voir lemme 16.1), on note cette classe de mises à jour $\mathcal{U}_{x:=y}$. La deuxième étape consiste à montrer que la classe $Aut(\mathcal{C}_{df},\mathcal{U}_{x:=y})$ est en bisimilarité forte avec les automates temporisés avec mises à jour de type $x:=y$ et $x:=c$ avec $c\in\mathbb{N}_+$ (voir lemme 16.2 page 88), on note cette classe de mises à jour $\mathcal{U}_{x:=y}\cup\mathcal{U}_{x:=c}$. Enfin, on montre que les automates temporisés avec des mises à jour du type $x:=y+d$ tel que $d\in\mathbb{Z}$ est en bisimilarité forte avec $Aut(\mathcal{C}_{df},\mathcal{U}_{x:=y})$ (voir lemme 16.3 page 90). Par transitivité, on en déduit le théorème 16.

Lemme 16.1. *Soient X un ensemble fini d'horloges et $\mathcal{U}_{x:=y}(X)$ l'ensemble des mises à jour de type $(x := y)$. Alors, on a :*

$$Aut(\mathcal{C}_{df},\mathcal{U}_{x:=y}) \equiv_s Aut(\mathcal{C}_{df},\mathcal{U}_0) \tag{5.2}$$

Démonstration. Soit $\mathcal{A} = (\Sigma, Q, T, I, F, R, X) \in Aut(\mathcal{C}_{df},\mathcal{U}_{x:=y})$ un automate temporisé avec des mises à jour de type $x := y$ et $x := 0$ uniquement. On cherche à construire un automate temporisé classique $\mathcal{B} = (\Sigma, Q', T', I', F', R', X) \in Aut(\mathcal{C}_{df},\mathcal{U}_0)$ tel que $\mathcal{A} \equiv_s \mathcal{B}$.

L'idée intuitive est de construire une copie de $\mathcal{A}$ pour chaque application $\sigma : X \to X$ et de remplacer les mises à jour $x := y$ par des transitions reliant la copie de départ à une copie où toutes les occurrences de x ont été substituées par y sur les gardes.

On pose :

– $Q' = Q \times X^X$,
– $I' = I \times \{\mathtt{Id}\}$ avec $\mathtt{Id} : X \to X$ tel que $\mathtt{Id}(x) = x$, $\forall x \in X$,
– $F' = F \times X^X$,
– $R' = R \times X^X$.

Finalement, pour tout état $(p, \sigma) \in Q'$ et toute transition $(p, g, a, up, q) \in T$, on construit une transition $((p, \sigma), g', a, up', (q', \sigma')) \in T'$.

Posons $up = \bigwedge_{x \in Y} up_x$ avec $\forall x \in Y$, $up_x = (x{:=}0)$ ou $up_x = (x{:=}y)$.

On définit tout d'abord σ' comme suit :

– Si $x \notin Y$, alors $\sigma'(x) = \sigma(x)$,
– Si $x \in Y$ et $up_x = (x := y)$, alors $\sigma'(x) = \sigma(y)$.

Reste à définir $\sigma'(x)$ quand on a $x \in Y$ et $up_x = (x := 0)$. Intuitivement, il suffit d'assigner à x n'importe quelle horloge non encore utilisée. Plus formellement, on définit un ensemble d'horloges $Z = (X \setminus Y) \cup \{x \in Y \mid up_x = (x := y)\}$ et une injection $\iota : X \setminus Z \to X \setminus \sigma'(Z)$.

Finalement, on pose :

– Si $x \in Y$ et $up_x = (x := 0)$, $\sigma'(x) = \iota(x)$.

Enfin, afin de compléter la définition de la transition, il suffit alors de poser :

– $g' = g[x \leftarrow \sigma(x)]$ (g' est obtenue à partir de g en substituant tout x par $\sigma(x)$),
– $up' = \bigwedge_{x \in X \setminus Z} (\sigma'(x) := 0)$.

Nous allons maintenant prouver la bisimilarité forte de $\mathcal{A}$ et de $\mathcal{B}$. Par construction les états de $\mathcal{T}_s(\mathcal{A})$ sont dans $Q \times \mathbb{T}^X$ et ceux de $\mathcal{T}_s(\mathcal{B})$ sont dans $Q \times X^X \times \mathbb{T}^X$. On définit alors $\rhd \subseteq (Q \times \mathbb{T}^X) \times (Q \times X^X \times \mathbb{T}^X)$ tel que :

$$\rhd = \big\{ \big[(q, v), (q', \sigma, v')\big] \mid (q' = q) \wedge (v = v' \circ \sigma) \big\} \tag{5.3}$$

Par définition on vérifie que $\rhd$ est bien une bisimulation entre $\mathcal{T}_s(\mathcal{A})$ et $\mathcal{T}_s(\mathcal{B})$. $\square$

Exemple 12. *Nous allons illustrer cette construction en appliquant la transformation que nous venons de décrire à l'automate temporisé $\mathcal{A}$ (fig. 5.1).*

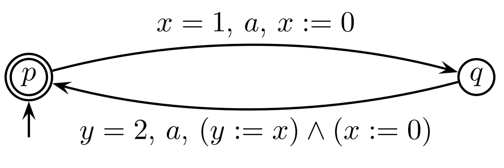

FIG. 5.1 – *Automate temporisé $\mathcal{A}$*

On énumère d'abord les différentes applications de X vers X :

$$\sigma_1 = \left| \begin{array}{l} (x, x) \\ (y, y) \end{array} \right. = \mathtt{Id} \quad \sigma_2 = \left| \begin{array}{l} (x, x) \\ (y, x) \end{array} \right. \quad \sigma_3 = \left| \begin{array}{l} (x, y) \\ (y, x) \end{array} \right. \quad \sigma_4 = \left| \begin{array}{l} (x, y) \\ (y, y) \end{array} \right.$$

On pose :

– $Q' = \{(p,\sigma_1),(p,\sigma_2),(p,\sigma_3),(p,\sigma_4),(q,\sigma_1),(q,\sigma_2),(q,\sigma_3),(q,\sigma_4)\}$,
– $I' = \{(p,\sigma_1)\}$,
– $R' = \{(p,\sigma_1),(p,\sigma_2),(p,\sigma_3),(p,\sigma_4)\}$.

Enfin, pour toutes les transitions $t = (p, g, a, up, q)$ de $\mathcal{A}$, on construit autant de transitions $((p,\sigma), g', a, up', (q,\sigma'))$ qu'il y a de $\sigma \in X^X$. L'automate résultant est représenté sur la figure 5.2, dans lesquelles les états inaccessibles ont été omis.

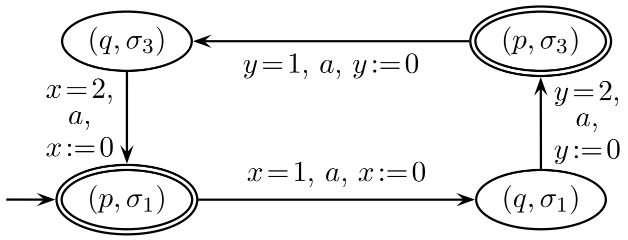

FIG. 5.2 – *Automate temporisé $\mathcal{B}$*

Intéressons nous maintenant à un modèle plus riche dans lequel des mises à jours de la forme $x:=c$ sont aussi autorisées. Nous allons montrer qu'en fait un tel automate est fortement bisimilaire à un automate ne contenant que des mises à jour de la forme $x:=y$.

Lemme 16.2. *Soient X un ensemble fini d'horloges et $\mathcal{U}_{x:=c}(X)$ l'ensemble des mises à jour de type $x := c$ avec $c \in \mathbb{Q}$. Alors, on a :*

$$Aut(\mathcal{C}_{df}, \mathcal{U}_{x:=y} \cup \mathcal{U}_{x:=c}) \equiv_s Aut(\mathcal{C}_{df}, \mathcal{U}_{x:=y}) \tag{5.4}$$

Démonstration. Soit $\mathcal{A} = (\Sigma, Q, T, I, F, R, X) \in Aut(\mathcal{C}_{df}, \mathcal{U}_{x:=y} \cup \mathcal{U}_{x:=c})$ un automate temporisé avec des mises à jour de type $x := y$ et $x := c$ avec $c \in \mathbb{Q}$). On cherche à construire un automate temporisé $\mathcal{B} = (\Sigma, Q', T', I', F', R', X) \in Aut(\mathcal{C}_{df}, \mathcal{U}_{x:=y})$ avec des mises à jour de type $x:=y$ et $x:=0$ uniquement, tel que $\mathcal{A} \equiv_s \mathcal{B}$.

Soit aussi C, l'ensemble des constantes apparaissant dans $\mathcal{A}$ dans les mises à jour de la forme $x:=c$. On définit $\Delta = X^C$ comme étant l'ensemble des fonctions de décalage.

Comme dans le lemme précédent, on va construire des copies de $\mathcal{A}$ dont on va modifier les gardes et substituer les transitions contenant des mises à jour de type $x := c$ par des transitions passant d'une copie à une autre et remettant les horloges concernées par des mises à jour de type $x:=c$ à zéro. Plus précisément, on construit autant de copies de $\mathcal{A}$ qu'il y a de façons d'assigner à une horloge la valeur d'une constante. Comme le nombre de constantes et d'horloges sont finis, on obtient un nombre fini de copies de $\mathcal{A}$ (cardinal de Δ). Pour chaque copie, on modifie les gardes en substituant $x \sim c$ par $x + \delta(x) \sim c$ (avec $\delta(x) \in \Delta$ la constante à laquelle on assigne x dans cette copie). Toutes les transitions contenant une mise à jour du type $x:=c$ sont substituées par des transitions remettant les horloges x à zéro et pointant vers l'état d'arrivé équivalent dans la copie d'automate pour laquelle $\delta(x)=c$.

On pose $\mathcal{B} = (\Sigma, Q', T', I', F', R', X)$ avec :

– $Q' = Q \times \Delta$,

- $I' = I \times \delta_0$ avec $\delta_0 \in \Delta$ tel que $\forall x \in X,\ \delta_0(x) = 0$,
- $F' = F \times \Delta$,
- $R' = R \times \Delta$,

Reste à définir l'ensemble des transitions T'.

Pour tout état $(p, \delta) \in Q'$ et toute transition $(p, g, a, up, q) \in T$ telle que $up = \bigwedge_{x \in Y} up_x$, avec $Y \subseteq X$ l'ensemble des horloges qui sont mises à jour dans up. On construit alors une transition $((p, \delta), g', a, up', (q, \delta'))$ de $\mathcal{B}$ en posant :

- $g' = g[x \leftarrow x + \delta(x)]$ (g' est obtenue à partir de g en substituant tout x par $x + \delta(x)$),
- $up' = up[(x := 0) \leftarrow (x := c)]$,
- $\forall x \in X,\ \delta'(x) = \begin{cases} c, & \text{si } (x := c) \in up \\ \delta(y), & \text{si } (x := y) \in up \\ \delta(x), & \text{si } x \notin Y \end{cases}$

Nous allons maintenant prouver la bisimilarité forte de $\mathcal{A}$ et de $\mathcal{B}$.

Par construction les états de $\mathcal{T}_s(\mathcal{A})$ sont dans $Q \times \mathbb{T}^X$ et ceux de $\mathcal{T}_s(\mathcal{B})$ sont dans $Q \times \Delta \times \mathbb{T}^X$. On définit alors $\rhd \subseteq (Q \times \mathbb{T}^X) \times (Q \times \Delta \times \mathbb{T}^X)$ tel que :

$$\rhd = \left\{ \left[(q, v), (q', \delta, v')\right] \mid (q' = q) \wedge (v = v' + \delta) \right\} \tag{5.5}$$

Il reste à vérifier que $\rhd$ est bien une bisimulation entre $\mathcal{T}_s(\mathcal{A})$ et $\mathcal{T}_s(\mathcal{B})$. On vérifie la condition d'initialisation en appliquant la définition de I', on a $(q, v_0) \in I \Leftrightarrow (q, \delta_0, v_0) \in I'$.

Supposons à présent que $(p, v) \rhd (p, \delta, v')$ et que $(p, v) \xrightarrow{g,a,up} (q, w)$. Par construction de la relation de transition de $\mathcal{B}$, on a $(p, \delta, v') \xrightarrow{g',a,up'} (q, \delta', w')$ avec :

- $g' = g[x \leftarrow x + \delta(x)]$ (g' est obtenue à partir de g en substituant tout x par $x + \delta(x)$),
- $up' = up[(x := 0) \leftarrow (x := c)]$,
- $\forall x \in X,\ \delta'(x) = \begin{cases} c, & \text{si } (x := c) \in up \\ \delta(y), & \text{si } (x := y) \in up \\ \delta(x), & \text{si } x \notin Y \end{cases}$

Donc, on en déduit que : $(w' + \delta')(x) = \begin{cases} c, & \text{si } (x := c) \in up \\ v'(y) + \delta(y), & \text{si } (x := y) \in up \\ v'(x) + \delta(x), & \text{sinon} \end{cases}$

Or, on sait par ailleurs que : $w(x) = \begin{cases} c, & \text{si } (x := c) \in up \\ v(y), & \text{si } (x := y) \in up \\ v(x), & \text{sinon} \end{cases}$

Finalement, comme on a posé que $(p, v) \rhd (p, \delta, v')$, on sait par définition (équation 5.5) que $v = v' + \delta$. On en déduit que $w = w' + \delta'$. Ce qui montre la bisimilarité entre $\mathcal{A}$ et $\mathcal{B}$. $\square$

Exemple 13. *Comme pour le lemme précédent, nous allons illustrer cette construction en appliquant la transformation que nous venons de décrire à un automate temporisé $\mathcal{A}$, qui est sensiblement différent (fig. 5.3 page suivante).*

On énumère d'abord les différentes applications de X vers C (Δ) :

$$\delta_0 = \left| \begin{array}{l} (x, 0) \\ (y, 0) \end{array} \right. = \mathtt{Id} \quad \delta_1 = \left| \begin{array}{l} (x, 10) \\ (y, 0) \end{array} \right. \quad \delta_2 = \left| \begin{array}{l} (x, 0) \\ (y, 10) \end{array} \right. \quad \delta_3 = \left| \begin{array}{l} (x, 10) \\ (y, 10) \end{array} \right.$$

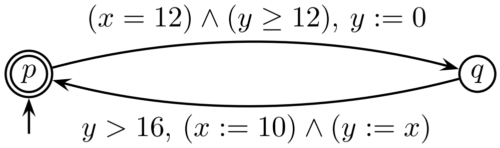

FIG. 5.3 – *Automate temporisé $\mathcal{A}$*

On pose :

- $Q' = \{(p,\delta_0),(p,\delta_1),(p,\delta_2),(p,\delta_3),(q,\delta_0),(q,\delta_1),(q,\delta_2),(q,\delta_3)\}$,
- $I' = \{(p,\delta_0)\}$,
- $R' = \{(p,\delta_0),(p,\delta_1),(p,\delta_2),(p,\delta_3)\}$.

Enfin, pour toutes les transitions $t = (p,g,a,up,q)$ de $\mathcal{A}$, on construit autant de transitions $((p,\delta),g',a,up',(q,\delta'))$ qu'il y a de $\delta \in \Delta$. L'automate résultant est représenté sur la figure 5.4. Nous avons cependant omis tous les états inaccessibles.

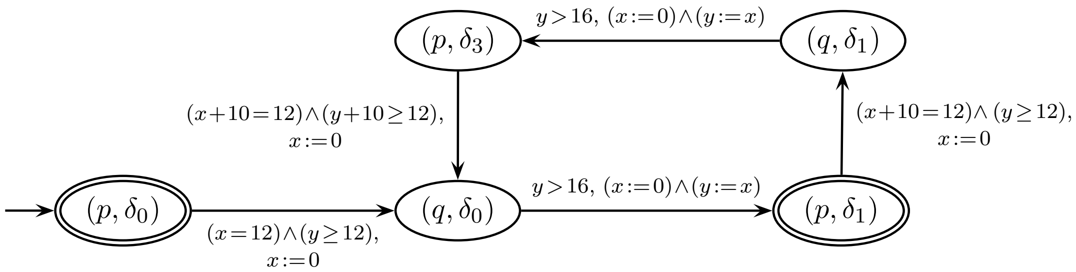

FIG. 5.4 – *Automate temporisé $\mathcal{B}$*

Pour finir, nous allons considérer le modèle qui comporte des mises à jour de type $x := y + c$. La preuve s'effectue en généralisant la construction précédente afin d'ajouter des décalages d'horloges pour les mises à jour $x := y$ aussi.

Lemme 16.3. *Soient X un ensemble fini d'horloges et $\mathcal{U}_{x:=y+c}$ l'ensemble des mises à jour de type $(x := y + c)$ avec $x, y \in X$ et $c \in \mathbb{Q}$. Alors, on a :*

$$Aut(\mathcal{C}_{df},\mathcal{U}_{x:=y+c} \cup \mathcal{U}_{x:=c}) \equiv_s Aut(\mathcal{C}_{df},\mathcal{U}_{x:=y} \cup \mathcal{U}_{x:=c}) \tag{5.6}$$

Démonstration. Nous allons considérer $\mathcal{A}$ un automate temporisé quelconque de la classe $Aut(\mathcal{C}_{df},\mathcal{U}_{x:=y+c} \cup \mathcal{U}_{x:=c})$, puis construire un automate temporisé $\mathcal{B} \in Aut(\mathcal{C}_{df},\mathcal{U}_{x:=y} \cup \mathcal{U}_{x:=c})$, tel que $\mathcal{A} \equiv_s \mathcal{B}$. Le lemme 16.3 en découle naturellement.

Nous allons commencer par dupliquer les états de $\mathcal{A}$. Pour cela, on considère tous les ensembles d'entiers relatifs $\Delta = (\delta_x)_{x\in X} \in \mathbb{Z}^X$ tels que $max_x + 1 \geq \delta_x$ avec $(max_x)_{x\in X}$ l'ensemble des constantes qui sont solutions du système Diophantien donné au chapitre 4 :

$$\begin{aligned}
\forall (x \sim c) \in C,\ & max_x \geq c, \text{ avec } \sim \in \{<,\leq,=,\geq,>\} \\
\forall (x := c) \in U,\ & max_x \geq c \\
\forall (x := y + c) \in U,\ & max_y + c \geq max_x
\end{aligned}$$

Pour chaque état q de $\mathcal{A}$, on construit un état de $\mathcal{B}$ qui sera une copie q_Δ de q. Intuitivement, ces vecteurs Δ, codent le décalage des horloges par rapport à leur valeur courante dans l'automate initial. Alors, dans l'état q_Δ, l'horloge x aura la valeur $v(x) + \delta_x$.

Pour toute transition $q \xrightarrow{g,a,up} q'$ de $\mathcal{A}$, on crée dans $\mathcal{B}$ une transition $q_\Delta \xrightarrow{q_\Delta,a,up_\Delta} q'_{\Delta'}$ pour chaque Δ, avec :

- $g_\Delta = g[x \leftarrow x + \delta_x]$,
- $up_\Delta = up[(x := y + c) \leftarrow (x := y)]$, et on complète la mise à jour up_Δ par des updates du type $x := x$, si x n'est pas remise à jour dans up.
- $\delta'_x = \begin{cases} \min(\delta_y + c, max_x + 1), & \text{si } x := y + c \in up \\ 0, & \text{sinon} \end{cases}$

On peut noter que le nombre des $\Delta = (\delta_x)_{x\in X} \in \mathbb{Z}^X$ n'est pas borné. Nous avons donc, a priori, un nombre infini de copies de chaque état q de $\mathcal{A}$. Cependant, nous allons montrer qu'à partir de l'état initial $(0, \ldots, 0)$ seul un nombre fini de ces copies d'états est accessible.

Supposons par l'absurde que ce ne soit pas le cas. Alors il existe une suite $(\Delta^{(i)})_{i\geq 0}$ avec $\Delta^{(0)} = (0, \ldots, 0)$ et telle que pour tout $i \geq 0$, il existe une suite de transitions $(\rightarrow_i)_{i>0}$ de $\mathcal{B}$ telles que $q_{\Delta^{(0)}} \rightarrow_1 q_{\Delta^{(1)}} \rightarrow_2 \cdots \rightarrow_i q_{\Delta^{(i)}}$, dont les $\Delta^{(i)}$ tendent vers $-\infty$ pour un x donné. Autrement dit, la suite $(\delta_x^{(i)})_{x\geq 0}$ tend vers $-\infty$.

Pour chaque $\rightarrow_i$, on a une mise à jour up_i qui peut s'exprimer sous la forme :

$$up_i ::= \bigwedge_{\substack{x \in X_1,\\ c_x \geq 0}} (x := c_x) \wedge \bigwedge_{\substack{x \in X_2,\\ d_x < 0}} (x := y_x + d_x) \wedge \bigwedge_{\substack{x \in X_3,\\ c_x \geq 0}} (x := y_x + c_x) \tag{5.7}$$

Avec X_1, X_2 et X_3 des sous-ensembles disjoints de X. On définit up_i^- comme l'ensemble des mises à jour de up_i qui sont de la forme $x := y + c$ avec $c < 0$:

$$up_i^- ::= \bigwedge_{x\in X_2} (x := y_x + d_x), \text{ avec } d_x < 0. \tag{5.8}$$

Et on définit la suite $(\Gamma^{(i)})_{i\geq 0}$ telle que :

- $\Gamma^{(0)} = \Delta^{(0)}$,
- $\Gamma^{(i+1)}$ est obtenu à partir de $\Gamma^{(i)}$ en utilisant up_i^-.

On peut alors vérifier que la suite $(\Gamma^{(i)})_{i\geq 0}$ est décroissante non stationnaire car $(\delta_x^{(i)})_{i\geq 0}$ tend vers $-\infty$ pour un x donné. Soit à présent z une horloge telle que $(\Gamma_z^{(i)})_{i\geq 0}$ tende vers $-\infty$. On a deux possibilités :

- Soit, il existe une mise à jour $up_i^- = (z := z + c)$ avec $c < 0$. Comme on sait, par hypothèse, que le système est décidable et que par conséquent, il existe une solution pour le système Diophatien, il devrait être possible de résoudre l'inéquation liée à cette mise à jour. L'inéquation correspondante est $max_z \leq max_z + c$ avec $c < 0$. Or, cette inéquation ne peut avoir de solution. Ce qui nous permet de conclure à une contradiction.
- Soit, on a un cycle de mises à jour up_i^-. En effet, comme l'ensemble des horloges est fini, la seule possibilité pour satisfaire le fait que $\delta_z^{(i)}$ tend vers $-\infty$ est d'avoir un cycle de mises à jour de la forme :
  - $z := z_1 + c_1$, avec $c_1 < 0$.

- $z_1 := z_2 + c_2$, avec $c_2 < 0$.
- ...
- $z_p := z + c_{p+1}$, avec $c_{p+1} < 0$.

Comme l'automate initial est supposé être décidable, les inéquations induites par ces mises à jour devrait avoir une solution, or on a :

- $max_z \leq max_{z_1} + c_1$, avec $c_1 < 0$.
- $max_{z_1} \leq max_{z_2} + c_2$, avec $c_1 < 0$.
- ...
- $max_{z_p} \leq max_z + c_{p+1}$, avec $c_1 < 0$.

On en déduit que $max_z \leq max_z + c$ avec $c < 0$. Or, cette inéquation ne peut avoir de solution. Ce qui nous permet, à nouveau, de conclure à une contradiction.

Comme nous venons de le montrer, supposer qu'il existe une suite de mises à jour qui tend vers $-\infty$ n'est pas valide. On en déduit donc qu'une telle suite n'existe pas si notre système est décidable (*i.e.* s'il vérifie le système d'inéquations que nous avons exhibé dans le chapitre 4).

Ce qui achève la construction de l'automate temporisés $\mathcal{B}$. Il nous reste à présent à définir une relation de bisimulation $\rhd \subset (Q \times \mathbb{T}^X) \times (Q_\Delta \times \mathbb{T}^X)$ entre $\mathcal{A}$ et $\mathcal{B}$, avec Q_Δ l'ensemble infini des états q_Δ.

$$(q, v) \rhd (q_\Delta, v_\Delta) \Leftrightarrow \begin{cases} v \text{ et } v_\Delta + \Delta \text{ sont } \mathcal{R}_\lambda - \text{régions équivalentes} \\ v(x) \leq max_x \Rightarrow v(x) = v_\Delta(x) + \delta_x, \text{ pour tout } x \in X \end{cases} \tag{5.9}$$

Montrons à présent que $\rhd$ est une bisimulation. Pour cela, on suppose que l'on a les relations $(q, v) \rhd (q_\Delta, v_\Delta)$ et $(q, v) \xrightarrow{a} (q', v')$. Cela signifie qu'il existe une transition $q \xrightarrow{g,a,up} q'$ dans $\mathcal{A}$ telle que $v \models g$ et $v' = up(v)$. Or, par construction de $\mathcal{B}$, il existe une transition $q_\Delta \xrightarrow{g_\Delta, a, up_\Delta} q'_{\Delta'}$. Posons $v'_{\Delta'} = up_\Delta(v_\Delta)$ et montrons que $(q', v') \rhd (q'_{\Delta'}, v'_\Delta)$. Deux cas sont à considérer :

- Si x est telle que $(x := c) \in up$, alors la mise à jour $(x := c)$ fait aussi partie de up_Δ, donc $v'_{\Delta'}(x) = c = v'(x)$.
- Si x est telle que $(x := y + c) \in up$, alors la mise à jour $(x := y)$ fait partie de up_Δ, on distingue deux cas :
  - Si $v'(x) \leq max_x$. Nous voulons montrer que $v'(x) = v'_{\Delta'}(x) + \delta'_x$. Comme $(x := y)$ fait partie de up_Δ, on a $v'_{\Delta'}(x) + \delta'_x = v_\Delta(y) + \delta'_z$. On distingue alors deux cas :
    - Soit $\delta'_x \leq max_x$, et on en déduit que $v'_{\Delta'}(x) + \delta'_x = v_\Delta(y) + \delta_y + c$. Or, par hypothèse, on sait que $(q, v) \rhd (q_\Delta, v_\Delta)$ et $v(y) < max_y$ (car $v'(x) = v(y) + c \leq max_x$ et $max_x \leq max_y + c$), on en déduit le résultat voulu : $v'_{\Delta'}(x) + \delta'_x = v(y) + c = v'(x)$.
    - Soit $\delta'_x > max_x$, alors on a $\delta_y + c > max_x$. Or, on sait par ailleurs que $v'(x) = v(y) + c = v_\Delta(y) + \delta_y + c$. On déduit des deux assertions précédentes que $v'(x) > max_x$. Or, cela n'est pas possible puisque nous avions supposé que $v'(x) \leq max_x$.
  - Si $v'(x) > max_x$, on distingue à nouveau deux cas :
    - Si $\delta'_x > max_x$, alors $v'_{\Delta'} + \delta'_x > max_x$.

- Si $\delta'_x \leq max_x$, alors $v'_{\Delta'}(x)+\delta'_x = v_\Delta(y)+\delta_y+c$. Et on distingue deux sous-cas :
  - Si $v_\Delta(y) + \delta_y \leq max_y$, alors $v'_{\Delta'}(x) + \Delta'_x = v(y) + c = v'(x)$. Et on en déduit que $v'_{\Delta'}(x) + \Delta'_x > max_x$.
  - Si $max_y < v_\Delta(y) + \delta_y$, alors comme $max_x \leq max_y + c$, on obtient que $max_x < v'_{\Delta'}(x) + \delta'_x$.

Quelque soit le cas considéré, on obtient bien le résultat escompté. A cela, on ajoute le fait que le pré-ordre sur les parties fractionnaires ne change pas lors d'une de ces transitions, ce qui nous permet de conclure que $(q', v') \rhd (q'_{\Delta'}, v'_{\Delta'})$ est bien une relation de simulation. La preuve pour la réciproque ($\rhd^{-1}$) est identique.

Ceci achève la preuve de la bisimulation entre $\mathcal{A}$ et $\mathcal{B}$. $\square$

Finalement, la preuve du théorème 16 page 86 s'obtient par transitivité de la relation de bisimulation forte en mettant bout à bout les lemmes 16.1, 16.2 et 16.3.

Plus formellement, on a :

$$Aut(\mathcal{C}_{df}, \mathcal{U}_0) \equiv_s Aut(\mathcal{C}_{df}, \mathcal{U}_{x:=y}) \equiv_s Aut(\mathcal{C}_{df}, \mathcal{U}_{x:=y} \cup \mathcal{U}_{x:=c}) \equiv_s Aut(\mathcal{C}_{df}, \mathcal{U}_{x:=y+c} \cup \mathcal{U}_{x:=c}) \tag{5.10}$$

On en déduit, donc, que :

$$Aut(\mathcal{C}_{df}, \mathcal{U}_0) \equiv_s Aut(\mathcal{C}_{df}, \mathcal{U}_{x:=y+c} \cup \mathcal{U}_{x:=c}) \tag{5.11}$$

Or, on sait par définition que :

$$Aut(\mathcal{C}_{df}, \mathcal{U}_{det}) = Aut(\mathcal{C}_{df}, \mathcal{U}_{x:=y+c} \cup \mathcal{U}_{x:=c}) \tag{5.12}$$

Donc, on en déduit, finalement, que :

$$Aut(\mathcal{C}_{df}, \mathcal{U}_0) \equiv_s Aut(\mathcal{C}_{df}, \mathcal{U}_{det}) \tag{5.13}$$

Ce qui achève la preuve du théorème 16. $\square$

Comme nous venons de le prouver avec le théorème 16, l'expressivité du modèle des automates temporisé classiques n'est pas augmentée par l'utilisation de mises à jour déterministes. Cependant, il est aisé de voir que ces mises à jour permettent de réaliser des modèles beaucoup plus compacts et plus compréhensibles (voir chapitre 7). En effet, comme nous l'ont montré les constructions incluses dans les preuves, on doit souvent passer par une explosion exponentielle du nombre des états entre la représentation qui contient les mises à jour déterministes et la représentation classique. Sans pour autant faire disparaître cette explosion, le modèle avec les mises à jour déterministes permet de reporter cette explosion au niveau du graphe des régions et d'automatiser la procédure de raffinement en libérant l'utilisateur de ce travail fastidieux.

Nous allons à présent poursuivre nos investigations en considérant les automates temporisés avec mises à jour non-déterministes. Il s'avère que ces mises à jour assez particulières ajoutent de l'expressivité au modèle classique. Ce qui rend cette extension du modèle non-triviale.

5.3 Pouvoir d'expression des mises à jour non-déterministes

Nous allons maintenant nous attacher à cerner l'expressivité du fragment décidable des automates temporisés avec mises à jour non-déterministes. Nous commençons par rappeler le fragment décidable des automates temporisés avec mises à jour. Puis nous introduirons la notion de forme normale pour un automate temporisé. Nous montrerons ensuite l'inclusion *stricte* de la classe des langages engendrés par des automates temporisés classiques dans la classe des langages engendrés par le fragment décidable des automates temporisés avec mises à jour non-déterministes (sans ε-transition). Finalement, nous montrerons que la classe des langages engendrés par le fragment décidable des automates temporisés avec mises à jour non-déterministes est inclus dans la classe des langages engendrés par les automates temporisés avec transitions silencieuses (ε-transitions).

La classe des automates temporisés avec mises à jour et actions silencieuses est notée $Aut_\varepsilon(C,U)$, avec $C \subset \mathcal{C}_{df}$ un ensemble de contraintes non-diagonales, $U \subset \mathcal{U}_{ndet}$ (voir p.63) un ensemble de mises à jour non-déterministes et cette classe est telle que si $A \in Aut_\varepsilon(C,U)$, alors l'alphabet de A contient une action silencieuse ε.

Rappelons que $\mathcal{U}_{ndet}$ est défini par la grammaire suivante :

$$up = \bigwedge_{x \in Y} up_x, \text{ avec } Y \subseteq X \text{ et } up_x ::= det_x \,|\, inf_x \,|\, sup_x \,|\, int_x, \text{ avec :}$$

- $det_x ::= x := c \,|\, x := y + d,$ avec $c \in \mathbb{N}, d \in \mathbb{Z}$ et $y \in X$
- $inf_x ::= x :\lhd c \,|\, x :\lhd y + d \,|\, inf_x \wedge inf_x,$ avec $\lhd \in \{<, \leq\}, c \in \mathbb{N}, d \in \mathbb{Z}$ et $y \in X$,
- $sup_x ::= x :\rhd c \,|\, x :\rhd y + d \,|\, sup_x \wedge sup_x,$ avec $\rhd \in \{<, \leq\}, c \in \mathbb{N}, d \in \mathbb{Z}$ et $y \in X$,
- $int_x ::= x :\in (c,d) \,|\, x :\in (c, y+d) \,|\, x :\in (y+c, d) \,|\, x :\in (y+c, y+d),$
  avec ')', '(' $\in$ {']', '['}, $c \in \mathbb{N}, d \in \mathbb{Z}$ et $y \in X$.

Une mise à jour up est donc la conjonction de ses mises à jour locales (det_x, inf_x, sup_x, int_x). Et on note une mise à jour générale $up = (up_x)_{x \in Y}$, avec $Y \subseteq X$ et up_x une mise à jour locale à x, pour tout $x \in Y$. Comme noté dans la remarque 4 page 37, nous pouvons, ici aussi, nous restreindre aux seules constantes entières dans les gardes et les mises à jour et étendre le résultat aux constantes rationnelles.

Afin de faciliter les démonstrations, nous allons définir une forme normale pour les automates temporisés avec mises à jour et sans contraintes diagonales. La forme normale n'est autre qu'une étape intermédiaire entre l'automate temporisé et son graphe des régions.

Définition 20 (Forme normale). *Soient $\mathcal{A}$ un automate temporisé sans garde diagonales donné, et $(max_x)_{x \in X}$ l'ensemble des constantes maximales de celui-ci. On définit alors $\mathcal{I}_x$ comme l'ensemble des* intervalles élémentaires *de $\mathcal{A}$:*

$$\mathcal{I}_x ::= \{]d, d+1[\,|\, 0 \leq d < max_x\} \cup \{\{d\} \mid 0 \leq d \leq max_x\} \cup \{]max_x, +\infty[\} \tag{5.14}$$

On définit aussi l'ensemble des intervalles élémentaires tels que $I_x \neq \{d\}$:

$$\mathcal{J}_x ::= \{]d, d+1[\,|\, 0 \leq d < max_x\} \cup \{]max_x, +\infty[\} \tag{5.15}$$

On appelle garde élémentaire*, les gardes de la forme suivante :*

$$g = \bigwedge_{x \in X} (x \in I_x), \text{ avec } \forall x \in X,\ I_x \in \mathcal{I}_x \tag{5.16}$$

Une mise à jour up_x est dites élémentaire *, si elle s'écrit suivant la grammaire suivante :*

$$\begin{aligned} up_x ::= x :\in I_x \mid (x := y + c) \wedge x :\in J_x \\ \mid \bigwedge_{y \in Y} (x :< y + c) \wedge (x :\in J_x) \\ \mid \bigwedge_{y \in Y} (x :> y + c) \wedge (x :\in J_x) \end{aligned} \tag{5.17}$$

Avec $I_x \in \mathcal{I}_x$, $J_x \in \mathcal{J}_x$ et $Y \subseteq X$.

Enfin, un automate temporisé $\mathcal{A} \in Aut_\varepsilon(\mathcal{C}_{df}, \mathcal{U})$ est dit en forme normale *si toutes les transitions (p, φ, a, up, q) sont telles que :*

- *φ est une garde élémentaire,*
- *$up = \wedge_{x \in Y} up_x$ avec $\forall x \in Y$, up_x est une mise à jour élémentaire.*

On peut remarquer qu'il est possible que plusieurs gardes élémentaires soient nécessaires pour simuler une garde non-élémentaire. Mais, il suffit alors de considérer plusieurs transitions.

Pour le cas précis des automates temporisés décidables, n'importe quel automate temporisé peut être exprimé sous une forme normale.

Proposition 4. *Pour tout $\mathcal{A}$ appartenant au fragment décidable de $Aut_\varepsilon(\mathcal{C}_{df}, \mathcal{U})$, il existe un $\mathcal{B} \in Aut_\varepsilon(\mathcal{C}_{df}, \mathcal{U})$ en forme normale tel que $\mathcal{A} \equiv_s \mathcal{B}$.*

Il est cependant important de signaler que la transformation d'un automate temporisé classique en un automate temporisé en forme normale, se fait au prix d'une augmentation importante du nombre des transitions.

Exemple 14. *Nous présentons sur les figures 5.5 et 5.6, un exemple d'automate temporisé classique noté $\mathcal{A}$ (fig. 5.5) et sa forme normale notée $\mathcal{B}$ (fig. 5.6). L'ensemble des constantes maximales est donné par $max_x = max_y = 1$. Et on a $\mathcal{I}_x = \mathcal{I}_y = \{\{0\},]0, 1[, \{1\},]1, +\infty[\}$.*

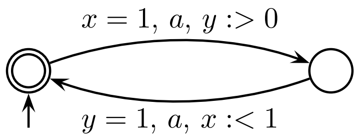

FIG. 5.5 – *Automate temporisé $\mathcal{A}$*

Nous allons maintenant nous intéresser à la preuve d'inclusion stricte de la classe des langages engendrés par les automates temporisés classiques dans la classe de langages engendrés par les automates temporisés avec mises à jour non-déterministes. Cette preuve a été publiée dans [BDFP00b]. Elle repose sur un résultat d'inclusion stricte qui établit qu'aucun automate temporisé classique sans action silencieuse ne peut reconnaître le langage engendré par $\mathcal{A}_\varepsilon$ (fig. 5.7 page suivante) [BDGP98]. La preuve consiste à construire un automate temporisé avec mises à jour non-déterministes qui reconnaît le même langage que $\mathcal{A}_\varepsilon$.

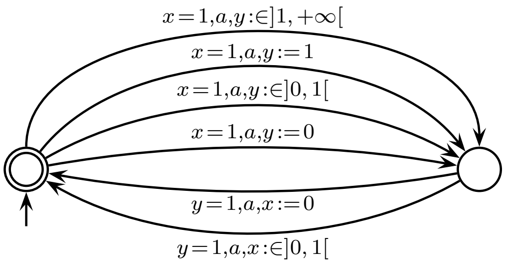

FIG. 5.6 – *Automate temporisé $\mathcal{B}$ (forme normale de $\mathcal{A}$)*

Théorème 17. *Soient X un ensemble fini d'horloges, $Aut(\mathcal{C}_{df},\mathcal{U}_0)$ l'ensemble des automates temporisés classiques sans gardes diagonales et $Aut(\mathcal{C}_{df},\mathcal{U}_{ndet})$ le fragment décidable de l'ensemble des automates temporisés avec mises à jour non déterministes. Alors on a :*

$$L(Aut(\mathcal{C}_{df},\mathcal{U}_0)) \subsetneq L(Aut(\mathcal{C}_{df},\mathcal{U}_{ndet})) \tag{5.18}$$

Démonstration. Il a été prouvé que le langage engendré par l'automate temporisé $\mathcal{A}_\varepsilon$ qui appartient à $Aut_\varepsilon(\mathcal{C}_{df},\mathcal{U}_0)$ (fig. 5.7), ne peut être reconnu par un automate temporisé classique [BDGP98]. $\mathcal{A}_\varepsilon$ reconnaît un langage dans lequel des a peuvent être tirés à chaque unité de temps, sauf si un b a été tiré durant l'unité de temps précédente. Un automate temporisé classique peut reconnaître un tel langage seulement si le nombre de b est borné. Si on considère un nombre quelconque de b, alors l'automate temporisé classique a besoin d'un nombre infini d'horloges, ce qui est impossible avec un automate temporisé avec transitions silencieuses.

La preuve du théorème 17 consiste à construire un automate temporisé $\mathcal{A} \in Aut(\mathcal{C}_{df},\mathcal{U}'_{ndet})$ qui a le même langage que l'automate temporisé $\mathcal{A}_\varepsilon$.

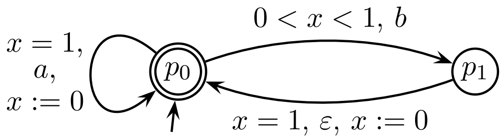

FIG. 5.7 – *Automate temporisé avec ε-transitions* ($\mathcal{A}_\varepsilon$)

La construction de $\mathcal{A}$ se fait en deux étapes. On commence par construire l'automate $\mathcal{A}'$ (fig. 5.8) dont le langage est le même que celui de $\mathcal{A}_\varepsilon$ mais qui n'appartient pas à $Aut(\mathcal{C}_{df},\mathcal{U}_{ndet})$ à cause de la mise à jour $x{:=}x-1$.

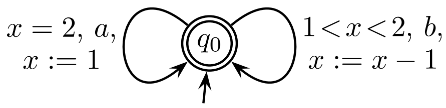

FIG. 5.8 – *Automate temporisé avec mises à jour* ($\mathcal{A}'$)

Puis, à partir de $\mathcal{A}'$, on construit l'automate $\mathcal{A}$ (fig. 5.9). On montre ensuite qu'il existe une relation de bisimulation qui lie ces deux automates. Celle que nous allons exhiber est donnée dans [Bou02a] p. 69–70.

Afin d'exhiber cette relation de bisimulation, on ajoute à l'automate $\mathcal{A}'$ un état puits q_1 relié à q_0 par une unique transition dont le label est '$0 < x < 1,\ b$'. Étant donné que q_1 est un

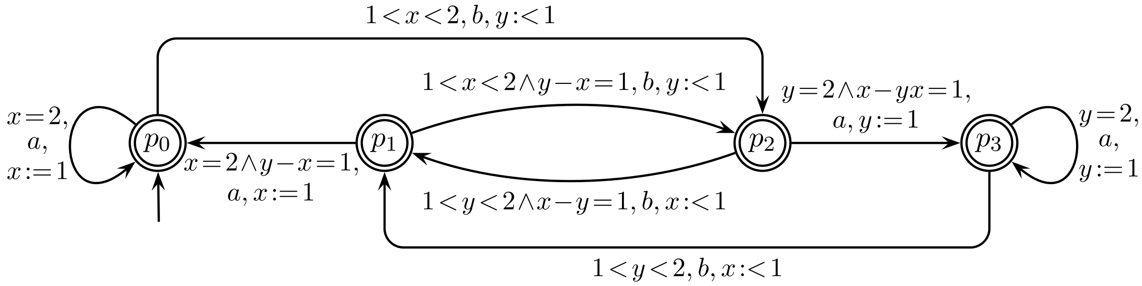

FIG. 5.9 – *Automate temporisé avec mises à jour* ($\mathcal{A}$)

état puits et qu'il n'est pas un état répété de l'automate, l'addition de cet état puits n'ajoute rien au pouvoir expressif de $\mathcal{A}'$. Le résultat de cette construction est montré sur la figure 5.10.

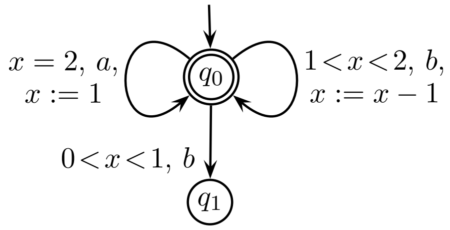

FIG. 5.10 – *Automate temporisé avec mises à jour* ($\mathcal{B}'$)

La relation de bisimilarité est donnée par :

$$\begin{aligned}\rhd = \;& \{(q_0,x),(p_0,(x+1,x+1)) \mid 0 \le x \le 1\} \cup \{(q_0,x),(p_3,(x+1,x+1)) \mid 0 \le x \le 1\} \\ & \cup \{(q_0,x),(p_2,(x+1,x)) \mid 0 < x \le 1\} \cup \{(q_0,x),(p_1,(x,x+1)) \mid 0 < x \le 1\} \\ & \cup \{(q_1,x),(p_2,(y,x)) \mid x > 0 \text{ et } y \neq x+1\} \cup \{(q_1,x),(p_1,(x,y)) \mid x > 0 \text{ et } y \neq x+1\}\end{aligned}$$

Les rôles de p_0 et p_3 sont symétriques, de même que pour p_1 et p_2. On vérifie aisément les propriétés d'*Initialisation* et de *Propagation* (voir p. 84) dans les deux sens, ce qui nous permet de conclure que les deux automates sont fortement bisimilaires. L'équivalence de langage entre $\mathcal{A}_\varepsilon$ et $\mathcal{A}$ en découle naturellement. □

Nous allons, à présent nous concentrer sur la preuve d'inclusion entre la classe des langages engendrés par les automates temporisés avec mises à jour non-déterministes ($Aut(\mathcal{C}_{df}, \mathcal{U}_{ndet})$) et la classe des langages engendrés par les automates temporisés classiques ($Aut(\mathcal{C}_{df}, \mathcal{U}_0)$). À cette fin nous allons montrer qu'il est toujours possible de transformer un automate temporisé avec mises à jour non-déterministe en un automate temporisé sans mises à jour non-déterministes, mais contenant dans certains cas des actions silencieuses. Nous commencerons par quelques exemples introductifs puis nous enchaînerons par l'énoncé du théorème puis la preuve proprement dite.

Exemples 15. *Le premier exemple que nous allons considérer ici, est une mise à jour non-déterministe du type* $x :< 1$. *L'automate* $\mathcal{A}$ *(fig. 5.11) reconnaît le langage :*

$$L(\mathcal{A}) = \{(a,\tau)(b,\tau') \mid 0 \le \tau < 2,\ 0 < \tau' - \tau < 1\} \tag{5.19}$$

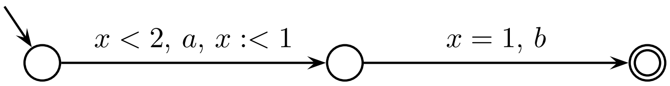

FIG. 5.11 – *Automate temporisé ($\mathcal{A}$) avec une mise à jour non-déterministe $x :< 1$.*

Il est possible de simuler l'automate $\mathcal{A}$ (fig. 5.11) par l'automate $\mathcal{B}$ (fig. 5.12). L'idée est de remplacer la mise à jour non déterministe par une ε-transition. Pour ce faire, nous introduisons une nouvelle horloge z_x qui aura pour fonction de borner le temps pendant lequel x peut rester non déterminée. En effet, lors de la mise à jour $x :< 1$, le cas limite consiste à remettre x à zéro. Dès lors x ne dispose ***au plus*** *que d'une unité de temps pour atteindre 1. La garde $z_x < 1$ permet de rendre compte de cette contrainte sans préjuger de la valeur de x. À un instant non déterministe, mais avant $z_x = 1$, x est mise à jour à 1 de façon silencieuse et l'on reprend l'exécution normale de l'automate.*

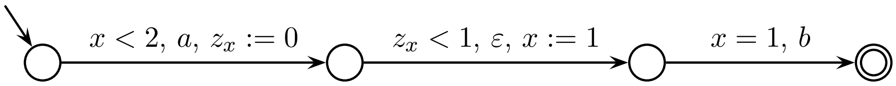

FIG. 5.12 – *Automate temporisé $\mathcal{B}$, sans mises à jour non-déterministes, simulant $\mathcal{A}$.*

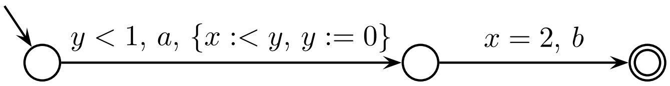

FIG. 5.13 – *Automate temporisé $\mathcal{A}$ avec mises à jour non-déterministe $x :< y$*

Le second exemple introduit une mise à jour non-déterministe de type $x :< y$. L'automate $\mathcal{A}$ (fig. 5.13) reconnaît le langage :

$$L(\mathcal{A}) = \{(a,\tau)(b,\tau') \mid (\tau < 1) \wedge (\tau' > 2)\} \tag{5.20}$$

Une solution naïve consisterait à reprendre la même idée que dans l'exemple précédent. À savoir, introduire une horloge z_x qui servirait de borne avant de prendre la transition. L'automate $\mathcal{B}$ représente cette solution (fig. 5.14).

Malheureusement, cette solution est fausse. L'automate $\mathcal{B}$ reconnaît un langage plus riche que celui de $\mathcal{A}$ (par exemple, le mot $(a, 0.5)\,(b, 1.8)$).

Le problème qui se pose ici est qu'aucune contrainte n'empêche x de prendre la valeur 1 avant *que y ne parvienne à 1, si elle n'a pas été mise à jour. Or la mise à jour non-déterministe impose bien que $x :< y$ (x prend une valeur inférieure à y).*

La solution, représentée par l'automate $\mathcal{B}'$ (fig. 5.15), consiste à encadrer x par une valeur minimale ($z_x := 0$) et une valeur maximale ($w_{x,y} := y$). Les contraintes sont donc posées de telle façon que x ne peut être mise à jour à la valeur 1, que si $z_x = 1$ ***avant*** *et $w_{x,y} = 1$* ***après****. Ce qui correspond exactement aux conditions réunies dans l'automate $\mathcal{A}$.*

Ce deuxième exemple conclut notre introduction au théorème 18. Comme nous venons d'en donner l'intuition, il est possible de remplacer des mises à jour $x :< c$ et $x :< y$ par des transitions silencieuses. Il faut cependant s'assurer que certaines conditions sont assurées ce qui nécessite l'utilisation d'horloges supplémentaires telles que z_x et $w_{x,y}$ qui représentent,

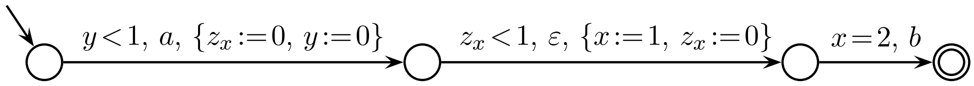

FIG. 5.14 – *Automate temporisé $\mathcal{B}$, sans mises à jour non-déterministes,* 'simulant' $(\mathcal{A})$.

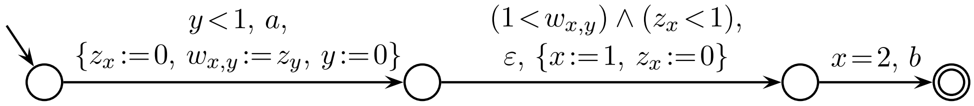

FIG. 5.15 – *Automate temporisé $\mathcal{B}'$, sans mises à jour non-déterministes, simulant $(\mathcal{A})$.*

respectivement, la borne inférieure et supérieure de la valeur que x peut prendre lorsqu'elle est mise à jour de façon non-déterministe. Comme nous allons le montrer plus formellement par la suite, cet encadrement est suffisant à assurer l'équivalence de langage entre l'automate temporisé avec mises à jour non-déterministes et son équivalent avec ε-transitions.

Théorème 18. *Soit X un ensemble fini d'horloges, alors on a :*

$$Aut(\mathcal{C}_{df},\mathcal{U}_{ndet}) \subseteq Aut_{\varepsilon}(\mathcal{C}_{df},\mathcal{U}_0) \tag{5.21}$$

Démonstration. Comme pour les lemmes 16.1 page 86, 16.2 page 88 et 16.3 page 90, la preuve consiste à montrer que pour un automate temporisé quelconque $\mathcal{A} \in Aut_{\varepsilon}(\mathcal{C}_{df},\mathcal{U}_{ndet})$, on peut construire un automate temporisé $\mathcal{B} \in Aut_{\varepsilon}(\mathcal{C}_{df},\mathcal{U}_{det})$ tel que $\mathcal{A} \equiv_w \mathcal{B}$. Grâce à la proposition 4 page 95, on ne considère ici que des automates de $Aut_{\varepsilon}(\mathcal{C}_{df},\mathcal{U}_{ndet})$ qui sont en forme normale.

Nous allons commencer par introduire la notion d'horloge *fixe* et d'horloge *flottante* que nous utiliserons dans la preuve. Puis, nous expliquerons de manière informelle la construction de l'automate $\mathcal{B}$. Enfin, nous détaillerons sa construction.

Une horloge x est dite *fixée* si on se trouve dans l'un des cas suivants :

- Elle n'a pas encore été mise à jour,
- Sa dernière mise à jour était de la forme $x := c$,
- Sa dernière mise à jour était de la forme $(x := y + c \wedge x :\in I'_x)$ où l'horloge y est elle même fixée.

Une horloge qui n'est pas fixée est dite *flottante.*

De façon similaire aux démonstrations précédentes, nous allons construire des copies de l'automate original $\mathcal{A}$ en leur ajoutant des horloges et en modifiant les transitions. Mais nous ajoutons aussi des ε-transitions qui permettent de passer d'une copie à l'autre.

Horloges supplémentaires Nous énumérons ici les horloges que nous ajoutons au système :

- Pour chaque horloge x de X, on définit une horloge z_x qui représentera la borne inférieure de la partie fractionnaire de x. On note $\mathcal{Z}$ l'ensemble de toutes les horloges de ce type.
- Pour chaque couple (x, y) de X^2, on définit également deux horloges $w_{x,y}$ et $w'_{x,y}$ qui serviront à comparer les parties fractionnaires de x et de y. On note $\mathcal{W}$ l'ensemble de toutes les horloges de ce type.
- Enfin, soit $\mathcal{X}$ l'ensemble de ces $|X| + |X^2|$ horloges supplémentaires.

Duplication de l'automate original Nous allons dupliquer l'automate original qui, par hypothèse, est sous forme normale (voir définition 20). On peut noter que nous n'avons besoin que de suivre les horloges flottantes, les horloges fixes ne nécessitent pas un traitement si fin pour être connues. Notre méthode consiste donc à créer une copie pour chaque sous-ensemble Y de X et pour chaque ordre partiel $\succ$ sur les horloges de Y, en modifiant les gardes et les mises à jour en conséquence. Le passage entre les copies se fait lorsqu'une horloge flottante devient fixe ou qu'une horloge quelconque est mise à jour de façon non-déterministe. Les transitions inter-copies sont, évidemment, des transitions silencieuses.

- Considérons Y un sous-ensemble de X, correspondant aux horloges flottantes de X. Et un ordre partiel $\succ$ sur Y, représentant les positions relatives des parties fractionnaires des horloges de Y,
- Pour chaque horloge $y \in Y$, soit I_y un intervalle de la forme $]d, d+1[$ avec $0 \leq d < max_y$ auquel la valeur de y va appartenir,
- Enfin, soit $W \subset \mathcal{W}$.

Pour chaque triplet $\tau = ((I_y)_{y \in Y}, \succ, W)$, on construit alors une copie $\mathcal{A}_\tau$ de l'automate $\mathcal{A}$, dans laquelle est ajoutée à la garde de chaque transition, la contrainte :

$$\bigwedge_{y \in Y} (y \in I_y) \wedge \bigwedge_{x \in X} (z_x < 1) \tag{5.22}$$

Il est à noter qu'un certain nombre de gardes de l'automate ainsi construit peuvent être équivalentes à `false`. Ces transitions peuvent être effacées. D'une manière équivalente, un certain nombre de gardes peuvent être évaluée à `true`, dans ce cas on enlève les conditions sur la transition.

Horloges fixées Lorsque la partie fractionnaire d'une horloge fixée atteint la valeur 1, l'exécution ne doit pas changer de copie d'automate. Afin de satisfaire cette contrainte, pour chaque copie $\mathcal{A}_\tau$ avec $\tau = ((I_y)_{y \in Y}, \succ, W)$ on ajoute sur chaque état et pour chaque horloge $x \in X \setminus Y$ une boucle sur cet état, étiquetée par :

$$z_x = 1,\ \varepsilon, z_x := 0 \tag{5.23}$$

Horloges flottantes Comme dans les exemples représentés sur les figures 5.12 et 5.15, il est possible de simuler les horloges flottantes en utilisant les ε-transitions. Ces transitions silencieuses permettent à une ou plusieurs horloges flottantes de redevenir fixes à un instant non-déterministe. Comme les copies d'automate sont indexées par l'ensemble des horloges flottantes qu'elles contiennent, les transitions silencieuses vont d'une copie d'automate à une autre (en tenant compte de la modification de l'ensemble des horloges flottantes qu'elles induisent). Cependant, seules les horloges qui sont maximales dans le pré-ordre $\succ$ peuvent être concernées par ce type de transition. Car parmi les horloges flottantes, ce sont elles qui deviendront fixes en premier lors de l'écoulement du temps.

Plus formellement, soit $\mathcal{A}_\tau$ avec $\tau = ((I_y)_{y \in Y}, \succ, W)$ et soit M l'ensemble des éléments maximaux de $\succ$. On relie chaque état q de $\mathcal{A}_\tau$ à l'état équivalent q' de la copie $\mathcal{A}_{\tau'}$ où $\tau' = ((I_y)_{y \in Y'}, \succ', W')$ avec :

- $Y' = Y \setminus M$,
- $\succ' = \succ \cap (Y' \times Y')$,

– $W' = W \setminus \{w_{x,y}, w'_{x,y} \mid x \in M,\ y \in X\}$.

Cette ε-transition est étiquetée par la contrainte :

$$\bigwedge_{x \in M, w_{x,y} \in W} (w_{x,y} \geq 1) \wedge \bigwedge_{x \in M, w'_{x,y} \in W} (w'_{x,y} < 1) \wedge \bigwedge_{y \in Y} (z_y < 1) \tag{5.24}$$

Et la mises à jour :

$$\bigwedge_{y \in M} (y := sup(I_y)) \tag{5.25}$$

Avec $sup(I_y)$ qui représente la borne supérieure de I_y, c'est à dire d, si $I_y =]d, d+1[$.

La présence d'une horloge $w_{x,y}$ (resp. $w'_{x,y}$) indique qu'une mise à jour de la forme $x :< y+c$ (resp. $x :> y+c$) a été effectuée précédemment. La contrainte $w_{x,y} \geq 1$ (resp. $w'_{x,y} < 1$) permet de ne pas perdre l'information de cette mise à jour.

Transitions Il nous reste à définir les transitions visibles du système. Pour cela on considère chaque copie $\mathcal{A}_\tau$, avec $\tau = ((I_y)_{y \in Y}, \succ, W)$, et, dans cette copie, chaque transition $t = (p, g, a, up, q)$, toutes directement héritées de l'automate initial $\mathcal{A}$, mais avec p et q des états de $\mathcal{A}_\tau$. Nous allons substituer à t, une transition $t' = (p, g', a, up', q')$, avec q' la copie de l'état q dans $\mathcal{A}_{(I'_y)_{y \in Y'}, \succ', W'}$, et avec g', up', Y', $(I'_y)_{y \in Y'}$ et $\succ'$ construit inductivement en examinant les mises à jours up_x les unes après les autres (l'ordre n'est pas important). Cette transformation a pour but de remplacer les mises à jour non déterministes par des mises à jour déterministes. On débute l'induction avec :

– $Y' = Y$,
– $I'_y = I_y$ pour tout $y \in Y'$,
– $\succ' = \succ$,
– $W' = W$,
– $g' = g$,
– $up' = \emptyset$.

Puis, pour une mise à jour simple up_x fixée, on fait les transformations suivantes :

1. Si $up_x = (x := c)$, alors x devient fixe :
   – On retire x des horloges flottantes : $Y' = Y \setminus \{x\}$,
   – On retire I_x des intervalles $(I_y)_{y \in Y}$: $I'_y = I_y$ pour tout $y \in Y'$,
   – On retire x du pré-ordre : $\succ' = \succ \cap (Y' \times Y')$,
   – On retire les contraintes diagonales sur x : $W' = W \setminus \{w_{x,y}, w'_{x,y} \in \mathcal{W} \mid y \in X\}$,
   – On ajoute la mise à jour $x := c$ à la mise à jour globale et on remet z_x à zéro : $up' = up \wedge (x := c) \wedge (z_x := 0)$.
2. Si $up_x = x :\in I'_x$, alors on distingue deux cas :
   (a) Si $I'_x =]max_x, +\infty[$, alors on a :
      – On retire x des horloges flottantes : $Y' = Y \setminus \{x\}$,
      – On retire x du pré-ordre : $\succ' = \succ \cap (Y' \times Y')$,
      – On retire les contraintes diagonales sur x : $W' = W \setminus \{w_{x,y}, w'_{x,y} \in \mathcal{W} \mid y \in X\}$,

  - On ajoute la mise à jour $x := max_x + 1$ à la mise à jour globale et on remet z_x à zéro : $up' = up \wedge (x := max_x + 1) \wedge (z_x := 0)$.

(b) Si $I'_x =]c, c+1[$, alors on a :
  - On ajoute x aux horloges flottantes : $Y' = Y \cup \{x\}$,
  - On considère n'importe quel pré-ordre $\succ'$ tel que :
    $\succ' \cap (Y \setminus \{x\} \times Y \setminus \{x\}) = \succ \cap (Y \setminus \{x\} \times Y \setminus \{x\})$,
  - On retire les contraintes diagonales sur x : $W' = W \setminus \{w_{x,y}, w'_{x,y} \in \mathcal{W} \mid y \in X\}$,
  - On remet z_x à zéro : $up' = up \wedge (z_x := 0)$.

3. Si $up_x = \left(\bigwedge_{y \in Y} x := y + c\right) \wedge x :\in I'_x$, alors on a :
   (a) Si $y \notin Y$, alors on a :
     - On retire x des horloges flottantes : $Y' = Y \setminus \{x\}$,
     - On retire x du pré-ordre : $\succ' = \succ \cap (Y' \times Y')$,
     - On retire les contraintes diagonales sur x : $W' = W \setminus \{w_{x,y}, w'_{x,y} \in \mathcal{W} \mid y \in X\}$,
     - On ajoute $x := y + c$ à la mise à jour globale et on aligne z_x sur z_y :
       $up' = up \wedge (x := y + c) \wedge (z_x := z_y)$.
   (b) Si $y \in Y$, alors on a :
     i. Soit I'_x est borné et on a :
       - On ajoute x aux horloges flottantes : $Y' = Y \cup \{x\}$,
       - On ajoute une relation d'équivalence entre x et y dans le pré-ordre :
         $\succ' = (\succ \cap (Y \setminus \{x\} \times Y \setminus \{x\})) \cup \{x \succ y\} \cup \{y \succ x\}$,
       - On retire les contraintes diagonales sur x :
         $W' = W \setminus \{w_{x,y}, w'_{x,y} \in \mathcal{W} \mid y \in X\}$,
       - On aligne z_x sur z_y : $up' = up \wedge (x := y + c) \wedge (z_x := z_y)$.
     ii. Soit $I'_x =]max_x, +\infty[$ et on a :
       - On retire x des horloges flottantes : $Y' = Y \setminus \{x\}$,
       - On retire x du pré-ordre : $\succ' = \succ \cap (Y' \times Y')$,
       - On retire les contraintes diagonales sur x :
         $W' = W \setminus \{w_{x,y}, w'_{x,y} \in \mathcal{W} \mid y \in X\}$,
       - On ajoute la mise à jour $x := max_x + 1$ à la mise à jour globale et on aligne z_x sur z_y : $up' = up \wedge (x := max_x + 1) \wedge (z_x := z_y)$.

4. Si $up_x = \left(\bigwedge_{y \in H} x :< y + c\right) \wedge x :\in I'_x$, alors on pose $H_{fixed} = H \setminus Y$ et $H_{float} = H \cap Y$. Et on a deux cas :
   (a) Soit I'_x est borné et on a :
     - On ajoute x aux horloges flottantes : $Y' = Y \cup \{x\}$,
     - Si $y \in H_{float}$, on ajoute une relation entre x et y dans le pré-ordre :
       $\succ' = (\succ \cap (Y \setminus \{x\} \times Y \setminus \{x\})) \cup \{x \not\succ y\} \cup \{y \succ x\}$,
     - On ajoute les contraintes diagonales sur x et y :
       $W' = W \setminus \{w_{x,y}, w'_{x,y} \in \mathcal{W} \mid y \in X\} \cup \{w_{x,y} \mid y \in H_{fixed}\}$,

- On aligne $w_{x,y}$ sur z_y pour tous les $y \in H_{fixed}$ et on met z_x à zéro : $up' = up \wedge_{y \in H_{fixed}} (w_{x,y} := z_y) \wedge (z_x := 0)$.

(b) Soit $I'_x =]max_x, +\infty[$ et on a :
- On retire x des horloges flottantes : $Y' = Y \setminus \{x\}$,
- On retire x du pré-ordre : $\succ' = \succ \cap (Y' \times Y')$,
- On retire les contraintes diagonales sur x : $W' = W \setminus \{w_{x,y}, w'_{x,y} \in \mathcal{W} \mid y \in X\}$,
- On ajoute la mise à jour $x := max_x + 1$ à la mise à jour globale et on aligne z_x sur z_y : $up' = up \wedge (x := max_x + 1) \wedge (z_x := z_y)$.

5. Si $up_x = \left(\bigwedge_{y \in H} x :> y + c\right) \wedge x :\in I'_x$, alors on pose $H_{fixed} = H \setminus Y$ et $H_{float} = H \cap Y$. Et on a deux cas :

(a) Soit I'_x est borné et on a :
- On ajoute x aux horloges flottantes : $Y' = Y \cup \{x\}$,
- Si $y \in H_{float}$, on ajoute une relation entre x et y dans le pré-ordre : $\succ' = (\succ \cap (Y \setminus \{x\} \times Y \setminus \{x\})) \cup \{x \succ y\} \cup \{y \not\succ x\}$,
- On ajoute les contraintes diagonales sur x et y : $W' = W \setminus \{w_{x,y}, w'_{x,y} \in \mathcal{W} \mid y \in X\} \cup \{w'_{x,y} \mid y \in H_{fixed}\}$,
- On aligne $w_{x,y}$ sur z_y pour tous les $y \in H_{fixed}$ et on met z_x à zéro : $up' = up \wedge_{y \in H_{fixed}} (w'_{x,y} := z_y) \wedge (z_x := 0)$.

(b) Soit $I'_x =]max_x, +\infty[$ et on a :
- On retire x des horloges flottantes : $Y' = Y \setminus \{x\}$,
- On retire x du pré-ordre : $\succ' = \succ \cap (Y' \times Y')$,
- On retire les contraintes diagonales sur x : $W' = W \setminus \{w_{x,y}, w'_{x,y} \in \mathcal{W} \mid y \in X\}$,
- On ajoute la mise à jour $x := max_x + 1$ à la mise à jour globale et on aligne z_x sur z_y : $up' = up \wedge (x := max_x + 1) \wedge (z_x := z_y)$.

Il reste à montrer que l'automate que nous venons de décrire simule faiblement (*i.e.* aux ε-transitions près) l'automate initial et qu'il reconnaît le même langage. Pour ce faire, nous allons proposer une relation de simulation $\rhd$ entre les deux automates.

L'idée de cette simulation est qu'un état de l'automate original va être en relation avec l'ensemble de toutes ses copies dans l'automate que nous avons construit. On remarque au passage que les états étendus de $\mathcal{A}$ sont dans $Q \times \mathbb{T}^X$ et ceux de $\mathcal{B}$ sont dans $Q' \times \mathbb{T}^{X'}$, avec $Q' = \{q_\tau \mid q \in Q \text{ et } \tau \in ((\mathcal{I}_x \cup \{\emptyset\})^X \times \mathcal{P}(X^2) \times W)\}$ et $X' = X \cup \{z_x \mid x \in X\} \cup W$ (avec $W = \{w_{x,y}, w'_{x,y} \mid (x,y) \in X^2\}$). On définit la relation $\rhd$ par :

$$\rhd = \left\{ ((q_\tau, v'), (q, v)) \;\middle|\; \begin{array}{l} \forall y \in Y, v(y) \in I_y \text{ et } 0 \leq v'(z_y) \leq 1, \\ \forall y \in X \setminus Y, \text{ soit } v(y) = v'(y), \text{ soit } (v(y) > c_y \text{ et } v'(y) > c_y), \\ y_1 \succ y_2 \Rightarrow frac(v(y_1)) \geq frac(v(y_2)), \\ w_{x,y} \in W \Rightarrow v'(w_{x,y}) > frac(v(x)) \text{ et} \\ w'_{x,y} \in W \Rightarrow frac(v(x)) > v'(w'_{x,y}). \end{array} \right\}$$

Montrer que $\rhd$ est une simulation ne contient pas de difficulté particulière, mais est extrêmement fastidieux. La reconnaissance du langage par l'automate que nous avons défini est vrai par construction. De plus, de par la définition de $\rhd$, on vérifie les conditions nécessaires à la décidabilité d'un automate temporisé avec mises à jour déterministes. Finalement, comme l'automate construit ne contient que des mises à jour déterministes et qu'il remplit les conditions que nous avons établi au chapitre 4, on peut appliquer le théorème 16. Ce qui nous permet de conclure la preuve du théorème 18 page 99. □

Conclusion

Le principal résultat de ce chapitre est de montrer que les automates temporisés avec mise à jour (non-déterministes) sont *strictement* plus expressifs que les automates temporisés classiques. Nous conjecturons que l'inclusion des automates temporisés avec mises à jour dans les automates temporisés avec mises à jour silencieuses est *stricte* car il semble "*difficile*" de simuler des actions silencieuses avec des mises à jour non-déterministes.

Conjecture 1. *Soit X un ensemble fini d'horloges, alors on a :*

$$Aut(\mathcal{C}_{df}, \mathcal{U}_{ndet}) \subsetneq Aut_\varepsilon(\mathcal{C}_{df}, \mathcal{U}_0) \tag{5.26}$$

Enfin, ces résultats permettent de compléter le tableau que nous avons commencé au chapitre 2 page 49. On notera TA (*Timed Automata*) les automates temporisés classiques ($Aut(\mathcal{C}_{df}, \mathcal{U}_0)$) et TA_ε (*Timed Automata with silent action*) les automates temporisés classiques avec ε-transitions ($Aut_\varepsilon(\mathcal{C}_{df}, \mathcal{U}_0)$). Le tableau 5.1 donne la complexité du problème du vide puis, si le vide est décidable, la classe d'automates temporisés dans lequel est inclus le fragment considéré.

<table>
<tr><th></th><th>Mises à jours</th><th>Contraintes non diagonales</th><th>Contraintes générales</th></tr>
<tr><td rowspan="5">Déterministes</td><td>$x := 0$</td><td rowspan="3">Pspace/TA</td><td rowspan="2">Pspace/TA</td></tr>
<tr><td>$x := c,\ c \in \mathbb{Q}_+$</td></tr>
<tr><td>$x := y + c,\ c \in \mathbb{Q}_+$</td><td rowspan="3">Indécidable</td></tr>
<tr><td>$x := y - 1$</td><td rowspan="2">Indécidable</td></tr>
<tr><td>$x := y + c,\ c \in \mathbb{Q}$</td></tr>
<tr><td rowspan="8">non-déterministes</td><td>$x :< c,\ c \in \mathbb{Q}_+$</td><td rowspan="7">Pspace/TA_ε</td><td>Pspace/TA_ε</td></tr>
<tr><td>$x :> c,\ c \in \mathbb{Q}_+$</td><td rowspan="7">Indécidable</td></tr>
<tr><td>$x :< y$</td></tr>
<tr><td>$x :> y$</td></tr>
<tr><td>$x :< y + c,\ c \in \mathbb{Q}_+$</td></tr>
<tr><td>$x :> y + c,\ c \in \mathbb{Q}_+$</td></tr>
<tr><td>$y + c <: x :< y + d,$
$c, d \in \mathbb{Q}_+$</td></tr>
<tr><td>$y + c <: x :< z + d,$
$c, d \in \mathbb{Q}_+,\ y \neq z$</td><td>Indécidable</td></tr>
</table>

TAB. 5.1 – *Décidabilité des différentes sous-classes d'automates temporisés.*

On peut à présent introduire une nouvelle classe d'automate temporisés que nous noterons UTA (*Updatable Timed Automata*) qui rassemble le fragment pour lequel le problème du vide est décidable. Comme vont l'illustrer les exemples, chapitre 7 page 111, cette classe facilite considérablement la modélisation d'algorithmes et de protocoles réels (voir chapitre 7 page 111). En partie parce qu'elle permet de manipuler de façon aisée la valeur des horloges. Mais, elle introduit en outre une notion de mise à jour aléatoire (mises à jour non déterministes) qui permet d'envisager la vérification d'algorithmes utilisant des mises à jour aléatoires pour les horloges.

Chapitre 6

Les Automates 0/1

Nothing is more sad than the death of an illusion.
— Arthur Koestler

Les résultats obtenus sur la décidabilité du problème du vide pour les automates temporisés avec mises à jour nous ont encouragé à regarder un peu plus loin. À l'occasion d'un séminaire de Franck Cassez qui venait présenter les résultats d'un article écrit en collaboration avec Kim G. Larsen [CL00b], nous avons eu l'occasion de nous pencher sur le problème des modèles avec une extension hybride des automates temporisés avec mises à jour. Comme nous étions particulièrement sensibilisés au problème des gardes diagonales ($x - y \sim c$), nous avons assez rapidement remarqué que la preuve de l'équivalence entre les automates hybrides et les automates temporisé 0/1 (*stopwatches automata*) présentée dans [CL00b] comportait un grand nombre de gardes diagonales.

Catherine Dufourd et moi avons travaillé sur cette question en espérant que le fait de retirer les gardes diagonales nous permettrait de garder la décidabilité du test du vide. Comme cela avait été le cas pour les automates temporisés avec mises à jour non déterministes. Mais la frontière entre décidable et indécidable semble être plus proche que nous l'avions envisagé au début.

Nous allons commencer par définir les automates 0/1, puis nous exposerons les résultats et les preuves de ceux-ci.

6.1 Les automates hybrides

Le modèle des systèmes hybrides englobe strictement celui des automates temporisés. On y retrouve des évolutions discrètes qui correspondent aux actions et des évolutions continues qui correspondent à l'écoulement du temps. La différence majeure entre les deux modèles est que l'on assigne à chaque état des contraintes sur l'évolution du temps de chaque horloge (*i.e.* la dérivée de x par rapport au temps, on la note $\dot{x}$). Le modèle des automates temporisés peut être vu comme un cas particulier où pour tous les états de l'automate et pour toutes les horloges $x \in X$, on a $\dot{x} = 1$.

Un *automate hybride* est un 8-uplet $\mathcal{H} = (\Sigma, Q, T, \delta, I, F, R, X)$ avec Σ un alphabet fini d'*actions*, Q un ensemble fini d'*états*, X un ensemble fini d'*horloges*, $I \subseteq Q$ l'ensemble des *états initiaux*, $F \subseteq Q$ l'ensemble des *états finaux*, $R \subseteq Q$ l'ensemble des *états répétés*, δ :

$Q \times X \to \mathbb{R}$ la fonction donnant les taux de croissance des horloges en fonction des états, et $T \subseteq Q \times [\mathcal{C}(X) \times \Sigma \times \mathcal{U}(X)] \times Q$ un ensemble fini de *transitions*. Ainsi, une transition est un 5-uplet (q, g, a, u, q'), avec :

- $q, q' \in Q$, deux états,
- $g \in \mathcal{C}(X)$, une *garde*,
- $a \in \Sigma$, une *action*,
- $u \in \mathcal{U}(X)$, une *mise à jour*.

Un *chemin* (*path*) p dans $\mathcal{H} = (\Sigma, Q, T, \delta, I, F, R, X)$ est une suite finie ou infinie de la forme :

$$p = q_0 \xrightarrow{g_1,\sigma_1,up_1} q_1 \xrightarrow{g_2,\sigma_2,up_2} q_2 \xrightarrow{g_3,\sigma_3,up_3} q_3 \ldots, \quad \text{avec } q_0 \in I, \text{ et } \forall i > 0,\ (q_{i-1}, g_i, \sigma_i, up_i, q_i) \in T \tag{6.1}$$

On note $inf(p) \subset Q$ l'ensemble des états qui sont infiniment souvent répétés sur le chemin p. Le chemin fini (resp. infini) p est *acceptant* si l'état final est dans F (resp. si $inf(p) \cap R \neq \emptyset$, *condition de Büchi*).

Une *exécution* (*run*) r sur le chemin p est une suite de la forme :

$$r = \langle q_0, v_0 \rangle \xrightarrow[t_1]{g_1,\sigma_1,up_1} \langle q_1, v_1 \rangle \xrightarrow[t_2]{g_2,\sigma_2,up_2} \langle q_2, v_2 \rangle \ldots, \text{ avec } \forall i \geq 0,\ v_i \in \mathbb{T}^X \tag{6.2}$$

avec $(t_i)_{i>0}$ une séquence temporisée et $(v_i)_{i\geq 0}$ des valuations d'horloges telles que :

- $v_0(x) = 0,\ \forall x \in X$,
- $\forall i > 0,\ v_{i-1} + \delta(q_{i-1})(t_i - t_{i-1}) \models g_i$,
- $\forall i > 0,\ v_i \in up_i(v_{i-1} + \delta(q_{i-1})(t_i - t_{i-1}))$.

avec $v + \delta(q)t = (x_i + \delta(q, x_i)t)_i$. Et on appelle les couples $\langle q_i, v_i \rangle$, des *états étendus*.

L'étiquette de r est le mot temporisé $(\sigma_1, t_1)(\sigma_2, t_2)\ldots$, qui est dit *accepté* par l'automate hybride $\mathcal{H}$. L'ensemble des mots qui sont l'étiquette d'une exécution acceptante de $\mathcal{H}$ forment le langage accepté (ou reconnu) par $\mathcal{H}$ et on le note $L(\mathcal{H}, \mathbb{T})$ ou plus simplement $L(\mathcal{H})$.

On note $Hyb(C(X), U(X))$ l'ensemble des automates hybrides avec mises à jour construits avec l'ensemble des contraintes sur les horloges contenues dans $C(X)$ et l'ensemble des mises à jour dans $U(X)$. Nous écrirons simplement $Hyb(U, C)$ lorsqu'il n'y aura pas d'ambiguïté.

Les *automates* 0/1 sont des automates hybrides tels que $\delta : Q \times X \to \{0, 1\}$. On note $Hyb_{0/1}(U, C)$ l'ensemble des automates temporisés 0/1 avec mises à jour construits avec l'ensemble des contraintes sur les horloges contenu dans C et l'ensemble des mises à jour dans U.

Exemple 16. *L'automate de la figure 6.1 page ci-contre est un exemple d'automate* 0/1 *avec mises à jour.*

6.2 Indécidabilité

Nous allons à présent montrer que le problème du vide est indécidable pour les automates 0/1 qui ne considère que des gardes non diagonales ($x \sim c$).

Théorème 19. *Le problème du vide est indécidable sur la classe des automates* 0/1 *avec gardes diagonales ($x - y \sim c$) et ayant au moins trois horloges.*

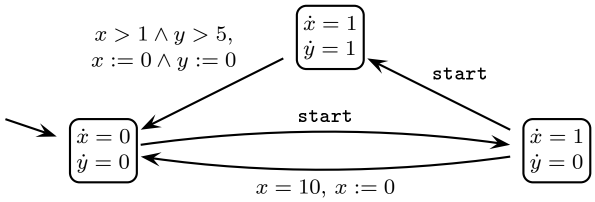

FIG. 6.1 – *Exemple d'automate* 0/1 *avec mise à jour.*

Démonstration. Comme dans les preuves du chapitre 3 page 55, nous allons simuler une machine à deux compteurs (incrémentation et décrémentation sur les horloges). Cette preuve est assez naturelle, on utilise l'horloge z pour stocker la valeur actuelle de l'horloge que l'on veut incrémenter/décrémenter et on vérifie ensuite que la différence entre z et l'horloge visée est bien de $+1/-1$.

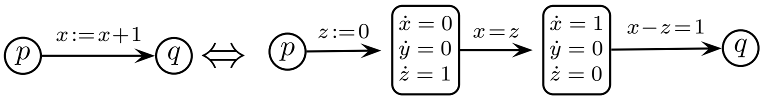

FIG. 6.2 – *Simulation d'une mise à jour "$x := x + 1$".*

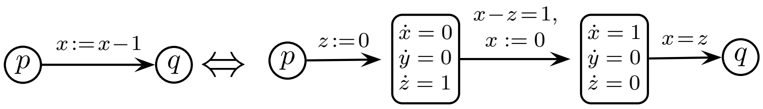

FIG. 6.3 – *Simulation d'une mise à jour "$x := x - 1$".*

□

Ce premier théorème n'est pas une nouveauté, il était facile de voir l'indécidabilité du problème du vide comem un corollaire de [CL00b]. Le résultat suivant est moins évident car la preuve de [CL00b] repose en grande partie sur l'utilisation de gardes diagonales $x - y \sim c$. Or, comme nous l'avons vu dans les chapitres précédents, l'utilisation ou non de gardes diagonales peut changer la décidabilité du problème du vide. La question que nous nous sommes posés était d'établir la décidabilité du problème du vide lorsqu'on se restreint à n'utiliser que des gardes simples ($x \sim c$).

Théorème 20. *Le problème du vide est indécidable sur la classe des automates* 0/1 *avec gardes non diagonales ($x \sim c$) et ayant au moins quatre horloges.*

Démonstration. Cette preuve repose sur une tout autre idée que les précédentes. Nous allons donc abandonner la machine à deux compteurs pour nous intéresser à un résultat publié par Catherine Dufourd et Béatrice Bérard.

Proposition 5 ([BD00]). *Le problème du vide est indécidable sur les automates temporisés qui autorisent des gardes du type $x + y = 1$.*

Nous allons donc simuler une garde $x + y = 1$ sans utiliser de garde diagonale avec des automates 0/1. L'horloge z sert à additionner les horloges x et y et l'horloge w sert à permettre de ne pas perdre l'information de la valeur de x et y.

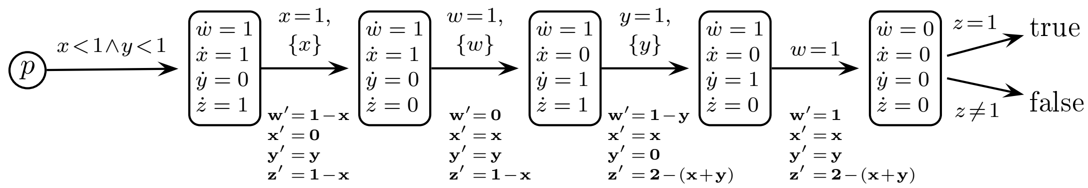

FIG. 6.4 – *Simulation d'une garde "$x + y = 1$".*

La deuxième transition permet de mettre la valeur $1-x$ dans w et z. La transition suivante permet de positionner x à sa valeur de départ en utilisant w. La transition suivante permet de rajouter $1 - y$ à z et d'obtenir la valeur $2 - (x + y)$. Comme pour x, on mémorise $1 - y$ dans w afin de remettre l'horloge y à sa valeur initiale à la transition suivante. Au final, nous avons l'horloge z qui contient la valeur $2 - (x + y)$. Si $x + y = 1$, alors $z = 1$, sinon $z \neq 1$.

$\square$

Conclusion

Comme nous venons de le voir, lorsqu'on considère un modèle, même restreint, d'automates hybrides, on perd la décidabilité du problème du vide. L'espoir que nous avions était que les mises à jour non déterministes introduisent des différences dans l'évolution des différentes horloges en permettant des 'sauts'. Mais visiblement, le fait de contrôler ces sauts dans le temps est trop expressif pour conserver la décidabilité du problème du vide.

Chapitre 7

Quelques Exemples

Few things are harder to put up with
than the annoyance of a good example.

— Mark Twain

Ce chapitre présente un panel d'exemples pour lesquels le modèle des automates temporisés avec mises à jour s'avère plus efficace que le modèle classique. Soit du point de vue de la compacité du modèle obtenu (files, voir section 7.1), soit parce que l'expressivité requise nécessite des mises à jour que les automates temporisés classiques ne peuvent simuler (protocole CSMA/CD, voir section 7.2 page 114).

7.1 Les files

Nous allons nous intéresser à différentes stratégies concernant la gestion d'un tampon emmagasinant des tâches ayant des contraintes temps-réel. Par '*contraintes temps-réel*', j'entends ici qu'à chaque tâche, on associe un temps maximum de vie. Plus formellement, à chaque type de tâche t_i, on associe une échéance (*deadline*) d_i. L'ensemble des types de tâches temps-réel est donc l'ensemble des couples $(t_i, d_i)_i$. Passée son échéance, la tâche est considérée comme invalide. Dans les exemples suivants, on veut vérifier une propriété de temps-réel dur (*hard real-time*) qui requiert que toutes les tâches soient traitées avant que leur échéance n'arrive à leur terme.

Le modèle que nous proposons dépend de deux autres processus que nous ne représenterons pas. Le premier processus produit des tâches et les insère dans la file en délivrant un message *put*, s'il n'y a qu'un type de tâche et put_i avec i le type de la tâche s'il y a plusieurs types de tâches. Le deuxième processus retire la première tâche de la file en délivrant un message *get*. On suppose que la file possède n places. Chaque place possède une horloge x_i avec $1 \leq i \leq n$. On suppose aussi que ces n places sont ordonnées et que la façon d'ordonner les tâches varie selon les types de files que l'on considère. Ainsi on a :

- *put* : Insère une tâche dans la file,
- put_i : Insère une tâche de type i dans la file,
- *get* : Retire la première tâche de la file.

7.1.1 FILO (*First In Last Out*)

La première stratégie d'ordonnancement que nous allons modéliser est la FILO (*First In Last Out*). C'est la plus facile à gérer car on extrait seulement la tâche qui est sur le haut de la pile, ce qui permet d'éviter d'avoir à réorganiser l'ordre des tâches à chaque insertion d'une nouvelle.

Si on considère un seul type de tâche, on obtient le modèle représenté figure 7.1. À chaque insertion d'une nouvelle tâche, on initialise à zéro l'horloge correspondant à cette place dans le tampon. Et à chaque retrait d'une tâche du tampon, on retire la dernière tâche mise dans la pile en redescendant d'un niveau et on vérifie que l'échéance n'a pas été dépassée (condition temps-réel dur).

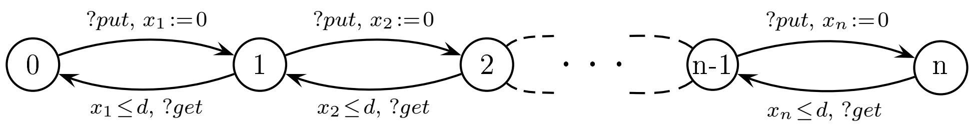

FIG. 7.1 – *File FILO avec un seul type de tâche* (t, d).

Si on considère plusieurs types de tâches, le modèle devient un peu plus subtil. Tout d'abord, on est forcé d'avoir autant de transitions qu'il y a de type de messages pour réceptionner les messages de type put_i. Mais si on se contentait de mettre les horloges à zéro, il faudrait se rappeler quel type de tâche est à chaque place afin de vérifier lors de la sortie de la tâche la condition temps-réel dur, ce qui augmenterait considérablement le nombre d'états et de transitions. On va utiliser des mises à jour déterministes de type $x := c$, avec c négatif[1] pour empêcher cette explosion. Il suffit de mettre à jour l'horloge de la tâche à $-d_i$. À la sortie de la tâche du tampon, on teste uniquement que cette horloge est inférieure à zéro. C'est le même test pour toutes les tâches. Le modèle obtenu est représenté sur la figure 7.2. Ce modèle est beaucoup plus compact et plus facile à manipuler que celui qui doit comporter tous les cas de figure lorsqu'on considère plusieurs types de tâches.

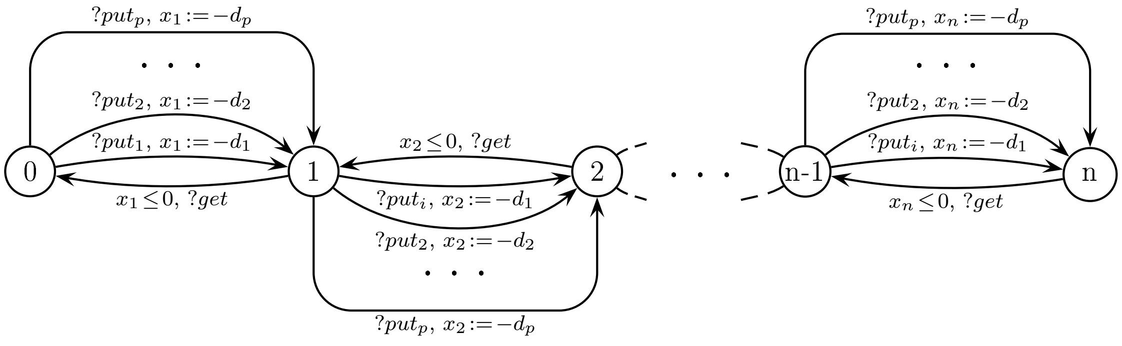

FIG. 7.2 – *File FILO avec* m *types de tâches* $(t_i, d_i)_{i \leq m}$.

[1] Seules les mises à jours de type $x := y + c$ requièrent que c soit positif.

7.1.2 FIFO (*First In First Out*)

La deuxième stratégie d'ordonnancement que nous modéliserons ici est sans doute la plus classique. Il s'agit de la FIFO (*First In First Out*). Elle pose un problème supplémentaire par rapport à la file FILO. En effet, à chaque fois que l'on retire une tâche de la file, on retire la plus ancienne. Ce qui force à réorganiser la file, soit à chaque insertion, soit à chaque sortie d'une tâche, même si on ne considère qu'un seul type de tâche. On utilise donc des mises à jour de type $x := y$ pour faire en sorte que le temps de vie de la première tâche soit toujours représentée par la première horloge (x_1), celui de la deuxième tâche dans la deuxième horloge (x_2), et ainsi de suite. Plus formellement, si on a p tâches dans la file, on aura un mise à jour de la forme $(x_1 := x_2) \wedge (x_2 := x_3) \wedge \cdots \wedge (x_{p-1} := x_p)$ lors de la suppression d'une tâche (voir figure 7.3).

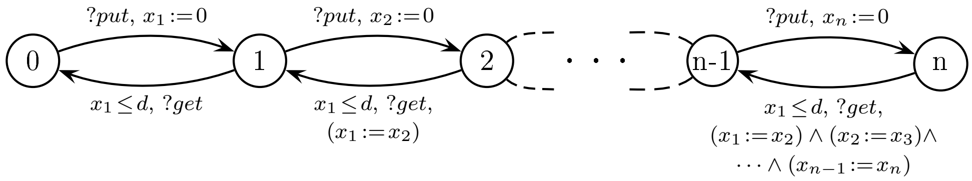

FIG. 7.3 – *File FIFO avec un seul type de tâche* (t, d).

Remarque 12. *Si nous ne disposions pas des mises à jour, nous serions obligés de nous souvenir du nom de l'horloge qui contient le temps de vie de la première tâche, ce qui accroîtrait de manière exponentielle le nombre d'états et de transitions. En utilisant les mises à jours, on évite une fois de plus cette explosion d'états sur le modèle (mais pas sur le graphe des régions).*

On peut étendre facilement ce modèle à un nombre quelconque de types de tâches en utilisant la même méthode que pour la FILO (voir figure 7.4).

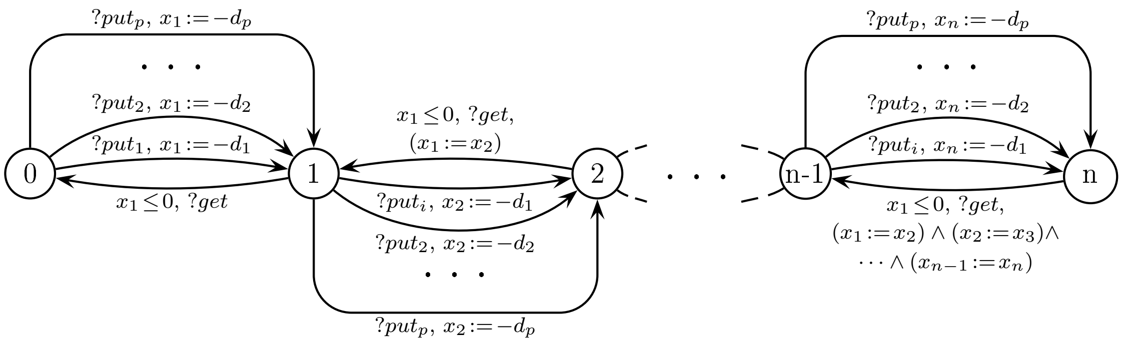

FIG. 7.4 – *File FIFO avec m types de tâches* $(t_i, d_i)_{i \leq m}$.

7.1.3 EDF (*Earliest Deadline First*)

La dernière stratégie d'ordonnancement à laquelle nous allons nous intéresser est typique des problèmes temps-réel. EDF (*Earliest Deadline First*) est une stratégie qui consiste à ordonner les tâches par ordre de priorité. Plus leur échéance est proche, plus elles sont prioritaires.

Cela se traduit par un ordonnancement en fonction de leur temps restant avant échéance, du plus petit au plus grand. Dans ce cas, si on ne considère qu'un seul type de tâche, on retombe sur une stratégie de FIFO. Les cas intéressants sont donc les files EDF avec plusieurs types de tâches.

Comme on suppose que l'échéance d'une tâche ne peut pas être modifiée par autre chose que le temps qui s'écoule, il suffit d'ordonner les tâches lors de leur arrivée dans la file. Pour cela, on insère la nouvelle tâche entre une tâche qui a une échéance plus petite et une tâche qui a une échéance plus grande (les extrémités ayant des échéances de 0 et $+\infty$). Plus le nombre de tâches présentes dans la file est grand, plus le nombre de possibilités pour l'insertion est grand (pour p taches en file, le nombre de possibilités est de $p+1$).

Plus formellement, on cherche à insérer la nouvelle tâche t_i entre deux tâches consécutives dans la file telles que $x_j \leq -d_i < x_{j+1}$ (ou $-d_i \leq x_1$ et $x_p < -d_i$, avec p le nombre de tâches présentes en file, pour les extrémités). Il faut ensuite réorganiser la file en mettant à jour les horloges de la façon suivante : $(x_{j+1} := -d_i) \wedge (x_{j+2} := x_{j+1}) \wedge \cdots \wedge (x_{p+1} := x_p)$ (voir figure 7.5).

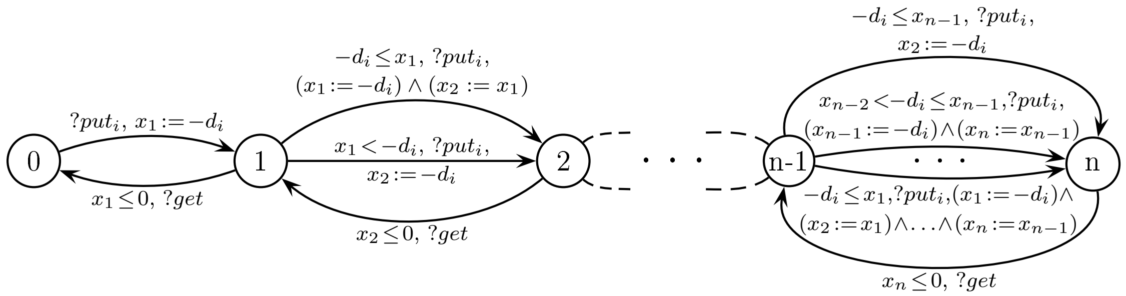

FIG. 7.5 – *File EDF avec m types de tâches* $(t_i, d_i)_{i \leq m}$.

Une fois de plus, les mises à jour ont évité une explosion du nombre d'états et de transitions, permettant de conserver un modèle compact et clair.

Nous allons ensuite explorer les possibilités des mises à jour non déterministes qui permettent de modéliser aisément des algorithmes et des protocoles non déterministes sur le choix d'une assignation d'horloge.

7.2 Le protocole CSMA/CD

Pour mieux illustrer les capacités des automates temporisés avec mises à jour nous allons modéliser le protocole réseau Ethernet. L'intérêt de l'exercice tient dans le fait qu'il n'est pas possible de faire un modèle précis de ces protocoles avec des automates temporisés classiques alors que l'utilisation d'automates temporisés avec mises à jour le permet. De plus, ce protocole se rencontre très fréquemment au sein des systèmes informatiques. Être capable de les modéliser précisément est donc un réel avantage. Plusieurs modèles de ce protocole existent déjà [ACH94], dont certains basés sur des automates temporisés [DOY95] (on peut aussi citer la protocole de '*collision avoidance*' [JLS96]), mais ils ne font qu'une modélisation imparfaite du protocole à cause du manque d'expressivité des automates temporisés classiques.

7.2.1 La famille des protocoles CSMA/XX

D'un point de vue général, on peut trouver deux grandes familles de protocole réseau :

– Les protocoles de *routage* (IP, ATM, ...),

– Les protocoles de *diffusion* (Ethernet, Token Ring, FDDI, ...).

Le rôle des protocoles de routage est de trouver une route permettant d'acheminer un message à travers un réseau représenté sous la forme d'un graphe. Le rôle du protocole de diffusion est de gérer les accès à un canal de communication partagé. La famille des protocoles de type CSMA/XX fait partie de la classe des protocoles de diffusion.

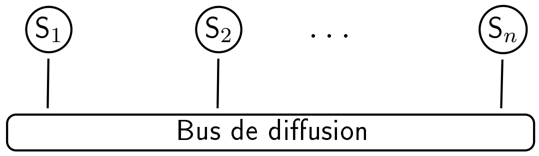

FIG. 7.6 – *Modèle du réseau pour CSMA/XX.*

CSMA/XX a un cadre matériel précis. On se donne un ensemble de sources $(S_i)_{i\leq n}$ et un bus de diffusion qui transmet les messages par une opération de *broadcast* (voir fig. 7.6). Outre les hypothèses déjà citées, on impose aussi au modèle de permettre les opérations suivantes :

– **Carrier Sense (CS)** : Capacité pour une source de détecter tout trafic sur le bus sans l'altérer,

– **Multi-Access (MA)** : Capacité pour le bus de recevoir *simultanément* les messages de plusieurs sources.

Ce genre de modèle représente une racine commune pour un grand nombre de types de réseaux (BNC, hubs, ondes radio, infra-rouge, ...). On peut citer trois protocoles qui utilisent ce genre d'environnement matériel avec cependant des hypothèses légèrement différentes :

– CSMA/CD (Collision Detection), IEEE 802.3 (Ethernet, 3Com Corp.),

– CSMA/DCR (Deterministic Collision Resolution), Ethernet Déterministe (G. Le Lann et P. Rollin [LR84], INRIA),

– CSMA/CA (Collision Avoidance), IEEE 802.11 (LocalTalk, Apple Computer, Inc.).

Si l'on considère uniquement des protocoles non déterministes, on peut dès à présent construire un protocole naïf qui permet le partage du bus de diffusion pour l'envoi de message (voir figure 7.7).

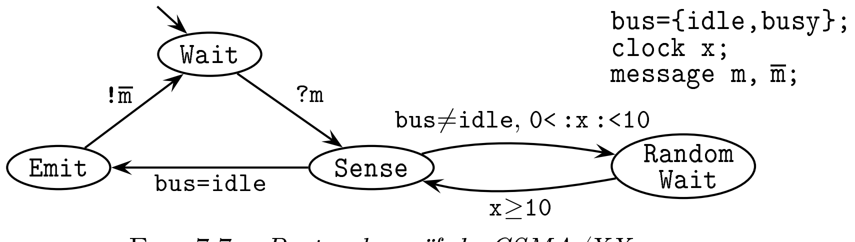

FIG. 7.7 – *Protocole naïf de CSMA/XX.*

Pour effectuer un envoi, le protocole naïf consiste à tester si le canal de communication qu'il partage avec les autres sources est libre ou non (état '`Sense`'). Si le canal est libre, il émet (état '`Emit`'). Dans le cas contraire, il tire un nombre aléatoire et attend autant d'unités de temps avant de recommencer à tester le canal (état '`Random Wait`').

Cependant, lorsqu'on s'intéresse à un type de réseau particulier, comme les réseau Ethernet, les hypothèses qui s'ajoutent à celles déjà présentes dans le modèle de base obligent à corriger ce protocole naïf. Nous allons donc nous intéresser plus spécifiquement à ce protocole et aux hypothèses qu'il impose.

7.2.2 CSMA/CD (Collision Detection)

Les prémisses de la norme Ethernet ont été posées en 1973 par Bob Metcalfe (fondateur du groupe 3Com) au centre de recherche Xerox de Palo Alto en Californie. Il s'agissait à l'époque de connecter des stations de travail à une imprimante laser. En 1980, DEC, Intel et Xerox ont conjointement créé la norme *DIX* pour l'Ethernet 10Mbps. En 1985, la norme IEEE 802.3 basée sur la norme DIX fut adoptée par l'ISO (International Organization for Standardization). Depuis, la norme Ethernet évolue régulièrement. En 1995, le *fast ethernet* (100Mbps) est standardisé par la norme IEEE 802.3u. En 1999 c'est au tour du *gigabit ethernet* (1Gbps) avec la norme IEEE 802.3z.

Le protocole CSMA/CD repose sur un algorithme non-déterministe de type Las Vegas[2]. Il s'utilise sur des réseaux en série à impulsions électroniques (câbles coaxiaux ou paires torsadées). Une conséquence de ce cadre matériel est que la diffusion d'un message n'est pas instantanée. En effet, comme on se place dans le contexte précis de signaux électroniques (codage Manchester) sur un conducteur électrique, la propagation du signal est de l'ordre de $0,77\,c = 23000\,km/s$. À titre d'exemple, pour un débit de 10Mbps, un bit occupe le signal électrique pendant $0,1\,\mu s$. Il s'étale donc sur une distance de $23\,m$. Sur un réseau de longueur de $2800\,m$ (maximum préconisé), on peut placer une dizaine de bits les uns à la suite des autres et il faut, au pire, $1\,\mu s$ au premier bit pour atteindre l'autre bout du réseau. Si l'on ramène ces grandeurs de temps à un cycle de processeurs actuels, on comprend qu'il puisse y avoir des collisions sur le bus. Les sources peuvent émettre et écouter le bus en même temps. Enfin, les sources peuvent aussi détecter trois états du bus : *Idle* ou *libre* (pas de signal), *Busy* ou *occupé* (un unique signal), *Jammed* ou *brouillé* (plusieurs signaux superposés).

L'hypothèse CD (*Collision Detection*) permet de résoudre le problème des collisions (superposition de plusieurs signaux) qui peuvent arriver à cause de la non instantanéité des transmissions. L'hypothèse CD n'est pas forcément présente dans tous les cas (réseau hertzien par exemple).
Pour résumer, les hypothèses supplémentaires sont les suivantes :

1. La diffusion d'un message n'est pas instantanée,
2. Les sources peuvent émettre et écouter simultanément,
3. Les sources peuvent détecter trois états du bus (`idle`, `busy`, `jammed`).

Le protocole CSMA/CD, inclus dans la norme IEEE 802.3, se décompose en deux protocoles distincts : un protocole d'envoi de messages et un protocole de récupération des collisions.

[2]Probabilité d'arrêt strictement inférieure à 1

• Le **protocole d'envoi de messages** ressemble au protocole général de la classe CSMA/XX (voir figure 7.7 page 115). Toutefois, il tient compte du temps de diffusion non nul du message sur le bus. En fait, il repose en grande partie sur le fait que l'on peut borner la latence du bus.

On note s le temps que met un signal à parcourir le bus d'une extrémité à l'autre. Et $r = 2.s$, la *tranche canal*, c'est à dire le temps d'un aller et retour d'un signal sur le bus. Le protocole d'envoi d'un message se déroule comme indiqué par l'automate de la figure 7.9 page 119. Lorsqu'une source veut émettre un message, elle teste si le bus est libre puis débute l'émission. Pendant un laps de temps r la source considère l'émission comme « *non sûre* » puisque dans le pire cas (lorsque la source se trouve à l'extrémité du bus) c'est le temps que va mettre le signal pour atteindre toutes les autres sources et revenir éventuellement brouillé. Si au bout de cette période aucun brouillage n'est détecté, l'émission est considérée comme « *sûre* » et peut se poursuivre.

Il est à noter que ce genre de protocole impose l'hypothèse supplémentaire imposant aux messages d'avoir une durée minimale de r (soit une vingtaine de bits dans le cas d'Ethernet 10Mbits).

• Le **protocole de résolution de collision** est la partie non-déterministe du protocole CSMA/CD. Il repose principalement sur le fait que le temps de diffusion d'un message sur le canal est borné (τ).

Lors d'une collision, toutes les sources qui émettent ou qui sont en attente d'émission débutent le protocole de résolution de collision :

- Les messages en cours d'émission sont prolongés jusqu'à une durée d'émission de r.
- Chaque source tire aléatoirement un nombre entier p entre 0 et un entier n_{max}.
- La source attend que le bus soit *libre* et attend $p.r$ (p fois la tranche canal) avant de réessayer d'émettre.
- Si une collision se produit :
  - 2^{me} tentative : $p := p + 2$

⋮

  -
  - $10^{\text{ème}}$ tentative : $p := p + 2$
  - $11^{\text{ème}}$ tentative : $p := p$
  - $12^{\text{ème}}$ tentative : $p := p$

⋮

  -

  Et on répète l'algorithme avec la nouvelle valeur de p. En règle générale après seize essais infructueux, le bus est considéré comme saturé et le message est considéré comme perdu.

7.2.3 Le modèle

Comme pour le modèle des files que nous avons établi précédemment, nous allons considérer des messages qui comportent des échéances. Plus formellement, on pose $(m_i, d_i)_i$ avec m_i un type de message et d_i son échéance. Ici, il s'agit évidemment d'un exemple jouet car nous ne pouvons pas vérifier de propriétés temps-réel dures avec un protocole de type Las Vegas. En effet, le protocole peut très bien ne jamais réussir à émettre si deux sources ont exactement le même temps et tirent le même nombre aléatoire p. On suppose de plus qu'un processus extérieur, que nous ne représenterons pas ici, fournit les messages aux sources.

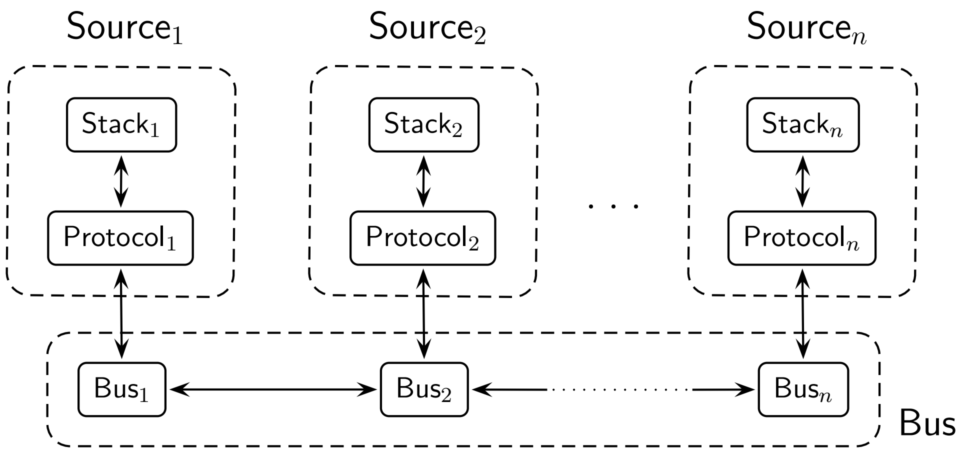

FIG. 7.8 – *Interactions entre les automates du système.*

Le découpage du problème est représenté sur la figure 7.8. Chaque source est composée d'un automate qui représente sa file (on peut imaginer reprendre l'une des files décrites précédemment) et de son protocole. Dans notre cas précis, le protocole sera celui de CSMA/CD. Le bus à diffusion, lui, est modélisé par autant d'automates qu'il y a de sources. Chaque partie du bus communiquant uniquement avec ses deux voisins et sa source. Pour des raisons de simplicité, nous nous restreindrons à deux sources ($Source_1$ et $Source_2$) et à un seul type de message (m, d).

Les automates se synchronisent de façon binaire par émission/réception de messages dont la liste est donnée dans le tableau 7.1 page suivante. Voici la description plus précise de ces messages :

– I_i, B_i, J_i : sont émis par $Protocol_i$ pour tester l'état du Bus_i, respectivement, à *idle*, *busy*, *jammed*,

– S_i, $\overline{S}_i$: sont émis par $Protocol_i$ vers Bus_i pour, respectivement, débuter et arrêter la transmission d'un message,

– $I_{i\to j}$, $B_{i\to j}$, $J_{i\to j}$: sont émis par Bus_i vers Bus_j pour signaler, respectivement, la propagation d'un signal *idle*, *busy* ou *jammed*,

– m_i : est émis par $Stack_i$ vers $Protocol_i$ pour signaler qu'un message au moins est présent en file,

– $\overline{m}_i$: est émis par $Protocol_i$ vers $Stack_i$ pour signaler que le message courant a été traité.

On ne représentera pas à nouveau les automates de gestion des files. On pourra se référer au début de ce chapitre pour imaginer ce à quoi cette partie doit ressembler.

L'automate modélisant le protocole de la $Source_1$ est représenté sur la figure 7.9 page ci-contre. Il suffit d'échanger 1 et 2 dans l'automate du protocole de la $Source_1$ pour obtenir celui de la $Source_2$. On a simplifié le choix aléatoire du temps d'attente, mais on pourrait très bien représenter fidèlement celui-ci. La notation $(A!; B!)$ signifie que l'on émet A *puis* B (sans atomicité).

	Messages Envoyés		Messages Reçus	
Bus_1	Vers $\mathsf{Protocole}_1$		De $\mathsf{Protocole}_1$	$I_1,\ B_1,\ J_1$ $S_1,\ \overline{S}_1$
	Vers Bus_2	$I_{1\to 2}$ $B_{1\to 2}$ $J_{1\to 2}$	De Bus_2	$I_{2\to 1}$ $B_{2\to 1}$ $J_{2\to 1}$
Bus_2	Vers $\mathsf{Protocole}_2$		De $\mathsf{Protocole}_2$	$I_2,\ B_2,\ J_2$ $S_2,\ \overline{S}_2$
	Vers Bus_1	$I_{2\to 1}$ $B_{2\to 1}$ $J_{2\to 1}$	De Bus_1	$I_{1\to 2}$ $B_{1\to 2}$ $J_{1\to 2}$
$\mathsf{Protocole}_1$	Vers Bus_1	$I_1,\ B_1,\ J_1$ $S_1,\ \overline{S}_1$	De Bus_1	
	Vers Stack_1	$\overline{m}_1$	De Stack_1	m_1
$\mathsf{Protocole}_2$	Vers Bus_2	$I_2,\ B_2,\ J_2$ $S_2,\ \overline{S}_2$	De Bus_2	
	Vers Stack_2	$\overline{m}_2$	De Stack_2	m_2
Stack_1	Vers $\mathsf{Protocole}_1$	m_1	De $\mathsf{Protocole}_1$	$\overline{m}_1$
Stack_2	Vers $\mathsf{Protocole}_2$	m_2	De $\mathsf{Protocole}_2$	$\overline{m}_2$

TAB. 7.1 – *Tableau des messages du système (CSMA/CD).*

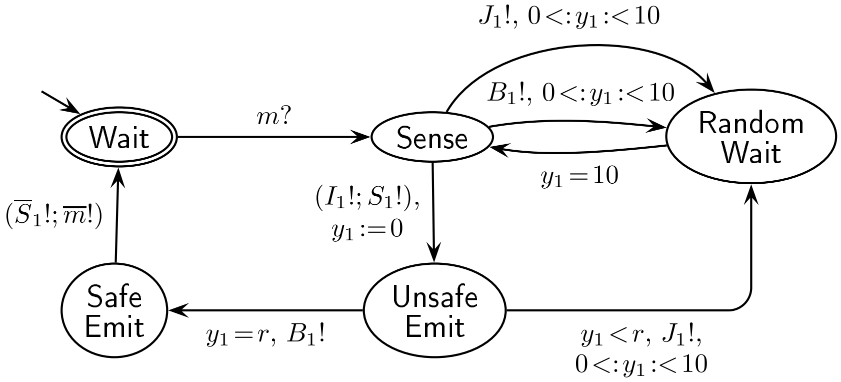

FIG. 7.9 – *Émetteur de la* $Source_1$ *(CSMA/CD).*

L'automate modélisant le Bus_1 est représenté sur la figure 7.10. De même que pour l'automate du $Protocol_1$, il suffit de substituer 1 et 2 et vice versa pour obtenir l'automate Bus_2. Le Bus_1 comporte six états. Les états $Idle_1$, $Busy_1$ et Jam_1 représentent respectivement les états *idle*, *busy* et *jammed* du bus. Mais aussi des états de transitions qui correspondent à la phase de diffusion d'un changement d'état le long du bus. Ces états sont $Idle_1 \to Busy_1$, $Busy_1 \to Idle_1$ et $Busy_1 \to Jam_1$. Toutes ces diffusions prennent un temps s et peuvent être modifiées en cours de route par un message d'un bus voisin.

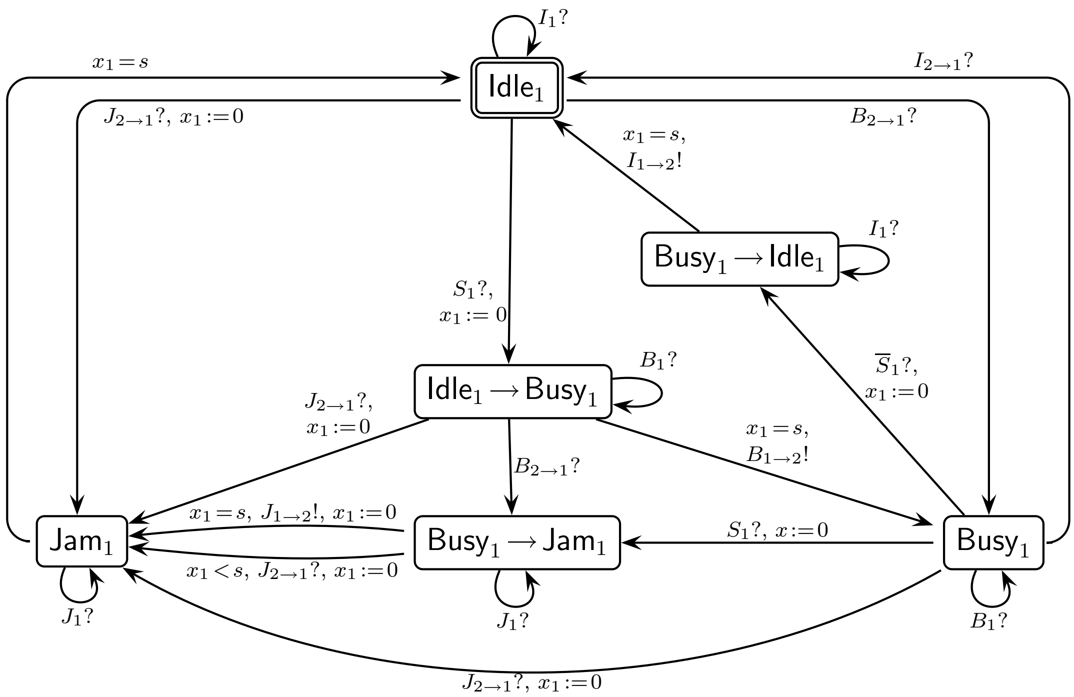

FIG. 7.10 – *Bus de la Source$_1$ (CSMA/CD).*

7.3 Conclusion

Pour conclure ce chapitre, nous avons montré que les mises à jour permettaient à la fois une plus grande compacité et une plus grande simplicité des modèles (exemple des files, section 7.1 page 111) et d'autre part que l'expressivité du modèle des automates avec mises à jour permettait de modéliser précisément des protocoles réalistes tels que CSMA/CD (section 7.2 page 114). L'absence de gardes de type $x - y \sim c$ ne semble pas interférer outre mesure dans la modélisation de tels protocoles.

L'ajout de mises à jour plus évoluées au modèle original des automates temporisés semble donc un avantage indéniable, d'autant plus que la complexité théorique du problème du vide reste la même qu'auparavant.

Conclusion

A conclusion is the place where you got tired thinking.
— Martin H. Fischer

Bilan de la thèse

À travers cette thèse nous avons présenté et analysé une extension non triviale des remises à zéro du modèle classique des automates temporisés.

Dans un premier temps nous avons rappelé la définition des automates temporisés classiques [AD90, AD94] et les principaux théorèmes qui s'y rattachent [AD94, AKV98]. Nous avons alors évoqué le graphe des régions et l'automate des régions qui constituent l'abstraction à travers laquelle on résoud le problème du vide pour les automates temporisés classiques.

Nous avons ensuite introduit le modèle complet des automates temporisés avec mises à jour en décrivant l'ensemble de ces mises à jour (déterministes et non-déterministes) [BDFP00a, BDFP00b]. La définition de ce modèle emprunte un grand nombre d'éléments au modèle original mais nous avons différencié les modèles d'automates temporisés avec mises à jour qui autorisent les gardes diagonales ($x - y \sim c$) et ceux qui ne le font pas. Comme nous l'avons souligné dans la suite, cette différence, apparemment sans conséquence pour le modèle original, s'avère cruciale dans le modèle que nous introduisons.

Une fois le modèle général défini, nous avons établi un ensemble de fragments pour lesquels nous avons étudié la décidabilité du problème du vide. Les preuves d'indécidabilité ont été réalisées en simulant une machine à deux compteurs (machine de Minsky [Min67]) à partir d'un des fragments indécidables. Cette technique permet d'obtenir des preuves à la fois courtes et facilement compréhensibles. Les preuves de décidabilité sont plus techniques. Elles consistent à donner une méthode de construction du graphe des régions en montrant qu'il est fini et compatible avec l'automate original. Ce qui permet d'en déduire la décidabilité du problème du vide. Pour avoir une idée claire de l'ensemble des résultats de décidabilité du problème du vide, on pourra se reporter au tableau 7.2 page suivante.

Après avoir établi les résultats de décidabilité, nous avons présenté nos résultats sur l'expressivité des parties décidables du modèle des automates temporisés avec mises à jour. Nous avons montré qu'il existe une relation de bisimulation forte entre le fragment décidable des automates temporisés avec mises à jour déterministes et les automates temporisés classiques. Puis, que le langage généré par le fragment décidable des automates temporisés avec mises à jour non-déterministes est strictement plus expressif que les automates temporisés classiques. Et enfin, que les langages générés par le fragment décidable des automates temporisés avec

<table>
<tr><th></th><th>Mises à jours</th><th>Contraintes non diagonales</th><th>Contraintes générales</th></tr>
<tr><td rowspan="5">Déterministes</td><td>$x := 0$</td><td rowspan="3">Pspace</td><td rowspan="2">Pspace</td></tr>
<tr><td>$x := c,\ c \in \mathbb{Q}_+$</td></tr>
<tr><td>$x := y + c,\ c \in \mathbb{Q}_+$</td><td rowspan="3">Indécidable</td></tr>
<tr><td>$x := y - 1$</td><td rowspan="2">Indécidable</td></tr>
<tr><td>$x := y + c,\ c \in \mathbb{Q}$</td></tr>
<tr><td rowspan="8">Non-déterministes</td><td>$x :< c,\ c \in \mathbb{Q}_+$</td><td rowspan="7">Pspace</td><td>Pspace</td></tr>
<tr><td>$x :> c,\ c \in \mathbb{Q}_+$</td><td rowspan="7">Indécidable</td></tr>
<tr><td>$x :< y$</td></tr>
<tr><td>$x :> y$</td></tr>
<tr><td>$x :< y + c,\ c \in \mathbb{Q}_+$</td></tr>
<tr><td>$x :> y + c,\ c \in \mathbb{Q}_+$</td></tr>
<tr><td>$y + c <: x :< y + d,$
$c, d \in \mathbb{Q}_+$</td></tr>
<tr><td>$y + c <: x :< z + d,$
$c, d \in \mathbb{Q}_+,\ y \neq z$</td><td>Indécidable</td></tr>
</table>

TAB. 7.2 – *Décidabilité des différentes sous-classes d'automates temporisés.*

mises à jour non-déterministes est inclus dans le langage des automates temporisés avec transitions silencieuses [BGP96, DGP97, BDGP98]. De façon plus concise, on a :

$$UTA_{det} \equiv_s TA \quad \text{et} \quad TA \subsetneq UTA_{ndet} \subseteq TA_\varepsilon \tag{7.1}$$

Pour parachever ces résultats, nous avons fait une brève incursion dans le domaine des automates hybrides pour y montrer l'indécidabilité d'une extension des mises à jour du modèle des *stopwatch automata* [CL00b]. Nous avons eu ici recours à un résultat d'indécidabilité qui établi que les gardes de type $x + y = 1$ rendent le modèle étudié indécidable pour le problème du vide [BD00].

Enfin, nous avons conclu par une série d'exemples utilisant le modèle des automates temporisés avec mises à jours qui mettent en évidence son utilité à travers la modélisation de problèmes temps-réels sur des files (FILO, FIFO, EDF) et sur des protocoles (Ethernet). Ces exemples mettent en exergue la compacité et la simplicité des modèles obtenus comparés à ceux qu'il aurait fallu générer avec les seuls automates temporisés classiques. les automates temporisés avec mises à jour permettent aussi de modéliser des modèles plus complexes grâces au gain d'expressivité qu'apportent les mises à jour non-déterministes. Par exemple, il devient possible de modéliser et de vérifier tout un ensemble de protocoles requérant la mise à jour d'une horloge à une valeur aléatoire, qui étaient jusqu'à présent inaccessibles.

Développements possibles

Un certain nombre de problèmes restent cependant ouverts. La première question que l'on peut se poser est de savoir s'il n'existe pas d'autres classes de mises à jour qui pourraient être décidables ? Nous nous sommes focalisés sur des types bien précis de mises à jours, peut-être pourrions-nous, à présent, élargir le champ des investigations et trouver d'autres mises à jour possibles. Comme par exemple les opérations de modulo sur les horloges [CG00].

D'un point de vue plus théorique, il pourrait aussi être intéressant de savoir si l'inclusion du langage des automates temporisés avec transitions silencieuses ($UTA_{ndet} \subseteq TA_{\varepsilon}$) est stricte ou non. Là aussi, nous conjecturons une inclusion stricte car il semble difficile de simuler des transitions silencieuses avec des mises à jour non-déterministes.

On peut aussi se poser la question de savoir si l'extension des mises à jour que nous avons ajouté aux automates temporisés classiques ne pourrait pas s'appliquer aussi à d'autre modèles proches, comme les *cost automata* [BFH+01, LBB+01] ou un fragment correctement choisi des automates temporisés probabilistes [D'A97, Seg95, PZ93, LS91, Var85].

Pour de futures implémentations dans Uppaal [BLL+96] ou Kronos [BDM+98], il est intéressant de remarquer que les résultats présentés dans le tableau 7.2 sont vrais pour des classes d'automates temporisés mais que l'on peut en raffiner les résultats. En effet, notre méthode de preuve établi une équivalence entre la décidabilité du problème du vide et l'existence d'une solution dans un système Diophantien donné par l'automate temporisé que l'on considère. On peut ainsi construire un algorithme qui teste la décidabilité du problème du vide pour un automate temporisé donné. Ce résultat est bien plus général que ceux présenté dans le tableau.

Par exemple, il existe des automates temporisés ayant des mises à jour de type $x := x-1$ et dont le système Diophantien possède une solution, ce qui permet d'en déduire que le problème du vide est décidable pour ces cas particuliers. Cependant, ce résultat n'est pas vrai de façon générale pour l'ensemble de la classe des ces automates temporisés. Un traitement au cas par cas reste donc envisageable et permet d'étendre encore le nombre de systèmes décidables.

En outre, rappelons que la plus petite solution du système Diophantien permet de construire un graphe des régions 'efficace' pour faire l'exploration de l'automate. Une conséquence immédiate de cette technique est que même lorsqu'on se restreint au cas classique ($x \sim c$, $x - y \sim c$ et $x := 0$), le graphe des régions obtenu est inclus ou égal à celui donné par l'algorithme classique qui consistait à collecter simplement les constantes maximales sur chaque garde et invariants. J'ai eu l'occasion d'implémenter cette technique dans Uppaal (version 3.2.10 et supérieures). Je n'ai cependant pas encore pris le temps de faire des tests pour évaluer les différences de performance par rapport à la méthode précédente.

Enfin, une autre voie à explorer est de ne plus construire un graphe des régions global de l'automate, mais de construire un graphe des régions local pour chaque état de l'automate. Cette solution permettrai à la fois d'augmenter encore le nombre des modèles décidables et de réduire la taille du graphe des régions que l'on considère. En combinant à cette approche des techniques d'analyses statiques sur les horloges et les variables entières, on peut imaginer un algorithme d'analyse qui pourrait réellement permettre la vérification de modèles jusqu'à présent inaccessibles.

Parmi les travaux probables qui s'effectueront dans le prolongement de cette thèse, l'intégration de ces mises à jour dans un outil existant tel que Uppaal [BLL+96], Kronos [BDM+98] ou CMC [LL98] semble envisageable. Cela permettrait assurément de valider (ou d'invalider) de manière expérimentale les possibilités offertes par ce genre de modèle. Un autre point intéressant serait de se poser la question de savoir s'il n'existe pas des structures plus efficaces que les DBM (*Difference Bounded Matrix*), qui intègrent par définition les différences d'horloges dans leur codage, pour coder les zones du graphe des régions. On sait déjà qu'on peut utiliser les DBM pour coder les automates temporisés avec mises à jour [Bou02b]. Cependant, la récente découverte de la non correction des algorithmes classiques d'extrapolation du *post** pour les modèles contenant des gardes diagonales ($x - y \sim c$) [Bou02b], laisse supposer une remise en question de ces structures de données et du modèle général qui était utilisé jusqu'à présent.

Ainsi se conclut ce rapport de thèse.

Annexe A

Documents complémentaires

A.1 Le rapport officiel du crash d'ATT

```
Date: 28 Jan 90 17:24:48 GMT
From: dhk@teletech.uucp (Don H Kemp)
Newsgroups: comp.dcom.telecom
Subject: AT&T Crash Statement: The Official Report
Organization: TELECOM Digest

Here's AT&T's official report on the Martin Luther King day network
problems, courtesy of the AT&T Consultant Liason Program.

Don
 ==========================================================

Technical background on AT&T's network slowdown, January 15, 1990

 *  *  *

At approximately 2:30 p.m. EST on Monday, January 15, one of AT&T's 4ESS
toll switching systems in New York City experienced a minor hardware
problem which activated normal fault recovery routines within the
switch. This required the switch to briefly suspend new call processing
until it completed its fault recovery action -- a four-to-six second
procedure. Such a suspension is a typical maintenance procedure, and is
normally invisible to the calling public.

As part of our network management procedures, messages were
automatically sent to connecting 4ESS switches requesting that no new
calls be sent to this New York switch during this routine recovery
interval. The switches receiving this message made a notation in their
programs to show that the New York switch was temporarily out of
service.
```

When the New York switch in question was ready to resume call processing a few seconds later, it sent out call attempts (known as IAMs - Initial Address Messages) to its connecting switches. When these switches started seeing call attempts from New York, they started making adjustments to their programs to recognize that New York was once again up-and-running, and therefore able to receive new calls.

A processor in the 4ESS switch which links that switch to the CCS7 network holds the status information mentioned above. When this processor (called a Direct Link Node, or DLN) in a connecting switch received the first call attempt (IAM) from the previously out-of-service New York switch, it initiated a process to update its status map. As the result of a software flaw, this DLN processor was left vulnerable to disruption for several seconds. During this vulnerable time, the receipt of two call attempts from the New York switch -- within an interval of 1/100th of a second -- caused some data to become damaged. The DLN processor was then taken out of service to be reinitialized.

Since the DLN processor is duplicated, its mate took over the traffic load. However, a second couplet of closely spaced new call messages from the New York 4ESS switch hit the mate processor during the vulnerable period, causing it to be removed from service and temporarily isolating the switch from the CCS7 signaling network. The effect cascaded through the network as DLN processors in other switches similarly went out of service. The unstable condition continued because of the random nature of the failures and the constant pressure of the traffic load in the network providing the call-message triggers.

The software flaw was inadvertently introduced into all the 4ESS switches in the AT&T network as part of a mid-December software update. This update was intended to significantly improve the network's performance by making it possible for switching systems to access a backup signaling network more quickly in case of problems with the main CCS7 signaling network. While the software had been rigorously tested in laboratory environments before it was introduced, the unique combination of events that led to this problem couldn't be predicted.

To troubleshoot the problem, AT&T engineers first tried an array of standard procedures to reestablish the integrity of the signaling network. In the past, these have been more than adequate to regain call processing. In this case, they proved inadequate. So we knew very early on we had a problem we'd never seen before.

At the same time, we were looking at the pattern of error messages and trying to understand what they were telling us about this condition. We have a technical support facility that deals with network problems, and they became involved immediately. Bell Labs people in Illinois, Ohio and

```
New Jersey joined in moments later. Since we didn't understand the
mechanism we were dealing with, we had to infer what was happening by
looking at the signaling messages that were being passed, as well as
looking at individual switches. We were able to stabilize the network by
temporarily suspending signaling traffic on our backup links, which
helped cut the load of messages to the affected DLN processors. At 11:30
p.m. EST on Monday, we had the last link in the network cleared.

On Tuesday, we took the faulty program update out of the switches and
temporarily switched back to the previous program. We then started
examining the faulty program with a fine-toothed comb, found the
suspicious software, took it into the laboratory, and were able to
reproduce the problem. We have since corrected the flaw, tested the
change and restored the backup signaling links.

We believe the software design, development and testing processes we use
are based on solid, quality foundations. All future releases of software
will continue to be rigorously tested. We will use the experience we've
gained through this problem to further improve our procedures.

It is important to note that Monday's calling volume was not unusual; in
fact, it was less than a normal Monday, and the network handled normal
loads on previous weekdays. Although nothing can be guaranteed 100% of
the time, what happened Monday was a series of events that had never
occurred before. With ongoing improvements to our design and delivery
processes, we will continue to drive the probability of this type of
incident occuring towards zero.

Don H Kemp, B B & K Associates, Inc., Rutland, VT
```

A.2 Analyse du Ver de Morris

```
Newsgroups: comp.risks
Subject: More on the Virus
Gene Spafford <spaf@purdue.edu>
Thu, 03 Nov 88 09:52:18 EST

All of our Vaxen and some of our Suns here were infected with the virus.
The virus forks repeated copies of itself as it tries to spread itself,
and the load averages on the infected machines skyrocketed.  In fact, it
got to the point that some of the machines ran out of swap space and
kernel table entries, preventing login to even see what was going on!

The virus seems to consist of two parts.  I managed to grab the source
code for one part, but not the main component (the virus cleans up after
itself so as not to leave evidence). The way that it works is as
```

```
follows:

1) Virus running on an infected machine opens a TCP connection to a
   victim machine's sendmail, invokes debug mode, and gets a shell.

2) The shell creates a file in /tmp named $$,l1.c (where the $$ gets
   replaced by the current process id) and copies code for a
   ``listener'' or ``helper'' program. This is just a few dozen lines
   long and fairly generic code. The shell compiles this helper using
   the ``cc'' command local to the system.

3) The helper is invoked with arguments pointing back at the infecting
   virus (giving hostid/socket/passwords as arguments).

4) The helper then connects to the ``server'' and copies a number of
   files
   (presumably to /tmp). After the files are copied, it exec's a shell
   with standard input coming from the infecting virus program on the
   other end of the socket.

From here, I speculate on what happens since I can't find the source to
this part lying around on our machines:

5) The newly exec'd shell attempts to compile itself from the files
   copied over to the target machine. I'm not sure what else the virus
   does, if anything -- it may also be attempting to add a bogus passwd
   file entry or do something to the file system. The helper program has
   an array of 20 filenames for the ``helper'' to copy over, so there is
   room to spare. There are two versions copied -- a version for Vax BSD
   and a version for SunOS; the appropriate one is compiled.

6) The new virus is dispatched. This virus opens all the virus source
   files, then unlinks the files so they can't be found (since it has
   them open, however, it can still access the contents). Next, the
   virus steps through the hosts file (on the Sun, it uses YP to step
   through the distributed hosts file) trying to connect to other
   machines' sendmail. If a connection succeeds, it forks a child
   process to infect it, while the parent continues to attempt infection
   of other machines.

7) The child requests and initializes a new socket, then builds and
   invokes a listener with the new socket number and hostid as arguments
   (#1, above).

The heavy load we see is the result of multiple viruses coming in from
multiple sites.  Since local hosts files tend to have entries for other
local hosts, the virus tends to infect local machines multiple times --
```

```
in some senses this is lucky since it helps prevent the spread of the
virus as the local machines slow down.

The virus also "cleans" up after itself.  If you reboot an infected
machine (or it crashes), the /tmp directory is normally cleaned up on
reboot.  The other incriminating files were already deleted by the virus
itself.

Clever, nasty, and definitely anti-social.

--spaf
```

A.3 Article dans le *Times* sur le Ver de Morris

Computer Network Disrupted by 'Virus'

By JOHN MARKOFF
Thu, 3 Nov 1988 N.Y. Times News Service

In an intrusion that raises new questions about the vulnerability of the nation's computers, a nationwide Department of Defense data network has been disrupted since Wednesday night by a rapidly spreading "virus" software program apparently introduced by a computer science student's malicious experiment.

The program reproduced itself through the computer network, making hundreds of copies in each machine it reached, effectively clogging systems linking thousands of military, corporate and university computers around the country and preventing them from doing additional work. The virus is thought not to have destroyed any files.

By late Thursday afternoon computer security experts were calling the virus the largest assault ever on the nation's computers.

"The big issue is that a relatively benign software program can virtually bring our computing community to its knees and keep it there for some time," said Chuck Cole, deputy computer security manager at Lawerence Livermore Laboratory in Livermore, Calif., one of the sites affected by the intrusion. "The cost is going to be staggering."

Clifford Stoll, a computer security expert at Harvard University, added : "There is not one system manager who is not tearing his hair out. It's causing enormous headaches." The affected computers carry routine communications among military officials, researchers and corporations.

While some sensitive military data are involved, the nation's most sensitive secret information, such as that on the control of nuclear weapons, is thought not to have been touched by the virus.

Computer viruses are so named because they parallel in the computer world the behavior of biological viruses. A virus is a program, or a set of instructions to a computer, that is deliberately planted on a floppy disk meant to be used with the computer or introduced when the computer is communicating over telephone lines or data networks with other computers.

The programs can copy themselves into the computer's master software, or operating system, usually without calling any attention to themselves. From there, the program can be passed to additional computers. Depending upon the intent of the software's creator, the program might cause a provocative but otherwise harmless message to appear on the computer's screen. Or it could systematically destroy data in the computer's memory.

The virus program was apparently the result of an experiment by a computer science graduate student trying to sneak what he thought was a harmless virus into the Arpanet computer network, which is used by universities, military contractors and the Pentagon, where the software program would remain undetected.

A man who said he was an associate of the student said in a telephone call to The New York Times that the experiment went awry because of a small programming mistake that caused the virus to multiply around the military network hundreds of times faster than had been planned.

The caller, who refused to identify himself or the programmer, said the student realized his error shortly after letting the program loose and that he was now terrified of the consequences. A spokesman at the Pentagon's Defense Communications Agency, which has set up an emergency center to deal with the problem, said the caller's story was a "plausible explanation of the events." As the virus spread Wednesday night, computer experts began a huge struggle to eradicate the invader.

A spokesman for the Defense Communications Agency in Washington acknowledged the attack, saying, "A virus has been identified in several host computers attached to the Arpanet and the unclassified portion of the defense data network known as the Milnet."

He said that corrections to the security flaws exploited by the virus are now being developed.

The Arpanet data communications network was established in 1969 and is designed to permit computer researchers to share electronic messages, programs and data such as project information, budget projections and research results.

In 1983 the network was split and the second network, called Milnet, was reserved for higher-security military communications. But Milnet is thought not to handle the most classified military information, including data related to the control of nuclear weapons.

The Arpanet and Milnet networks are connected to hundreds of civilian networks that link computers around the globe. There were reports of the virus at hundreds of locations on both coasts, including, on the East Coast, computers at the Massachusetts Institute of Technology, Harvard University, the Naval Research Laboratory in Maryland and the University of Maryland and, on the West Coast, NASA's Ames Research Center in Mountain View, Calif. ; Lawrence Livermore Laboratories ; Stanford University ; SRI International in Menlo Park, Calif. ; the University of California's Berkeley and San Diego campuses and the Naval Ocean Systems Command in San Diego.

A spokesman at the Naval Ocean Systems Command said that its computer systems had been attacked Wednesday evening and that the virus had disabled many of the systems by overloading them. He said that computer programs at the facility were still working on the problem more than 19 hours after the original incident.

The unidentified caller said the Arpanet virus was intended simply to "live" secretly in the Arpanet network by slowly copying itself from computer to computer. However, because the designer did not completely understand how the network worked, it quickly copied itself thousands of times from machine to machine.

Computer experts who disassembled the program said that it was written with remarkable skill and that it exploited three security flaws in the Arpanet network. [No. Actually UNIX] The virus' design included a program designed to steal passwords, then masquerade as a legitimate user to copy itself to a remote machine.

Computer security experts said that the episode illustrated the vulnerability of computer systems and that incidents like this could be expected to happen repeatedly if awareness about computer security risks was not heightened.

A.4 Arrestation de Robert T. Morris

Suspect in Virus Case

From the Philadelphia Inquirer, Sunday, November 6, 1988
(From Inquirer Wire Services)

ITHACA, N.Y. - A Cornell University graduate student whose father is a top government computer-security expert is suspected of creating the "virus" that slowed thousands of computers nationwide, school officials said yesterday. The Ivy League university announced that it was investigating the computer files of 23-year-old Robert T. Morris, Jr., as experts across the nation assessed the unauthorized program that was injected Wednesday into a military and university system, closing it for 24 hours. The virus slowed an estimated 6,000 computers by replicating itself and taking up memory space, but it is not believed to have destroyed any data. M. Stuart Lynn, Cornell vice president for information technologies, said yesterday that Morris' files appeared to contain passwords giving him un- authorized access to computers at Cornell and Stanford Universities. "We also have discovered that Morris' account contains a list of passwords substantially similar to those found in the virus," he said at a news conference. Although Morris "had passwords he certainly was not entitled to," Lynn stressed, "we cannot conclude from the existence of those files that he was responsible." FBI spokesman Lane Betts said the agency was investigating whether any federal laws were violated. Morris, a first-year student in a doctoral computer-science program, has a reputation as an expert computer hacker and is skilled enough to have written the rogue program, Cornell instructor Dexter Kozen said.

[...]

Reached at his home yesterday in Arnold, Md., Robert T. Morris, Sr., chief scientist at the National Computer Security Center in Bethesda, Md., would not say where his son was or comment on the case.

The elder Morris has written widely on the security of the Unix operating system, the target of the virus program. He is widely known for writing a program to decipher passwords, which give users access to computers.

[...]

A.5 Article sur le bogue de la '*Bank of New York*'

A Computer Snafu Snarls the Handling of Treasury Issues

by Phillip L. Zweig and Allanna Sullivan
Wall Street Journal, Monday 25 November 1985

NEW YORK- A computer malfunction at Bank of New York brought the Treasury bond market's deliveries and payments systems to a near- standstill for almost 28 hours Thursday and Friday. Although bond prices weren't affected, metal traders bid up the price of platinum futures Friday in the belief that a financial crisis had struck the Treasury bond market. However, Bank of New York's problems appeared to be more electronic than financial. The foul-up temporarily prevented the bank, the nation's largest clearer of government securities, from delivering securities to buyers and making payments to sellers - a service it performs for scores of securities dealers and other banks. The malfunction was cleared up at 12 :30 p.m. EST Friday, and an hour later the bank resumed delivery of securities. But Thursday the bank, a unit of Bank of New York Co., had to borrow a record $20 billion from the Federal Reserve Bank of New York so it could pay for securities received. The borrowing is said to be the largest discount window borrowing ever from the Federal Reserve System. Bank of New York repaid the loan Friday, Martha Dinnerstein, a senior vice president, said. Although Bank of New York incurred an estimated $4 million interest expense on the borrowing, the bank said any impact on its net income "will not be material." For the first nine months this year, earnings totaled $96.7 million. Bank of New York stock closed Friday at $45.125, off 25 cents from the Thursday, as 16,500 shares changed hands in composite trading on the New York Stock Exchange. Bank of New York said that it had paid for the cost of carrying the securities so its customers wouldn't lose any interest. Bank of New York's inability to accept payments temporarily left other banks with $20 billion on their hands. This diminished the need of many banks to borrow from others in the federal funds market. Banks use the market for federal funds, which are reserves that banks lend each other, for short-term funding of certain operations. The cash glut caused the federal funds rate to plummet to 5.5% from 8.375% early Thursday. The electronic snafu is by far the largest of computer problems that periodically have bedeviled the capital markets. Almost all goverment securities transactions are settled electronically through the New York Federal Reserve Bank. In this system, computers of clearing banks are linked to one another through a central computer, to enable banks to settle purchases and sales of securities by customers. According to Wall Street sources, the malfunction occurred at 10 a.m. Thursday as Bank of New York was preparing to change software in a computer system and begin the days operations. Until Friday afternoon, Bank of New York received billions of dollars in securities that it couldn't deliver to buyers. The Fed settlement system, which officially closes at 2 :30 p.m., remained open until 1 :30 a.m. Friday in the expectation that technicians would be able to solve the problem. Rumors about bank problems often send commodity traders scurrying to buy precious metals. In the platinum pit at the New York Mercantile Exchange, the price for January delivery surged $12.40 an ounce to $351.20 Friday on volume of 11,929 contracts, a 29-year *record. Reports that the Fed was investigating transfer problems at Bank of New York prompted the platinum buying.*

A.6 Premiere annonce de la perte de Phobos 1

```
Newgroups: comp.risks
Subject: Soviet Mars Probe
Peter G. Beumann <Neumann@KL.SRI.COM>
Tue, 13 Sep 88 15:20:18 PDT

For the "single-character" doubters:

The Soviet Mars probe was mistakenly ordered to "commit suicide" when
ground control beamed up a 20 to 30 page message in which a single
character was inadvertently omitted. The change in progam was required
because the Phobos 1 control had been transferred from a command center
in the Crimea to a new facility near Moscow. "The [changes] would not
have been required if the controller had been working the computer in
Crimea." The commands caused the spacecraft's solar panels to point the
wrong way, which would prevent the batteries from staying charged,
ultimately causing the spacecraft to run out of power.

[From the SF Chronicle, 10 Sept 88, item (page A11), thanks to Jack
Goldberg.]
```

A.7 Deuxième annonce de la perte de Phobos 1

```
Newsgroups: comp.risks
Subject: Soviets See Little Hope of Controlling Spacecraft
Gary Kremen (The Arb) <89.KREMEN@GSB-HOW.Stanford.EDU>
Sat 10 Sep 88 15:22:49-PDT

According to today's (Saturday, September 10, 1988) New York Times, the
Soviets lost their Phobos I spacecraft after it tumbled in orbit and the
solar cells lost power. The tumbling was caused when a ground controller
gave it an improper command.

This has to one of the most expensive system mistakes ever.

Gary Kremen, Stanford Graduate School of Business

[Several people reported on radio items that attributed the problem to a
console operator's single keystroke in error, which it was speculated
might have triggered the Mars probe's self-destruct signal. After the
command was sent, contact with the probe was lost completely. PGN]
```

A.8 Résumé du problème de '*Mars Pathfinder*'

Newsgroups: comp.risks
Subject: What really happened on Mars Rover Pathfinder
Mike Jones <mbj@MICROSOFT.com>
Sunday, December 07, 1997 6:47 PM

The Mars Pathfinder mission was widely proclaimed as "flawless" in the early days after its July 4th, 1997 landing on the Martian surface. Successes included its unconventional "landing" -- bouncing onto the Martian surface surrounded by airbags, deploying the Sojourner rover, and gathering and transmitting voluminous data back to Earth, including the panoramic pictures that were such a hit on the Web. But a few days into the mission, not long after Pathfinder started gathering meteorological data, the spacecraft began experiencing total system resets, each resulting in losses of data. The press reported these failures in terms such as "software glitches" and "the computer was trying to do too many things at once".

This week at the IEEE Real-Time Systems Symposium I heard a fascinating keynote address by David Wilner, Chief Technical Officer of Wind River Systems. Wind River makes VxWorks, the real-time embedded systems kernel that was used in the Mars Pathfinder mission. In his talk, he explained in detail the actual software problems that caused the total system resets of the Pathfinder spacecraft, how they were diagnosed, and how they were solved. I wanted to share his story with each of you.

VxWorks provides preemptive priority scheduling of threads. Tasks on the Pathfinder spacecraft were executed as threads with priorities that were assigned in the usual manner reflecting the relative urgency of these tasks.

Pathfinder contained an "information bus", which you can think of as a shared memory area used for passing information between different components of the spacecraft. A bus management task ran frequently with high priority to move certain kinds of data in and out of the information bus. Access to the bus was synchronized with mutual exclusion locks (mutexes).

The meteorological data gathering task ran as an infrequent, low priority thread, and used the information bus to publish its data. When publishing its data, it would acquire a mutex, do writes to the bus, and release the mutex. If an interrupt caused the information bus thread to be scheduled while this mutex was held, and if the information bus thread then attempted to acquire this same mutex in order to retrieve published data, this would cause it to block on the mutex, waiting until the meteorological thread released the mutex before it could continue.

```
The spacecraft also contained a communications task that ran with medium
priority.

Most of the time this combination worked fine. However, very
infrequently it was possible for an interrupt to occur that caused the
(medium priority) communications task to be scheduled during the short
interval while the (high priority) information bus thread was blocked
waiting for the (low priority) meteorological data thread. In this case,
the long-running communications task, having higher priority than the
meteorological task, would prevent it from running, consequently
preventing the blocked information bus task from running. After some
time had passed, a watchdog timer would go off, notice that the data bus
task had not been executed for some time, conclude that something had
gone drastically wrong, and initiate a total system reset.

This scenario is a classic case of priority inversion.

HOW WAS THIS DEBUGGED?

VxWorks can be run in a mode where it records a total trace of all
interesting system events, including context switches, uses of
synchronization objects, and interrupts. After the failure, JPL
engineers spent hours and hours running the system on the exact
spacecraft replica in their lab with tracing turned on, attempting to
replicate the precise conditions under which they believed that the
reset occurred. Early in the morning, after all but one engineer had
gone home, the engineer finally reproduced a system reset on the
replica. Analysis of the trace revealed the priority inversion.

HOW WAS THE PROBLEM CORRECTED?

When created, a VxWorks mutex object accepts a boolean parameter
that indicates whether priority inheritance should be performed by the
mutex. The mutex in question had been initialized with the parameter
off; had it been on, the low-priority meteorological thread would have
inherited the priority of the high-priority data bus thread blocked on
it while it held the mutex, causing it be scheduled with higher priority
than the medium-priority communications task, thus preventing the
priority inversion. Once diagnosed, it was clear to the JPL engineers
that using priority inheritance would prevent the resets they were
seeing.

VxWorks contains a C language interpreter intended to allow developers
to type in C expressions and functions to be executed on the fly during
system debugging. The JPL engineers fortuitously decided to launch the
spacecraft with this feature still enabled. By coding convention, the
initialization parameter for the mutex in question (and those for two
```

others which could have caused the same problem) were stored in global variables, whose addresses were in symbol tables also included in the launch software, and available to the C interpreter. A short C program was uploaded to the spacecraft, which when interpreted, changed the values of these variables from FALSE to TRUE. No more system resets occurred.

ANALYSIS AND LESSONS

First and foremost, diagnosing this problem as a black box would have been impossible. Only detailed traces of actual system behavior enabled the faulty execution sequence to be captured and identified.

Secondly, leaving the "debugging" facilities in the system saved the day. Without the ability to modify the system in the field, the problem could not have been corrected.

Finally, the engineer's initial analysis that "the data bus task executes very frequently and is time-critical -- we shouldn't spend the extra time in it to perform priority inheritance" was exactly wrong. It is precisely in such time critical and important situations where correctness is essential, even at some additional performance cost.

HUMAN NATURE, DEADLINE PRESSURES

David told us that the JPL engineers later confessed that one or two system resets had occurred in their months of pre-flight testing. They had never been reproducible or explainable, and so the engineers, in a very human-nature response of denial, decided that they probably weren't important, using the rationale "it was probably caused by a hardware glitch".

Part of it too was the engineers' focus. They were extremely focused on ensuring the quality and flawless operation of the landing software. Should it have failed, the mission would have been lost. It is entirely understandable for the engineers to discount occasional glitches in the less-critical land-mission software, particularly given that a spacecraft reset was a viable recovery strategy at that phase of the mission.

THE IMPORTANCE OF GOOD THEORY/ALGORITHMS

David also said that some of the real heroes of the situation were some people from CMU who had published a paper he'd heard presented many years ago who first identified the priority inversion problem and proposed the solution. He apologized for not remembering the precise details of the paper or who wrote it. Bringing things full circle, it

```
turns out that the three authors of this result were all in the room,
and at the end of the talk were encouraged by the program chair to stand
and be acknowledged. They were Lui Sha, John Lehoczky, and Raj Rajkumar.
When was the last time you saw a room of people cheer a group of
computer science theorists for their significant practical contribution
to advancing human knowledge? :-)
It was quite a moment.

POSTLUDE

For the record, the paper was:

L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols:
An Approach to Real-Time Synchronization. In IEEE Transactions on
Computers, vol. 39, pp. 1175-1185, Sep. 1990.
```

A.9 Article du '*Boston Globe*' sur le *Therac-25*

Man Killed by Accident with Medical Radiation

The Boston Globe, June 20, 1986
by Richard Saltos, Globe Staff

A series of accidental radiation overdoses from identical cancer therapy machines in Texas and Georgia has left one person dead and two others with deep burns and partial paralysis, according to federal investigators.
Evidently caused by a flaw in the computer program controlling the highly automated devices, the overdoses - unreported until now - are believed to be the worst medical radiation accidents to date.
The malfunctions occurred once last year and twice in March and April of this year in two of the Canadian-built linear accelerators, sold under the name Therac 25.
Two patients were injured, one who died three weeks later, at the East Texas Cancer Center in Tyler, Texas, and another at the Kennestone Regional Oncology Center in Marietta, Ga.
The defect in the machines was a "bug" so subtle, say those familiar with the cases, that although the accident occurred in June 1985, the problem remained a mystery until the third, most serious accident occurred on April 11 of this year.
Late that night, technicians at the Tyler facility discovered the cause of that accident and notified users of the device in other cities.
The US Food and Drug Administration, which regulates medical devices, has not yet completed its investigation. However, sources say that discipline or penalty for the manufacturer is unlikely.
Modern cancer radiation treatment is extremely safe, say cancer specialists. "This is the first time I've ever heard of a death" from a therapeutic rediation accident, said FDA official Edwin Miller. "There have been overtreatments to various degrees, but nothing quite as serious as this that I'm aware of."

Physicians did not at first suspect a rediation overdose because the injuries appeared so soon after treatment and were far more serious than an overexposure would ordinarily have produced.

"It was certainly not like anything any of us have ever seen," said Dr. Kenneth Haile, director of radiation oncology of the Kennestone radiation facility. "We had never seen an overtreatment of that magnitude."

Estimates are that the patients received 17,000 to 25,000 rads to very small body areas. Doses of 1,000 rads can be fatal if delivered to the whole body.

The software fault has since been corrected by the manufacturer, according to FDA and Texas officials, and some of the machines have been retured to service.

... [Description des accidents]

The Therac 25 is designed so that the operator selects either X-ray or electron-beam treatment, as well as a series of other items, by typing on a keyboard and watching a video display screen for verification of the orders.

It was revealed that if an extremely fast-typing operater inadvertently selected the X-ray mode, then used an editing key to correct the command and select the electron mode instead, it was possible for the computer to lag behind the orders. The result was that the device appeared to have made the correct adjustment but in fact had an improper setting so it focussed electrons at full power to a tiny spot on the body.

David Parnas, a programming specialist at Queens University in Kingston, Ontario, said that from a description of the problem, it appeared there were two types of programming errors.

First, he said, the machine should have been programmed to discard "unreasonable" readings - as the injurious setting presumably would have been. Second, said Parnas, there should have been no way for the computer's verifications on the video screen to become unsynchronized from the keyboard commands.

Bibliographie

[ACH94] R. Alur, C. Courcoubetis, and T.A. Henzinger. The Observational Power of Clocks. In *Proceedings of the 5^{th} International Conference on Concurrency Theory (CONCUR'94)*, volume 836 of *Lecture Notes in Computer Science*, pages 162–177, Uppsala, Sweden, August 1994. Springer-Verlag.

[AD90] R. Alur and D.L. Dill. Automata for Modeling Real-Time Systems. In *Proceedings of the 17^{th} International Colloquium on Automata, Languages and Programming (ICALP'90)*, volume 443 of *Lecture Notes in Computer Science*, pages 322–335, Warwick, England, 1990. Springer-Verlag.

[AD94] R. Alur and D.L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2) :183–235, April 1994.

[AHV93] R. Alur, T.A. Henzinger, and M.Y. Vardi. Parametric real-time reasoning. In *Proceedings of the 25^{th} ACM Symposium on Theory of Computing*, pages 592–601, 1993.

[AKV98] R. Alur, R. Kurshan, and M. Viswanathan. Membership Problems for Timed and Hybrid Automata. In *Proceedings of the 19^{th} IEEE Real-Time Systems Symposium (RTSS'98)*, Madrid Spain, December 1998.

[AL02] L. Aceto and F. Laroussinie. Is your Model-Checker on Time ? *Journal of Logic and Algebraic Programming*, pages 7–53, 2002.

[Büc62] R. Büchi. On a Decision Method in Restricted Second-order Arithmetic. In *Proceedings of the International Congress on Logic, Methodology and Philosophy of Science*, pages 1–12. Stanford University Press, 1962.

[BD00] B. Bérard and C. Dufourd. Timed Automata and Additive Clock Constraints. *Information Processing Letters*, 75(1–2) :1–7, July 2000.

[BDFP00a] P. Bouyer, C. Dufourd, E. Fleury, and A. Petit. Are timed Automata Updatable ? In *Proceedings of the 12^{th} International Conference in Computer Aided Verification (CAV'2000)*, volume 1855 of *Lecture Notes in Computer Science*, pages 464–479, Chicago, IL, United States, July 2000. Springer-Verlag.

[BDFP00b] P. Bouyer, C. Dufourd, E. Fleury, and A. Petit. Expressiveness of Updatable Timed Automata. In *Proceedings of the 25^{th} International Symposium of Mathematical Foundation of Computer Science (MFCS'2000)*, volume 1893 of *Lecture Notes in Computer Science*, pages 232–242, Bratislava, Slovakia, August 2000. Springer-Verlag.

[BDGP98] B. Bérard, V. Diekert, P. Gastin, and A. Petit. Characterization of the Expressive Power of Silent Transitions in Timed Automata. *Fundamenta Informaticae*, 36 :145–182, 1998.

[BDM+98] M. Bozga, D. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos : A Model-Checking Tool for Real-Time Systems. In A.J. Hu and M.Y. Vardi, editors, *Proceedings of the 10th International Conference on Computer Aided Verification (CAV'98)*, volume 1427 of *Lecture Notes in Computer Science*, pages 546–550, Vancouver, BC, Canada, 1998. Springer-Verlag.

[BEM97] A. Bouajjani, J. Esparza, and O. Maler. Reachability Analysis of Pushdown Automata : Application to Model-Checking. In *Proceedings of the 8th International Conference on Concurrency Theory (CONCUR'97)*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150, Warsaw, Poland, July 1997. Springer-Verlag.

[BF99] B. Bérard and L. Fribourg. Automated Verification of a Parametric Real-Time Program : the ABR Conformance Protocol. In N. Halbwachs and D. Peled, editors, *Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99)*, volume 1633 of *Lecture Notes in Computer Science*, pages 96–107, Trento, Italy, July 1999. Springer-Verlag.

[BFH+01] G. Behrmann, A. Fehnker, T. Hune, K. G. Larsen, P. Pettersson, J. Romijn, and F. Vaandrager. Minimum-Cost Reachability for Priced Timed Automata. In Maria Domenica Di Benedetto and Alberto L. Sangiovanni-Vincentelli, editors, *Proceedings of the 4th International Workshop on Hybrid Systems : Computation and Control (HSCC'01)*, volume 2034 of *Lecture Notes in Computer Science*, pages 147–161, Rome, Italy, March 2001. Springer-Verlag.

[BFKM99] B. Bérard, L. Fribourg, F. Klay, and J.F. Monin. A Compared Study of Two Correctness Proofs for the Standardized Algorithm of ABR Conformance. Research report LSV-99-7, Laboratoire Spécification et Vérification, École Normale Supérieure de Cachan, France, 1999.

[BFKM02] B. Bérard, L. Fribourg, F. Klay, and J.F. Monin. A Compared Study of Two Correctness Proofs for the Standardized Algorithm of ABR Conformance. *Formal Methods in System Design*, 2002. To appear.

[BGP96] B. Bérard, P. Gastin, and A. Petit. On the Power of Non Observable Actions in Timed Automata. In C. Puech and R. Reischuk, editors, *Proceedings of the 13th Annual Symposium on Theoretical Aspects of Computer Science (STACS'96)*, volume 1046 of *Lecture Notes in Computer Science*, pages 257–268, Grenoble, France, February 1996. Springer-Verlag.

[BJP90] G. Berthelot, C. Johnen, and L. Petrucci. PAPETRI : Environment for the Analysis of Petri Nets. In *Proceedings of the 2nd International Workshop in Computer Aided Verification (CAV'90)*, volume 531 of *Lecture Notes in Computer Science*, pages 13–22, New Brunswick, NJ, United States, June 1990. Springer-Verlag.

[BLL+96] J. Bengtsson, K.G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL – a Tool Suite for Automatic Verification of Real-Time Systems. In R. Alur, T.A. Henzinger, and E.D. Sontag, editors, *Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science*, pages 232–243. Springer-Verlag, 1996.

[BLO98] S. Bensalem, Y. Lakhnech, and S. Owre. Computing Abstraction of Infinite State Systems Compositionally and Automtically. In Hu A.J. and Vardi M.Y., editors, *Proceedings of the 10th International Conference in Computer Aided Verification (CAV'98)*, volume 1427 of *Lecture Notes in Computer Science*, pages 319–331, Vancouver, BC, Canada, June 1998. Springer-Verlag.

[Bou02a] P. Bouyer. *Modèles et Algorithmes pour la Vérification des Systèmes Temporisés.* PhD thesis, École Normale Supérieure de Cachan, Cachan, France, Avril 2002.

[Bou02b] P. Bouyer. Timed Automata May Cause Some Troubles. Technical Report LSV-02-9, Lab. Specification and Verification, École Normale Supérieure de Cachan, Cachan, France, July 2002.

[CC76] P. Cousot and R. Cousot. Static Determination of Dynamic Properties of Programs. In *Proceeding of the 2nd International Symposium on Programming*, pages 106–130, Paris, France, 1976. Dunod.

[CG00] C. Choffrut and M. Goldwurm. Timed Automata with Periodic Clock Constraints. *Journal of Automata, Languages and Combinatorics*, 5(4) :371–404, 2000.

[CJM97] S. Christensen, J. B. Jørgensen, and Kristensen L. M. Design/CPN - A Computer Tool for Coloured Petri Nets. In E. Brinksma, editor, *Proceedings of the 3rd International Workshop Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97)*, volume 1217 of *Lecture Notes in Computer Science*, pages 209–223, Enschede, Nederlands, April 1997. Springer-Verlag.

[CL00a] F. Cassez and F. Laroussinie. Model-Checking for Hybrid Systems by Quotienting and Constraints Solving. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV'2000)*, volume 1855 of *Lecture Notes in Computer Science*, pages 373–388, Chicago, IL, United States, July 2000. Springer-Verlag.

[CL00b] F. Cassez and K.G. Larsen. The Impressive Power of Stopwatches. In C. Palamidessi, editor, *Proceedings of the 11th International Conference on Concurrency Theory (CONCUR'00)*, volume 1877 of *Lecture Notes in Computer Science*, pages 138–152, University Park, PA, United States, August 2000. Springer-Verlag.

[COR+95] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A Tutorial Introduction to PVS. In IEEE Computer Society Press, editor, *Proceedings of Workshop on Industrial-Strength Formal Specification Techniques (WIFT'95)*, pages 64–78, Boca Raton, FL, United States, April 1995.

[D'A97] P.R. D'Argenio. *Algrebras and Automata for Timed and Stochastic System.* PhD thesis, University of Twente, Twente, Holland, 1997.

[DGP97] V. Diekert, P. Gastin, and A. Petit. Removing ε-transitions in Timed Automata. In R. Reischuk and M. Morvan, editors, *Proceedings of the 14th Annual Symposium on Theoretical Aspects of Computer Science (STACS'97)*, volume 1200 of *Lecture Notes in Computer Science*, pages 583–594, Lübeck, Germany, March-February 1997. Springer-Verlag.

[Dom91] E. Domenjoud. Solving Systems of Linear Diophantine Equations : an Algebraic Approach. In *Proceedings of the 16th Mathematical Foundations of Computer Science (MFCS'91)*, volume 520 of *Lecture Notes in Computer Science*, pages 141–150, Kazimierz Dolny, Poland, September 1991. Springer-Verlag.

[DOTY96] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The Tool KRONOS. In R. Alur, T.A. Henzinger, and E.D. Sontag, editors, *Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science*, pages 208–219. Springer-Verlag, 1996.

[DOY94] C. Daws, A. Olivero, and S. Yovine. Verifying ET-LOTOS Programs with KRONOS. In *Proceedings of the 7th International Conference on Formal Description

Techniques for Distributed Systems and Communications Protocols (FORTE'94), pages 227–242, Berne, Switzerland, October 1994. Chapman & Hall.

[DOY95] C. Daws, A. Olivero, and S. Yovine. The Tool Kronos : Two Case Studies : CSMA/CD and FDDI. Technical Report 95-08, VERIMAG, Grenoble, France, July 1995.

[DZ98] F. Demichelis and W. Zielonka. Controlled Timed Automata. In *Proceedings of the 9th International Conference on Concurrency Theory (CONCUR'98)*, volume 1466 of *Lecture Notes in Computer Science*, pages 455–469, Nice, France, September 1998. Springer-Verlag.

[Esp97] J. Esparza. Decidability of Model Checking for Infinite-State Concurrent Systems. *Acta Informatica*, 34(2) :85–107, 1997.

[FWW97] A. Finkel, B. Willems, and P. Wolper. A Direct Symbolic Approach to Model Checking Pushdown Systems. In *Proceedings of the 2nd International Workshop on Verification of Infinite State Systems (INFINITY'97)*, volume 9 of *Electronic Notes in Theorical Computer Science*, pages 30–40. Elsevier Science, 1997.

[Gra97] B. Grahlmann. The Reference Component of PEP. In E. Brinksma, editor, *Proceedings of the 3rd International Workshop Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97)*, volume 1217 of *Lecture Notes in Computer Science*, pages 209–223, Enschede, Nederlands, April 1997. Springer-Verlag.

[Hen96] T. A. Henzinger. The Theory of Hybrid Automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS'96)*, pages 278–292, New Brunswick, NJ, United States, July 1996. IEEE Computer Society.

[HH95] T. A. Henzinger and P.-H. Ho. A User Guide to HyTech. In A. Skou U. H. Engberg, K. G. Larsen, editor, *Proceedings of the 1st International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'95)*, volume 1019 of *Lecture Notes in Computer Science*, pages 41–71, Århus, Denmark, May 1995. Springer-Verlag.

[HHWT97] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech : A Model Checker for Hybrid Systems. *Journal of Software Tools for Technology Transfer*, 1(1–2) :110–122, 1997.

[HKC97] G Huet, G. Kahn, and Paulin-Mohring C. The COQ Proof Assistant – A Tutorial, Version 6.1. Technical Report 204, INRIA, France, August 1997.

[HSLL97] K. Havelund, A. Skou, K.G. Larsen, and K. Lund. Formal Modelling and Analysis of an Audio/Video Protocol : An Industrial Case Study Using UPPAAL. In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS'97)*, San Francisco, CA, United States, December 1997.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison–Wesley, 1979.

[JLS96] H.E. Jensen, K.G. Larsen, and A. Skou. Modelling and Analysis of a Collision Avoidance Protocol using SPIN and UPPAAL. In *Proceedings of the 2nd SPIN Workshop*, Rutgers University, NJ, United States, August 1996.

[Kos82] S. R. Kosaraju. Decidability of Reachability in Vector Addition Systems. In *Proceedings of the 14th ACM Symposium on Theory of Computing (STOC'82)*, pages 267–281, San Francisco, CA, United States, May 1982.

[KR88] B.W. Kernighan and D.M. Ritchie. *The C Programming Language (2nd Edition).* Prentice Hall, May 1988.

[LBB+01] K. G. Larsen, G. Behrmann, E. Brinksma, A. Fehnker, T. Hune, P. Pettersson, and J. Romijn. As Cheap as Possible : Efficient Cost-Optimal Reachability for Priced Timed Automata. In A. Finkel G. Berry, H. Comon, editor, *Proceedings of the 13th International Conference on Computer-Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 18–22, Paris, France, July 2001. Springer-Verlag.

[LL98] F. Laroussinie and K.G. Larsen. CMC : A Tool for Compositional Model-Checking of Real-Time Systems. In *Proceedings of Formal Description Techniques (FORTE'98) and Protocol Specification, Testing and Verification (PSTV'98)*, pages 439–456, Paris, France, November 1998.

[LPY97] K.G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1–2) :134–152, October 1997.

[LR84] G. Le Lann and P. Rolin. Procédé et dispositif pour la transmission de messages entre stations à travers un réseau local à diffusion. Technical Report (Brevet 8416957), INRIA, France, November 1984.

[LS91] K.G. Larsen and A. Skou. Bisimulation through Probabilistic Checking. *Information and Computation*, 94 :1–28, 1991.

[LT93] N. Leveson and C.S. Turner. An Investigation of the Therac-25 Accidents. *IEEE Computer*, 26(7) :18–41, July 1993.

[May84] E. W. Mayr. An Algorithm for the General Petri Net Reachability Problem. *SIAM Journal on Computing*, 13(3) :441–460, 1984.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall Int., 1989.

[Min67] M. Minsky. *Computation : Finite and Infinite Machines.* Prentice Hall Int., 1967.

[MMP91] O. Maler, Z. Manna, and A. Pnueli. From Timed to Hybrid Systems. In W. P. de Roever J. W. de Bakker, C. Huizing and G. Rozenberg, editors, *Real-Time : Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 447–484, Mook, Netherlands, June 1991. Springer-Verlag.

[MN66] R. Mc Naughton. Testing and Generating Infinite Sequences by Finite Automaton. *Information and Control*, 9 :521–530, 1966.

[Par81] D.M. Park. Concurrency on Automata and Infinite Sequences. In *Proceedings of the Conference on Category Theory and Computer Science (CTCS'81)*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer-Verlag, 1981.

[Pet62] C. A. Petri. *Kommunikation mit Automaten.* PhD thesis, Schriften des Institutes fur Instrumentelle Mathematik, Bonn, West Germany, 1962.

[PL00] P. Pettersson and K.G. Larsen. Uppaal2k. *Bulletin of the European Association for Theoretical Computer Science*, 70 :40–44, 2000.

[PM95] C. Paulin-Mohring. Circuits as Streams in COQ : Verification of a Sequential Multiplier. In Mario Coppo Stefano Berardi, editor, *Proceedings of the International Workshop on Types for Proofs and Programs (TYPES'95)*, volume 1158 of *Lecture Notes in Computer Science*, pages 216–230, Torino, Italy, June 1995. Springer-Verlag.

[PZ93] A. Pnueli and L.D. Zuck. Probabilistic Verification. *Information and Computation*, 103 :1–29, 1993.

[Rus99] J. Rushby. Systematic Formal Verification for Fault-Tolerant Time-Triggered Algorithms. *IEEE Transactions on Software Engineering*, 25(5) :651–660, September 1999.

[Rus00a] J. Rushby. From Refutation to Verification. In Tommaso Bolognesi and Diego Latella, editors, *Proceedings of the 13^{th} Conference on Formal Description Techniques (FORTE'2K) and 20^{th} Conference on Protocol Specification, Testing and Verification (PSTV'2K)*, pages 369–374, Pisa, Italy, October 2000. Kluwer Academic Publishers.

[Rus00b] J.M. Rushby. Theorem Proving for Verification. In F. Cassez, C. Jard, B. Rozoy, and M. Ryan, editors, *Proceedings of Summer School of Modelling and Verification of Parallel Processes (MOVEP'2K)*, volume 2067 of *Lecture Notes in Computer Science*, pages 24–32, Nantes, France, June 2000. Springer-Verlag.

[Sav70] W. J. Savitch. Relationships between Non-Deterministic and Deterministic Tape Complexities. *Journal of Computer and System Sciences*, 4(2) :177–192, April 1970.

[Seg95] R. Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology (MIT), Cambridge, MA, United States, 1995.

[SS99] H. Saïdi and N. Shankar. Abstract and Model-Check while you Prove. In Hu A.J. and Vardi M.Y., editors, *Proceedings of the 11^{th} International Conference in Computer Aided Verification (CAV'99)*, volume 1633 of *Lecture Notes in Computer Science*, pages 443–454, Trento, Italy, July 1999. Springer-Verlag.

[Tho90] W. Thomas. Automata on Infinite Objects. *Handbook of Theorical Computer Science*, B :133–191, 1990.

[Var85] M.Y. Vardi. Automatic Verification of Probabilistic Concurrent Finite State Programs. In *Proceedings of the 26^{th} Annual Symposium on Foundations of Computer Science*, pages 327–338, Portland, OR, United States, 1985. IEEE Computer Society Press.

[Var96] M.Y. Vardi. *An Automata-Theoretic Approach to Linear Temporal Logic*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer-Verlag, New York, NY, United States, 1996.

[Wil94] T. Wilke. Specifying Timed State Sequences in Powerful Decidable Logics and Timed Automata. In *Proceedings of the 4^{th} International School and Symposium in Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'96)*, volume 863 of *Lecture Notes in Computer Science*, pages 694–706, Uppsala, Sweden, September 1994. Springer-Verlag.

Index