École Normale Supérieure de Cachan

THÈSE

pour l'obtention du Diplôme de

Docteur de l'École Normale Supérieure de Cachan

Spécialité : INFORMATIQUE

Modèles et Algorithmes pour la Vérification des Systèmes Temporisés

Patricia Bouyer

Thèse soutenue le 5 avril 2002 devant le jury composé de :

- Claude Jard, rapporteur
- Kim G. Larsen, rapporteur
- Antoine Petit, directeur de thèse
- Olivier Roux, président du jury
- Joseph Sifakis, rapporteur
- Pascal Weil, examinateur

Laboratoire Spécification et Vérification – CNRS UMR 8643 & ENS de Cachan
61, avenue du Président Wilson – 94230 Cachan – France

« La confiance est un élément majeur : sans elle, aucun projet n'aboutit.»
Éric Tabarly

Remerciements

À mon arrivée à Cachan : « *Je ne veux pas faire d'informatique, j'en ai marre de programmer des pivots de Gauss...* »
C'était il y a plus de cinq ans...
Je remercie mes enseignants de licence d'avoir su me faire appliquer l'adage « il n'y a que les imbéciles qui ne changent pas d'avis »... ils se reconnaîtront, je pense.

Je remercie Claude Jard, Kim G. Larsen, Olivier Roux, Joseph Sifakis et Pascal Weil d'avoir accepté de faire partie de mon jury.

Je tiens à remercier très chaleureusement Antoine Petit, mon directeur de thèse, qui m'a guidée pendant plusieurs années. Je le remercie pour toute l'attention qu'il a portée à mon travail, pour sa confiance, son exigence constructive ainsi que son soutien moral.

Je voudrais aussi remercier François Laroussinie sans qui je ne saurais peut-être toujours pas ce qu'est un automate temporisé (oh non, quand même ! ! ! !). Je tiens à le remercier d'une part pour m'avoir permis de faire un stage de première année très enrichissant, d'autre part pour avoir toujours pris le temps de répondre à mes (nombreuses ?) questions et enfin pour son amitié.

I would like to thank a lot Kim G. Larsen who accepted me, more than four years ago, as a « summer student » with no much knowledge on verification and no much knowledge in english either. It was my first experience in research and it has had many consequences for my further studies.

Je tiens aussi à remercier toutes les personnes avec lesquelles j'ai travaillé, outre les personnes déjà citées, Luca Aceto, Béatrice Bérard, Augusto Burgueño Arjona, Deepak D'Souza, Catherine Dufourd, Emmanuel Fleury et Denis Thérien.

Je remercie tous mes amis cachanais, en particulier Marie pour les nombreuses rigolades, les parties de Diablo II, les tâches administratives ingrates et même la relecture d'un bout de ma thèse, Nico pour les pauses k'fé, les foots ainsi que ses précieuses aides LaTeXniques et Alex pour ses coups de gueule qui m'ont fait bien rire (c'est quand, le prochain ?). Je n'oublierai jamais les squats du 4$^{\text{ème}}$ du G ni même les foots nocturnes ou dans la neige que nous avons faits durant nos « années Cachan »... ;-) Je remercie aussi les rugueuses, tennis-ballonneuses, handballeuses cachanaises ainsi que les multiples coachs non moins cachanais pour les heures de défoulement passées ensemble. Je remercie par la même occasion ma kiné pour avoir réparé un bon nombre de blessures... Jeg er også taknemmelig for den venlige modtagelse pigerne fra mit hold i Aalborg Håndboldklub gav mig. Det har været en god morals støtte mens jeg skrev min PhD afhandling.

Je remercie tout spécialement les membres du LSV qui ont rendu mon séjour très agréable grâce notamment à leur bonne humeur et leur humour. Jeg også takker medlemer af datalogi afdelingen i Aalborg Universitetet for deres venlighed.

Enfin, merci à ma famille pour m'avoir soutenue depuis toujours, quoi qu'il arrive...

Table des matières

Première partie

Introduction

Chapitre 1

Introduction

1.1 Les enjeux

« [...]
À H0 + 36,7 secondes (environ 30 secondes après le décollage), le calculateur du système de référence inertielle de secours, qui fonctionnait en mode de veille pour la fonction de guidage et de contrôle d'altitude, est devenu inopérant. Cette panne s'explique par le fait qu'une variable interne liée à la vitesse horizontale du lanceur a dépassé une limite inscrite dans le logiciel de ce calculateur.
[...]
C'est la perte totale des informations de guidage et d'attitude 37 secondes après le démarrage de la séquence d'allumage du moteur principal (30 secondes après le décollage) qui est à l'origine de l'échec d'Ariane 501. Cette perte d'informations est due à des erreurs de spécification et de conception du logiciel du système de référence inertielle.
[...]
Les revues et essais approfondis effectués dans le cadre du programme de développement d'Ariane 5 ne comportaient pas les analyses ou essais adéquats du système de référence inertielle ou du système complet de contrôle de vol qui auraient pu mettre en évidence la défaillance potentielle.
[...] »

Le texte ci-dessus est extrait du rapport de la commission d'enquête Ariane 501 [ari96] qui a été mise en place à la suite de l'échec de la première fusée Ariane 5 en 1996. Des histoires telles que celles-ci reportant des erreurs logicielles se retrouvent malheureusement assez fréquemment dans des rapports techniques. Parmi ceux-ci nous pouvons aussi citer les défaillances de l'appareil médical Therac-25 qui, entre 1985 et 1987, a provoqué la mort de plusieurs personnes à la suite d'overdoses de radiations massives. Une étude détaillée des causes de ces accidents est arrivée à la conclusion que ces erreurs étaient dues à des problèmes logiciels [LT93].

Les deux exemples présentés ci-dessus ne sont que des illustrations parmi bien d'autres de systèmes critiques où la composante logicielle est primordiale. En effet, il n'est maintenant plus nécessaire d'insister sur l'omniprésence de l'informatique dans notre vie quotidienne (bases de données, systèmes de contrôle/commande, technologies du transport, des télécommunications...). En outre, avec l'évolution de la technologie de l'informatique, les machines ont de plus en plus de mémoire et sont de plus en plus rapides. Par conséquent, les programmes qu'il est possible de faire tourner sur ces machines sont de plus en plus gros. Ils deviennent alors très complexes et il est de plus en plus difficile de les concevoir sans erreur.

Comme nous avons pu le constater, des erreurs dans des logiciels peuvent avoir des conséquences dramatiques, aussi bien humaines qu'économiques. Depuis quelques crashs retentissants comme ceux exposés au début de cette partie, la nécessité de prévenir les bugs informatiques est devenue une évidence. Dans le rapport de la commission d'enquête pour la fusée Ariane 5, nous trouvons par exemple la recommandation suivante :

> *« [...]*
> *13. Mettre sur pied une équipe qui sera chargée d'élaborer la procédure de qualification des logiciels, de proposer des règles strictes de confirmation de la qualification, et de s'assurer que la spécification, la vérification et les essais des logiciels seront d'un niveau de qualité systématiquement élevé dans le programme Ariane 5. Envisager de faire appel à des spécialistes externes en matière de RAMS (Fiabilité, disponibilité, facilité de maintenance, sécurité).*
> *[...] »*

Dans l'article [LT93], un conseil assez similaire est donné pour le développement futur de logiciels médicaux afin que des erreurs telles que celles du Therac-25 ne se reproduisent plus :

> *« [...]*
> *The software should be subjected to extensive testing and formal analysis at the module and software level ; system testing alone is not adequate.*
> *[...] »*

Il devient crucial de disposer de méthodes pour *certifier* les logiciels dans lesquels une défaillance peut avoir des conséquences dramatiques.

1.2 Les méthodes formelles

Une des raisons de la présence d'erreurs dans les logiciels réside dans le fait que les personnes qui rédigent les cahiers des charges sont très rarement les personnes qui programment le logiciel. Or, le cahier des charges est très souvent écrit en langage naturel, ce qui est une source d'ambiguïtés, d'imprécisions voire d'omissions ou de contradictions. Une autre raison qui explique la présence d'erreurs dans les logiciels est la complexité des programmes que les machines actuelles permettent de réaliser.

Une approche souvent privilégiée pour vérifier le fonctionnement des programmes est de les tester en les faisant tourner sur des exemples (appelés *scénarios*). Cependant, cette méthode ne peut en général pas être exhaustive, il est rarement possible de tester tous les cas possibles. Le test reste quand même une composante indispensable du développement de logiciels car il permet de découvrir des erreurs.

Il apparaît crucial de disposer de méthodes de conception rigoureuses et de techniques de validation automatiques et efficaces. Celles-ci doivent être utilisées dès le début de la conception d'un logiciel. En effet, plus une erreur sera détectée tard, plus elle risque d'être coûteuse. En outre, un tel travail ne peut être réalisé que dans un cadre mathématique bien défini permettant d'éviter tous les problèmes sous-jacents à un langage non mathématique.

Les *méthodes formelles* visent à offrir un cadre mathématique permettant de décrire de manière précise et stricte les systèmes et programmes que nous voulons construire. Ce cadre formel permet de lever les ambiguïtés existant au niveau du cahier des charges et du langage naturel. Parmi ces méthodes formelles, nous pouvons citer :

– La **génération de tests** : les techniques de génération de séquences de tests consistent à calculer, à partir d'une spécification formelle du système et de la propriété à tester, un ensemble de « scénarios » intéressants permettant d'accroître la confiance que l'on peut avoir envers le comportement du logiciel [BT01].

– La **démonstration automatique** : elle permet, à partir d'un système et d'une propriété exprimée dans un même langage de spécification, de prouver que la propriété est vérifiée ou non par le système. Ceci est réalisé en utilisant des règles de déduction, comme on pourrait le faire pour montrer un théorème de mathématiques. Un démonstrateur permettant de faire une telle preuve pour chaque système et chaque propriété est bien entendu impossible à construire, mais, actuellement, des « assistants à la preuve » permettent d'écrire des preuves, l'utilisateur devant donner des indications à l'outil (voir par exemple comme points d'entrée [HKPM97, Rus01]).

– Le **model-checking** : c'est une procédure automatique qui permet de vérifier qu'un modèle d'un système vérifie une spécification. Cette procédure de vérification ne se fait pas « par déductions » comme dans le cadre de la démonstration automatique mais grâce à des algorithmes tirant profit des modèles utilisés pour le système et pour la propriété, par exemple un étiquetage d'états d'un graphe avec des sous-formules [CGP99, SBB^{+}01].

Ces méthodes n'ont pas la prétention de pouvoir certifier n'importe quel système ou programme. En effet, les techniques que nous venons de décrire s'appliquent à des modèles et pas à des systèmes réels. Ce modèles ne permettent pas toujours de représenter tout ce qui peut se passer dans la réalité : les systèmes réels sont « influencés » par un environnement non contrôlé, alors que les modèles de ces systèmes ne peuvent refléter qu'une partie minime du comportement de cet environnement. Le test classique doit toujours venir en complément de ce genre de méthode : il permet de mettre en conditions réelles les systèmes, ce que ne permettent pas le model-checking et pas toujours la démonstration automatique.

En pratique, ces méthodes formelles, plutôt d'origines académiques, commencent à se développer dans les entreprises et des outils issus du monde académique ont déjà fait leurs preuves sur de vrais exemples industriels.

1.3 Le travail du model-checking

Mon travail de thèse se situe dans le cadre du model-checking. Nous allons en décrire le principe de manière plus précise.

Supposons qu'il nous soit donné d'une part un système (sous la forme d'un programme, ou d'un cahier des charges par exemple) et d'autre part une propriété de ce système. L'étape préliminaire du model-checking consiste d'une part à *modéliser* le système, c'est-à-dire construire un objet dans le cadre formel bien défini qui « reproduit » aussi fidèlement que possible le comportement du système réel, et d'autre part à *modéliser* la propriété à vérifier dans un langage de spécification adapté. Le model-checking consiste alors à vérifier que le modèle construit pour le système vérifie la propriété exprimée dans le langage précité (d'où le terme « model-checking », ce sont les modèles que l'on vérifie). Pour pouvoir réaliser cela, il est bien entendu nécessaire de disposer d'*algorithmes de model-checking*.

La figure 1.1 schématise le principe général du model-checking.

De ce principe général se dégagent assez clairement trois grands axes de recherche complémentaires permettant d'améliorer les techniques du model-checking : le développement de modèles pour représenter les systèmes que l'on étudie ainsi que les propriétés que l'on cherche à vérifier, le

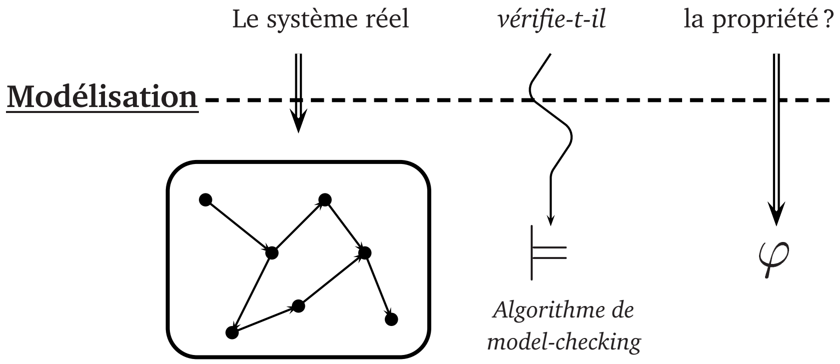


Figure 1.1: Schéma général du model-checking

développement d'algorithmes efficaces et d'outils prenant en compte les spécificités des modèles choisis, et enfin la validation des techniques développées par des études de cas.

1.3.1 Développement de modèles...

Le premier travail est de définir des classes de modèles, pour les systèmes comme pour les propriétés, adaptées à la problématique considérée. Selon la nature des systèmes que l'on cherche à valider, les modèles proposés devront tenir compte des spécificités de ces systèmes. Par exemple, si l'on cherche à développer des modèles pour représenter des protocoles de communication, il paraît assez naturel de proposer dans le modèle des facilités pour représenter des canaux de communication entre différents composants. Ceci sera peut-être moins une nécessité si l'on cherche à modéliser d'autres types de systèmes comme des systèmes de contrôle/commande.
De manière analogue, selon les propriétés que l'on cherche à vérifier, les langages de spécification ne seront pas les mêmes. Par exemple, si l'on cherche à vérifier des propriétés qui doivent être vérifiées sur toutes les exécutions du système ou bien uniquement sur une des exécutions, on n'utilisera pas les mêmes langages.

Un des critères de développement de modèles pour le model-checking est que ce dernier soit décidable, c'est-à-dire qu'il soit possible de développer des algorithmes qui « calculent » si le modèle du système vérifie ou non le modèle de la propriété. Les modèles proposés sont alors le résultat d'un compromis entre une *expressivité* permettant de décrire de nombreux systèmes de manière relativement naturelle, une *concision* permettant de les décrire de manière synthétique et de limiter le problème de l'explosion combinatoire et une *simplicité* faisant que le model-checking est au moins décidable, voire décidable avec une complexité raisonnable. La complexité du model-checking dépend tout à la fois du modèle utilisé pour représenter le système et de celui utilisé pour représenter la propriété. Comme nous le verrons par la suite, la complexité peut être très variable.

Pour certaines classes de modèles, privilégiant l'expressivité au détriment de la simplicité, le problème du model-checking est un problème indécidable. Malgré cela, ces classes de modèles peuvent parfois être utilisées : des *semi-algorithmes*, c'est-à-dire des procédures qui ne terminent pas toujours, permettent parfois d'obtenir des résultats.

1.3.2 ... et d'algorithmes

Comme nous l'avons déjà dit, les algorithmes de model-checking dépendent énormément des classes de modèles choisies. Une fois les formalismes pour les systèmes et les propriétés fixés, il faut développer des algorithmes en vue de leur implémentation. En effet, les résultats théoriques de décidabilité et de complexité obtenus lors de l'étude des modèles, ne donnent pas toujours un véritable algorithme pouvant être implémenté.

Les critères de développement de ces algorithmes sont principalement l'*efficacité* et la *facilité d'implémentation*. Bien sûr, l'efficacité est fortement liée à la complexité théorique du problème du model-checking, mais elle ne lui est pas équivalente (des algorithmes traitant un problème dans P[1] peuvent avoir des efficacités pratiques très variées, par exemple en n ou bien en n^{76}, ce qui est totalement différent...). De plus, le développement des algorithmes va de pair avec la proposition de structures de données en vue de leur implémentation. Ces structures de données doivent être relativement compactes (pour limiter les problèmes de mémoire) et doivent permettre de réaliser efficacement toutes les opérations apparaissant dans l'algorithme.

Nous faisons déjà remarquer que, en général, les problèmes de model-checking sont des problèmes d'une complexité assez élevée (pour des problèmes devant être implémentés). En outre, les systèmes sont souvent modélisés de manière modulaire (c'est-à-dire par plusieurs « petits » modèles se synchronisant entre eux). La vérification du système complet se réduit très rarement à la vérification séparée de tous les petits systèmes, il faut bien souvent faire le produit synchronisé de tous les sous-systèmes avant d'appliquer l'algorithme, ce qui augmente grandement la taille du modèle : nous parlons alors du problème de l'*explosion combinatoire* du nombre d'états. Ce problème survient régulièrement lorsque l'on veut modéliser des gros systèmes.

La dernière phase de ce processus de développement d'algorithmes est l'implémentation, en vue de la réalisation d'un outil utilisable (voir la figure 1.2). Bien sûr, même si cet outil apparaît alors comme une boîte noire où il suffit d'appuyer sur un bouton pour avoir une réponse, une bonne connaissance des bases théoriques de l'outil sont nécessaires pour utiliser au mieux toutes les possibilités qu'il offre.

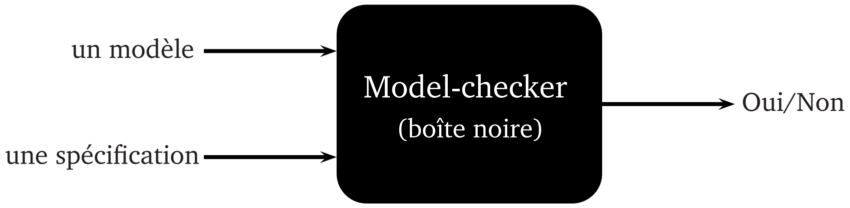


Figure 1.2: Un logiciel de vérification

1.3.3 Les études de cas

Comme l'objectif final est de vérifier de vrais systèmes, tous ces travaux de recherche doivent être validés. Depuis quelques années, de nombreux outils ont vu le jour et ont permis de réaliser de nombreuses études de cas.

Lors d'une étude de cas, la partie « modélisation » est difficile : il n'y a pas vraiment de règles générales, mais elle se fait au cas par cas. Un cahier des charges décrivant le système réel étant fourni, la première étape consiste à réaliser une première modélisation assez grossière du système : cela sert à se rendre compte quelle classe de modèles paraît bien adaptée à la représentation du système et cela permet aussi éventuellement de déjà choisir un outil qui permettra d'implémenter la modélisation. La deuxième phase consiste à faire une modélisation complète du système, en

[1] c'est-à-dire qu'ils peuvent être résolus en temps polynômial

l'adaptant éventuellement aux choix qui viennent d'être faits. Ce travail est plutôt difficile car les classes de modèles implémentées dans les outils ne permettent pas toujours d'exprimer le système en entier, mais uniquement une simplification de ce système (ce que l'on appelle une *abstraction*). Après avoir implémenté le modèle et la propriété dans l'outil choisi, théoriquement, il ne reste plus qu'à appuyer sur le bouton de l'outil pour obtenir le résultat voulu, à savoir la réponse à la question « est-ce que le modèle vérifie la spécification ? ». Cependant, les limitations dues aux machines utilisées peuvent faire échouer ce calcul (par exemple à cause d'un manque de mémoire). Il faut alors encore simplifier la modélisation (tout en préservant les caractéristiques importantes du système) et recommencer...

Avec la plupart des outils, des études de cas provenant de problèmes industriels ont été menées avec succès, ce qui a grandement motivé le développement de nouvelles techniques pour le model-checking.

1.4 Des formalismes de descriptions

Dans ce qui précède, nous avons décrit les trois grands axes de recherche qui permettent de faire progresser le model-checking. Nous constatons que l'élément central de ces trois axes est le formalisme permettant de représenter les systèmes. Du modèle choisi dépendent les algorithmes qui pourront être utilisés pour faire du model-checking. Depuis quelques années, l'intérêt pour les systèmes dans lesquels l'aspect quantitatif du temps est important s'est fortement accrû. Dans cette partie, nous allons développer cet aspect « modèles » en nous attachant plus particulièrement aux modèles temporisés, puis nous présenterons le modèle qui sera à la base de tous nos travaux.

Les systèmes de transitions, du non temporisé au temporisé. Tout système réel (par exemple un programme) peut se décrire par un *système de transitions*, c'est-à-dire un ensemble (quelconque) d'états et de transitions étiquetées par des actions entre les états. Lorsque l'on se trouve dans l'un des états d'un système de transitions, il est possible de changer d'état en effectuant une action qui étiquète l'une des transitions sortant de l'état. Une exécution dans un système de transitions est alors une séquence d'actions $(a_i)_{i\geq 0}$ qui est souvent notée sous la forme d'un *mot* $a_0a_1a_2\ldots$ représentant la succession des actions.

Les *systèmes de transitions temporisés* sont des systèmes de transitions particuliers dans lesquels deux types d'actions peuvent être distingués :
– des « vraies » actions,
– des actions d'attente qui correspondent à un écoulement du temps et non à une action visible. L'attente de d unités de temps (pour d réel quelconque) est notée $\epsilon(d)$.

Une sémantique particulière est donnée aux exécutions dans les systèmes de transitions temporisés. Une exécution $(\alpha_i)_{i\geq 0}$ va être décrite par le *mot temporisé* $(a_i, t_i)_{i\geq 0}$ où le mot $a_0a_1\ldots$ correspond au mot $\alpha_0\alpha_1\ldots$ dans lequel toutes les actions d'attente ont été effacées (c'est-à-dire que les a_i sont des « vraies » actions). Les t_i sont des réels qui représentent la date à laquelle l'action a_i a été effectuée, ce qui correspond à la somme des attentes avant l'action a_i. Par exemple, l'exécution $\epsilon(1.2)\, a\, b\, \epsilon(0.6)\, \epsilon(0.2)\, c\, \epsilon(3.1)\, d$ va être représentée par le mot temporisé $(a, 1.2)(b, 1.2)(c, 2)(d, 5.1)$.

Les systèmes de transitions, un modèle de trop bas niveau. Cependant, ces modèles très généraux ne sont pas directement utilisables en vérification car ils peuvent représenter n'importe quel système, sans restrictions. En plus, en général, il n'est pas possible de représenter de manière finie de tels systèmes, ce qui n'est pas vraiment ce que l'on cherche pour faire de la vérification. Ce dont nous avons besoin, ce sont des modèles que l'on puisse représenter de manière finie voire concise, même si, en fin de compte, la sémantique de ce modèle sera donnée par un système de transitions (temporisé ou pas). Si l'on comparait les modèles à des langages de programmation, nous dirions que les systèmes de transitions sont des langages de bas niveau, et que ce que nous cherchons, ce sont des langages de très haut niveau pour pouvoir écrire de manière simple et concise des comportements assez compliqués.

Dans le cadre non temporisé, le plus simple des modèles est sans doute celui des *automates finis* qui n'est rien d'autre que l'ensemble des systèmes de transitions ayant un nombre fini d'états. D'autres modèles sont très utilisés, comme les *réseaux de Petri* [Pet62] ou bien les *algèbres de processus* comme CCS (*Calculus of Communicating Systems*) [Mil89]. Ces modèles peuvent être décrits de manière finie, même si leurs dépliages en systèmes de transitions[2] sont généralement infinis.

Dans le cadre temporisé, des modèles « haut niveau » ont été proposés en adjoignant aux modèles précédents des notions de temps. Parmi ces modèles, nous pouvons par exemple citer les *automates temporisés* [AD90, AD94], qui proviennent du modèle des automates finis, les *réseaux de Petri temporisés* [GS94, BD99, Abd01] ou bien les *algèbres de processus temporisées*, comme TCCS, l'une des extensions temporisées de CCS [Yi90, Yi91].

Dans ma thèse, le modèle qui va servir de base à tous les travaux, est celui des automates temporisés, défini dans le début des années 1990 par Rajeev Alur et David Dill [AD90, AD94]. Nous allons maintenant le présenter de manière plus détaillée.

Les automates temporisés. Dans les automates temporisés, le temps intervient *via* des variables réelles appelées des *horloges*. Ces horloges avancent toutes à la même vitesse (celle du temps universel), leurs valeurs peuvent être comparées à des constantes ou entre elles (grâce aux *contraintes d'horloges*) et elles peuvent être remises à zéro. Le comportement d'une horloge peut être décrit par le schéma de la figure 1.3. Sur ce schéma, l'horloge x est remise à zéro deux fois, la première fois après deux unités de temps, la seconde après une nouvelle unité de temps.

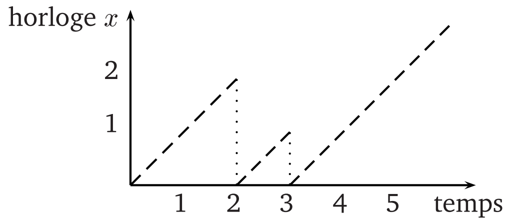


Figure 1.3: Exemple de l'évolution d'une horloge x au cours du temps

Les automates temporisés peuvent alors être décrits comme des automates finis auxquels ont été rajoutées des horloges. Il y a alors deux types d'évolution possible pour un tel système :

- soit le temps s'écoule alors que le système reste dans le même état. Les horloges avancent alors toutes d'une valeur égale à la durée d'attente.
- soit il est possible de franchir une transition et ainsi d'effectuer l'action indiquée sur celle-ci. Au préalable, il faut vérifier que les valeurs des horloges satisfont la contrainte d'horloges qui est placée sur la transition, puis, après que l'action est réalisée, il faut remettre à zéro les horloges spécifiées sur la transition.

La sémantique classique des automates finis est la même que celle des systèmes de transitions, à savoir des séquences d'actions (ou *mots*) sur un alphabet fini. Ces mots décrivent l'enchaînement des actions lors d'une exécution. Dans le cadre des automates temporisés, la sémantique classique

[2]Nous appelons « dépliage » d'un modèle un système de transitions qui a les mêmes exécutions que ce modèle.

est celle des *mots temporisés*. Comme nous l'avons dit, ces derniers associent à chaque action la date à laquelle celle-ci est effectuée. Un mot temporisé est donc une suite de couples (action,date) qui indiquent chacun quelle action est effectuée et à quelle date.

1.5 Le contenu de la thèse

Les travaux que nous présentons s'inscrivent dans le cadre de la vérification de systèmes temporisés. Comme nous l'avons déjà dit, notre modèle de base est celui des *automates temporisés*. Ces travaux peuvent être divisés en trois parties correspondant aux différents axes présentés dans la partie 1.3.

1.5.1 Étude de modèles

Étude d'une extension du modèle des automates temporisés. Dans le modèle des automates temporisés, la manipulation des horloges est assez limitée, ce qui rend difficile de représenter de manière simple certains systèmes. Par exemple, il n'est pas possible d'affecter le contenu d'une horloge à une autre horloge. Il est alors naturel de chercher à étendre cette classe d'automates pour faciliter certaines modélisations. Cependant, comme nous l'avons dit, pour qu'un modèle soit vraiment intéressant, il est préférable qu'il vérifie certaines propriétés de décidabilité. Il faut donc être assez prudent dans les extensions que l'on considère car l'indécidabilité n'est vraiment pas loin du modèle des automates temporisés. Par exemple, les automates hybrides [ACH+95] ou les machines de Minsky [Min67] sont des modèles indécidables.

Nous examinons alors dans quelles directions nous pouvons étendre le modèle des automates temporisés. Au niveau des contraintes d'horloges, le modèle des automates temporisés autorise des comparaisons du type $x \sim c$ ou $x - y \sim c$ où x, y sont des horloges, c une constante et $\sim$ un opérateur de comparaison. Malheureusement, l'introduction de contraintes du type $x + y \sim c$ rend le modèle indécidable [BD00]. Il va donc être difficile d'étendre ce modèle en autorisant des contraintes d'horloges plus compliquées. Par contre, les opérations possibles sur les horloges sont assez limitées, les seules opérations autorisées étant les remises à zéro. Nous étendons alors le modèle d'Alur et Dill en agrandissant l'ensemble des opérations possibles sur les horloges : nous définissons le modèle des *automates temporisés avec mises à jour* qui permet par exemple d'affecter de manière non déterministe à une horloge une valeur plus grande qu'une constante.

Ce modèle général est indécidable car il permet de coder une machine de Minsky, mais il contient au moins une sous-classe décidable intéressante, la classe des automates temporisés classiques. Nous verrons qu'il en contient bien d'autres et nous décrirons précisément la limite entre les sous-classes décidables et les sous-classes indécidables. Les résultats d'indécidabilité sont principalement obtenus en codant une machine de Minsky. Les propriétés de décidabilité sont obtenues en étendant la preuve faite pour les automates temporisés classiques dans [AD94]. Bien sûr, la preuve n'est valable que pour certaines sous-classes des automates temporisés avec mises à jour. Nous associons alors à chaque automate temporisé avec mises à jour un système Diophantien d'inéquations linéaires (qui dépend exclusivement des mises à jour et des contraintes d'horloges utilisées). Ce système a une solution si et seulement si l'automate appartient à une sous-classe décidable.

Nous étudions par ailleurs le pouvoir d'expression de ce modèle, en nous attachant plus particulièrement aux classes décidables (les sous-classes indécidables ayant été montrées équivalentes aux machines de Minsky donc aux machines de Turing lors de l'étude précédente). Nous montrons alors que les classes décidables des automates temporisés avec mises à jour ne sont pas plus expressives que les automates temporisés avec transitions silencieuses, mais qu'elles permettent de représenter de manière plus compacte de larges classes de systèmes temporisés. Cette concision est souvent un facteur déterminant pour vérifier des systèmes réels en raison de la complexité de la vérification de ces systèmes temporisés. Les automates temporisés avec mises à jour apparaissent alors comme un modèle de plus « haut niveau » que les automates temporisés classiques.

Fondement de la théorie des langages temporisés. Dans le cadre non temporisé, différents formalismes de natures variées permettent de définir les langages acceptés par des automates finis. En effet, les langages reconnaissables (c'est-à-dire ceux qui sont reconnus par des automates finis) peuvent être définis de manière équivalente par des *expressions rationnelles* (c'est le *théorème de Kleene* [Kle56, Büc62]), ou bien par la logique monadique du second ordre $\mathsf{MSO}(<)$ [Büc62], voire par les monoïdes finis (une présentation en est faite par exemple dans [Pin96]). Notons que les expressions rationnelles permettent de décrire aisément des langages, ce qui en fait un langage de spécification privilégié.
Toutes ces caractérisations forment le socle théorique sur lequel est basée une grande partie des travaux sur la vérification et en particulier le model-checking. À titre d'exemple, décider si un langage reconnu par un automate fini est définissable par une formule de la logique temporelle du temps linéaire LTL [Pnu77] nécessite l'utilisation de deux résultats majeurs d'équivalence entre la logique du premier ordre d'un successeur $\mathsf{FO}(<)$ et LTL d'une part [Kam68] et les langages apériodiques d'autre part [Sch65]. Ces résultats profonds et difficiles conduisent ensuite à un algorithme relativement simple : il suffit de calculer le monoïde syntaxique de l'automate considéré et de vérifier s'il est apériodique.

Ayant constaté tout ce qu'un socle théorique solide pouvait apporter à l'étude d'un modèle, il est tout à fait normal et naturel de chercher à avoir un cadre analogue pour les langages temporisés.

Les expressions rationnelles sont un formalisme simple pour spécifier des ensembles de comportements séquentiels. Le théorème de Kleene assure que le formalisme des expressions rationnelles est équivalent à celui des automates finis. Cherchant à obtenir un tel théorème dans le cadre temporisé, la première difficulté est de définir une concaténation correcte sur les mots temporisés. Il y a, de manière naturelle, deux concaténations qui peuvent être définies : l'une, partielle, permet de mettre en séquence deux mots temporisés à partir du moment où le deuxième mot commence après que le premier mot se termine ; l'autre, totale, permet de mettre en séquence deux mots temporisés en décalant le deuxième mot de la durée totale du premier. Ces concaténations ont par exemple été étudiées dans [ACM97, Asa98, ACM01] où un théorème de Kleene est proposé. Malheureusement, il nécessite l'utilisation d'opérations d'intersection et surtout de renommage, rendant ainsi le résultat assez éloigné du cadre non temporisé. L'utilisation du renommage nécessite en particulier l'écriture « d'expressions rationnelles » beaucoup plus lourdes pour spécifier un langage temporisé.

Pour résoudre ce problème et ainsi se passer de cette opération de renommage, nous proposons une nouvelle sémantique pour les automates temporisés, les *langages d'horloges*. Un mot d'horloges contient plus d'informations qu'un mot temporisé. En plus du nom de l'action et de la date à laquelle cette action est effectuée, il contient la valeur de toutes les horloges au moment où l'action est effectuée. Grâce à cette nouvelle sémantique, nous obtenons un théorème « à la Kleene/Büchi », très proche de celui existant dans le cadre des langages formels. Un langage d'horloges sera reconnu par un automate temporisé si et seulement si il est définissable à partir d'un nombre fini d'objets de base en utilisant un nombre fini d'opérateurs d'union, concaténation et itérations finie ou infinie.

Une des caractérisations les plus élégantes des langages formels est celle, de nature purement algébrique, basée sur les monoïdes finis. Comme nous l'avons expliqué juste au début de cette partie, cette caractérisation de nature théorique est à l'origine d'algorithmes simples et surprenants pour la vérification. Les travaux sur les langages sur des alphabets infinis pourraient à première vue se rapprocher de ce que nous cherchons à faire dans le cadre temporisé, les mots temporisés étant alors vus comme des mots sur un alphabet infini. Cependant, les résultats que nous avons trouvés dans la littérature [ABB80, KF94, NSV01] ne nous semblent pas pertinents lorsque l'on essaie de les adapter aux langages temporisés.

Le principe général d'un mécanisme algébrique permettant de reconnaître des langages consiste à envoyer les éléments d'un ensemble infini sur un monoïde fini. Les propriétés des éléments de l'ensemble infini initial dépendent alors uniquement de leur image dans le monoïde fini. Pour les langages reconnaissables, ce mécanisme est très simple : c'est un morphisme qui envoie chaque mot sur un élément d'un monoïde fini. Un mécanisme aussi simple n'est pas satisfaisant pour les langages temporisés, car des langages qui « devraient » être reconnus (vue leur simplicité) ne le seraient pas si l'on utilisait un simple morphisme. Ce serait par exemple le cas pour le langage temporisé $\{(a,t)(a,t+1) \mid t > 0\}$.

Nous avons alors été amenés à généraliser les langages temporisés et à définir les *langages de données*. Un ensemble de *données* est un ensemble quelconque, fini ou infini, qui peut être un domaine de temps, mais ce n'est pas nécessaire. Un mot de données est une suite de couples (action,donnée), un langage de données est un ensemble de tels mots de données. Nous proposons un formalisme purement algébrique basé lui aussi sur les monoïdes finis pour reconnaître les langages de données. Ce formalisme utilise, pour effectuer ses calculs dans le monoïde, une mémoire auxiliaire bornée qui lui permet de stocker des données.
En outre, nous définissons un modèle d'*automates de données*. Ces automates sont munis de registres (qui servent de mémoire auxiliaire) et peuvent ranger les données lues dans ces registres. Nous montrons alors que ce modèle d'automates est équivalent au formalisme algébrique que nous avons défini auparavant. Notons que la restriction de nos formalismes à un ensemble de données à un seul élément correspond exactement au cas des langages formels.
Nous montrons également en nous restreignant à un domaine de temps, que les automates de données sont plus puissants que les automates temporisés d'Alur et Dill et nous décrivons une condition sous laquelle le modèle des automates de données est un modèle décidable.

La logique est souvent un des moyens privilégiés pour modéliser des propriétés. Dans le cadre non temporisé, la logique monadique du second ordre $\mathsf{MSO}(<)$ permet d'exprimer exactement les langages acceptés par des automates finis [Büc62]. La logique du premier ordre $\mathsf{FO}(<)$ permet de décrire la sous-classe des langages apériodiques qui correspond exactement à l'ensemble des langages qui peuvent être définis par une formule de LTL [Pnu77].

Dans le cadre temporisé, Thomas Wilke a proposé un fragment d'une logique monadique du second ordre temporisée qui permet d'exprimer exactement les langages acceptés par des automates temporisés classiques [Wil94]. Afin de montrer l'intérêt des langages de données, nous étendons les résultats précédents en proposant un formalisme logique pour les langages de données équivalent aux autres formalismes proposés jusqu'à présent.

1.5.2 Des algorithmes de model-checking

Étude d'un algorithme d'analyse en avant. Comme nous l'avons rappelé dans la partie 1.3.2, les résultats de décidabilité et de complexité, même s'ils sont montrés de manière constructive, ne donnent en général pas un algorithme intéressant pour tester les modèles. Par exemple, pour les automates temporisés classiques, l'algorithme basé sur les régions, qui a permis à Alur et Dill de montrer la décidabilité du modèle, n'est pas implémenté dans les outils comme UPPAAL [BLL+98] et KRONOS [Yov97] car il est peu efficace en pratique.

L'algorithme qui nous a permis de montrer que de larges classes d'automates temporisés avec mises à jour étaient décidables n'est pas non plus voué à être implémenté. Nous proposons une extension de l'algorithme implémenté pour les automates temporisés classiques qui permet de vérifier les automates temporisés avec mises à jour. Nous montrons en détail sa correction, obtenant ainsi comme cas particulier, la correction de l'algorithme implémenté dans UPPAAL et KRONOS. Nous montrons aussi comment la structure de données des DBMs, la même que celle utilisée dans UPPAAL et KRONOS, peut être utilisée pour calculer les opérations plus compliquées qui apparaissent dans notre algorithme. De cette manière, il devrait être facile de rajouter le modèle des automates temporisés avec mises à jour aux outils existants. Il faut alors bien remarquer que la complexité de l'algorithme que nous proposons pour les automates temporisés avec mises à jour est identique à celle de l'algorithme pour les automates temporisés classiques.

Les limites de l'accessibilité. Les propriétés d'accessibilité sont, pour de nombreux systèmes, les propriétés les plus simples à vérifier. Il est alors naturel de vouloir transformer, *via* une réduction adéquate, la vérification de propriétés plus générales en la vérification de telles propriétés d'accessibilité. La réduction que nous étudions est celle des automates de test. Pour chaque propriété ϕ, nous construisons un « observateur » appelé *automate de test* et que nous notons T_ϕ. Cet automate vérifie alors que si $\mathcal{A}$ est un automate temporisé,

$$\mathcal{A} \text{ vérifie } \phi \iff \mathcal{A} \parallel T_\phi \text{ ne permet pas d'atteindre un état rejetant de } T_\phi$$

Ainsi, pour tester si $\mathcal{A}$ vérifie ϕ, il suffit de pouvoir tester l'accessibilité de certains états du système formé de la mise en parallèle de l'automate $\mathcal{A}$ avec l'observateur T_ϕ. Nous définissons alors un fragment d'un langage proche d'une logique modale temporisée et nous proposons un algorithme qui permet de construire, de manière syntaxique, pour chaque formule ϕ de notre langage, un automate de test T_ϕ comme décrit ci-dessus. Nous montrons alors que ce fragment de langage logique caractérise complètement l'ensemble des propriétés dont la vérification peut se réduire de la manière décrite ci-dessus à un test d'accessibilité. Ce résultat permet en particulier de décrire quelles propriétés peuvent être verifiées avec l'outil UPPAAL, qui, *a priori*, ne permet de vérifier directement que des propriétés d'accessibilité.

1.5.3 Une étude de cas

De manière orthogonale aux travaux précédemment présentés, nous présentons une étude de cas. Nous modélisons le protocole de communication de France Télécom R&D appelé PGM (*Pragmatic General Multicast*). Le protocole PGM est un protocole multicast de transmission de données sur un réseau non fiable. Des sources envoient des données dans le réseau à différents destinataires. Le réseau pouvant perdre des données, certains destinataires ne les reçoivent pas toutes. Ils peuvent s'en rendre compte car, régulièrement, les sources envoient sur le réseau des paquets indiquant quels sont les noms des données qui ont été envoyées. Lorsqu'un destinataire se rend compte qu'il n'a pas une donnée qu'il aurait dû recevoir, il envoie un message d'erreur aux sources. Lorsqu'une source reçoit un de ces messages d'erreur, elle peut éventuellement envoyer une réparation. Le but est alors d'étudier la correction de ce protocole, à savoir, est-ce que chaque destinataire reçoit chaque donnée ? Si ce n'est pas le cas, les sources se rendent-elles toujours compte qu'une donnée a été définitivement perdue ?

Nous proposons une modélisation simplifiée de ce protocole et nous l'implémentons dans l'outil de vérification UPPAAL. Nous vérifions ensuite certaines propriétés du protocole en utilisant le module de vérification d'UPPAAL. Cette étude permet de mettre en évidence deux problèmes dans le protocole au niveau de l'envoi des données.

1.6 Plan de la thèse

L'introduction que nous venons de faire constituait la première partie de ma thèse. La suite est organisée en trois autres grandes parties.

La deuxième partie présente les travaux réalisés sur les modèles et commence par un chapitre introductif, le **chapitre 2**. Nous y définissons précisément le cadre temporisé dans lequel nos travaux vont se placer. Nous y présentons le modèle des automates temporisés tel que défini par Alur et Dill [AD90, AD94] et énonçons les propriétés essentielles de ce modèle. Nous faisons alors un petit état de l'art des travaux qui ont été faits autour des automates temporisés puis nous situons les nôtres dans ce cadre.

Au **chapitre 3**, nous exposons les travaux sur le modèle des automates temporisés avec mises à jour. Nous présentons en premier lieu les résultats d'indécidabilité que nous obtenons par simulation d'une machine de Minsky, puis les résultats de décidabilité, basés sur une généralisation de la construction de l'automate des régions décrite par Alur et Dill. Nous étudions enfin l'expressivité de toutes les classes décidables que nous avons dégagées.

Au **chapitre 4**, nous présentons la sémantique des langages d'horloges pour les automates temporisés. Celle-ci nous permet de définir naturellement une concaténation, ainsi que des itérations finies et infinies permettant de définir des expressions rationnelles temporisées. Nous obtenons alors un théorème d'équivalence entre les langages acceptés par les automates temporisés et les langages définis *via* les expressions rationnelles temporisées, ce qui revient à obtenir un théorème « à la Kleene » pour le cadre temporisé.

Au **chapitre 5**, nous définissons la notion de langages de données et nous proposons un formalisme algébrique ainsi qu'un modèle d'automates permettant d'accepter des langages de données. Nous montrons ensuite que ces deux formalismes sont équivalents. En nous restreignant à un domaine de temps, nous replaçons ces travaux dans le cadre temporisé.

Au **chapitre 6**, nous définissons notre langage logique et montrons qu'il est équivalent au modèle des automates de données vu dans le chapitre précédent. Ceci termine la partie sur l'étude des modèles et nous permet de dessiner un parallèle entre les langages de données et les langages formels.

Dans la troisième partie de cette thèse, nous présentons des travaux sur des algorithmes de model-checking, ainsi qu'une étude de cas. Le **chapitre 7** en est un chapitre introductif. Dans celui-ci, nous revenons sur le problème du model-checking. Nous présentons en particulier plusieurs formalismes qui permettent d'exprimer des propriétés ainsi que différentes techniques de model-checking assez couramment utilisées. Ceci nous permet de placer nos travaux sur les algorithmes de model-checking dans l'ensemble des travaux existants.

Au **chapitre 8**, nous présentons notre algorithme qui permet de tester les automates temporisés avec mises à jour d'une façon analogue à celle qui est implémentée dans plusieurs outils de model-checking. Nous présentons une preuve complète de sa correction et nous décrivons une implémentation possible de cet algorithme en utilisant une structure de données assez communément utilisée.

Au **chapitre 9**, nous présentons la technique des automates de test qui permet de vérifier des propriétés grâce à de l'analyse d'accessibilité. Nous proposons un langage logique entièrement testable de cette manière et qui permet même de caractériser entièrement l'ensemble des propriétés qui peuvent être testées grâce aux automates de test.

Au **chapitre 10**, nous présentons l'étude de cas que nous avons effectuée en décrivant la modélisation faite, les expériences effectuées ainsi que les résultats obtenus.

La dernière partie ne comprend qu'un seul chapitre, le **chapitre 11**. Nous y faisons un bilan complet des travaux effectués dans cette thèse et nous présentons quelques perspectives de recherche, qui s'inscrivent en partie dans la continuité de la thèse.

Cette introduction contenait peu de références pour en faciliter la lecture. Nous avons juste donné quelques points d'entrée dans la bibliographie. Nous donnerons plus de références bibliographiques dans les chapitres 2 et 7 qui sont aussi des chapitres introductifs et qui présentent de manière plus précise le cadre de ma thèse.

Deuxième partie

Modèles

Chapitre 2

Le cadre temporisé

Dans ce chapitre introductif, nous présentons tout le formalisme qui nous sera nécessaire pour étudier les systèmes temporisés, en particulier les *automates temporisés* (parties 2.1 et 2.2) puis nous citons quelques résultats fondamentaux de ce modèle (parties 2.3 et 2.4). À la fin de cette partie, nous présentons brièvement quelques directions de travaux réalisés dans le cadre des systèmes temporisés (partie 2.5). Nous terminons ce chapitre en intégrant nos travaux sur les modèles dans les différentes catégories que nous aurons dégagées juste avant.

2.1 Préliminaires

2.1.1 Alphabet et notion de temps

Si Z est un ensemble, nous notons Z^* (respectivement Z^ω) l'ensemble des suites *finies* (respectivement *infinies*) d'éléments de Z. Souvent, nous parlerons de *mots finis* sur l'alphabet Z pour les éléments de Z^* et de *mots infinis* sur l'alphabet Z pour les éléments de Z^ω. Nous noterons de manière plus générale Z^∞ l'union de ces deux ensembles Z^* et Z^ω.

Nous noterons de manière générique $\mathbb{T}$ un domaine de temps : ce pourra donc être $\mathbb{Q}^+$, l'ensemble des nombres rationnels positifs, ou bien $\mathbb{R}^+$, l'ensemble des nombres réels positifs.

Dans tout ce qui suit, Σ représente un alphabet fini d'*actions*. En outre, ε désigne une action qui n'est pas dans Σ. Cette action est dite *silencieuse*. L'ensemble $\Sigma \cup \{\varepsilon\}$ est noté Σ_ε. Une *suite de dates* sur le domaine $\mathbb{T}$ sera une suite finie ou infinie, croissante (strictement ou pas) $\tau = (t_i)_{i \geq 1} \in \mathbb{T}^\infty$. Un *mot temporisé* $u = (a_i, t_i)_{i \geq 0}$ est un élément de $(\Sigma \times \mathbb{T})^\infty$ tel que $(t_i)_{i \geq 0}$ est une suite de dates. Nous pourrons aussi écrire u comme une paire (σ, τ) où $\sigma = (a_i)_{i \geq 1}$ et $\tau = (t_i)_{i \geq 0}$ sont des suites de même longueur (par convention, deux suites infinies seront dites de même longueur). Un mot classique représente une mise en séquence d'actions. Un mot temporisé modélise aussi une mise en séquence d'actions en donnant en plus une information sur les dates auxquelles ces actions sont effectuées.

2.1.2 Horloges et opérations sur les horloges

Nous considérons un ensemble X de variables appelées *horloges*. Ces variables prennent leurs valeurs dans le domaine de temps $\mathbb{T}$ que nous considérons. Une *valuation* des horloges de X est ainsi une application $v : X \longmapsto \mathbb{T}$. L'ensemble de toutes les valuations des horloges de X est noté $\mathbb{T}^X$. Nous écrirons aussi parfois $\mathbb{T}^n$ à la place de $\mathbb{T}^X$ (quand $|X| = n$).

Si $d \in \mathbb{T}$ et si v est une valuation, la notation $v + d$ désigne la valuation définie par $(v + d)(x) = v(x) + d$ pour tout $x \in X$. Si $v \in \mathbb{T}^X$ est une valuation et si $C \subseteq X$, nous notons $[C \leftarrow 0]v$ la valuation obtenue à partir de v en remettant les horloges de C à zéro, c'est-à-dire la valuation définie par :

$$\begin{cases} ([C \leftarrow 0]v)(x) = 0 & \text{si } x \in C \\ ([C \leftarrow 0]v)(x) = v(x) & \text{si } x \notin C \end{cases}$$

Si v est une valuation et si $C \subseteq X$, alors nous notons $v \upharpoonright C$ la restriction de v aux horloges de C. Étant donnés deux ensembles d'horloges disjoints X_1 et X_2, ainsi que deux valuations v_1 et v_2 pour les ensembles d'horloges X_1, respectivement X_2, $v_1 : v_2$ représente la valuation pour les horloges de $X_1 \cup X_2$ telle que

$$\begin{cases} (v_1 : v_2)(x) = v_1(x) & \text{si } x \in X_1 \\ (v_1 : v_2)(x) = v_2(x) & \text{si } x \in X_2 \end{cases}$$

2.1.3 Contraintes d'horloges

Si X est un ensemble d'horloges, nous définissons l'ensemble des *contraintes d'horloges* (ou tout simplement *contraintes*) sur X comme étant l'ensemble, noté $\mathcal{C}(X)$, engendré par la grammaire suivante :

$$\varphi \ ::= \ x \sim c \mid x - y \sim c \mid \varphi \wedge \varphi$$

où $x,\ y \in X$, $c \in \mathbb{Q}$ et $\sim \in \{<, \leq, =, \geq, >\}$.

Nous considérons aussi un sous-ensemble propre de ces contraintes d'horloges. Cet ensemble conserve les contraintes qui comparent une horloge et une constante, mais n'autorise plus de comparaisons entre horloges. Ces contraintes sont appelées *non diagonales*. L'ensemble des contraintes d'horloges non diagonales est noté $\mathcal{C}_{df}(X)$ et peut être décrit par la grammaire suivante :

$$\varphi \ ::= x \sim c \mid \varphi \wedge \varphi$$

où $x \in X$, $c \in \mathbb{Q}$ et $\sim \in \{<, \leq, =, \geq, >\}$.

Nous disons qu'une contrainte d'horloges est *k-bornée* (k étant un entier) si toutes les constantes qui apparaissent dans sa définition sont bornées par k.

La relation de satisfaction $\models$ pour les contraintes d'horloges est définie sur l'ensemble des valuations des horloges de X, inductivement, de la façon suivante (v est une valuation de $\mathbb{T}^X$) :

$$\begin{aligned} v \models x \sim c \quad &\iff \quad v(x) \sim c \\ v \models x - y \sim c \quad &\iff \quad v(x) - v(y) \sim c \\ v \models \varphi_1 \wedge \varphi_2 \quad &\iff \quad v \models \varphi_1 \text{ et } v \models \varphi_2 \end{aligned}$$

Une contrainte d'horloges g représente un sous-ensemble (convexe) de $\mathbb{T}^X$. De façon classique, nous noterons parfois $v \in g$ (si $v \in \mathbb{T}^X$) au lieu de $v \models g$. Le fait que les sous-ensembles de $\mathbb{T}^X$ définis par une contrainte soient convexes facilite leur manipulation. Cependant, dans la définition des contraintes d'horloges, nous aurions pu ajouter l'opérateur de comparaison $\neq$ ainsi que la disjonction $\vee$, mais les ensembles de $\mathbb{T}^X$ définis n'auraient alors pas été convexes. Grâce à des transformations semblables à celles que l'on peut effectuer dans le cadre du calcul propositionnel, nous nous serions alors ramenés au cas défini ici.

Nous allons maintenant présenter un des modèles temporisés les plus étudiés, celui des *automates temporisés*, défini par Rajeev Alur et David Dill dans le début des années 1990 [AD90], *cf* aussi [Alu91, AD94].

2.2 Les automates temporisés

Les automates temporisés sont des automates munis d'un contrôle fini et un nombre fini d'horloges. Comme nous l'avons vu auparavant, les horloges sont des variables qui évoluent au cours du temps, toutes à la même vitesse, celle du *temps universel* qui pourrait être représenté par une horloge qui n'est jamais remise à zéro. Cette horloge universelle est souvent implicite.

2.2.1 Définition du modèle à la Alur et Dill

Un *automate temporisé* est un 7-uplet $\mathcal{A} = (Q, X, \Sigma, Q_0, F, R, T)$ où
- Q est un ensemble fini d'états,
- X est un ensemble fini d'horloges,
- Σ est un alphabet fini d'actions,
- $Q_0 \subseteq Q$, $F \subseteq Q$, $R \subseteq Q$ sont respectivement les états initiaux, les états finals et les états répétés,
- $T \subseteq Q \times \mathcal{C}(X) \times \Sigma \times 2^X \times Q$ est l'ensemble des transitions.

Un *chemin* dans l'automate temporisé $\mathcal{A}$ est une suite finie ou infinie de transitions consécutives

$$q_0 \xrightarrow{g_1, a_1, C_1} q_1 \xrightarrow{g_2, a_2, C_2} \dots \xrightarrow{g_n, a_n, C_n} q_n \dots$$

Si $u = (a_1, t_1) \dots (a_n, t_n) \dots$ est un mot temporisé de $(\Sigma \times \mathbb{T})^\infty$, une *exécution* sur le chemin précédent pour le mot u est

$$(q_0, v_0) \xrightarrow[t_1]{g_1, a_1, C_1} (q_1, v_1) \xrightarrow[t_2]{g_2, a_2, C_2} \dots \xrightarrow[t_n]{g_n, a_n, C_n} (q_n, v_n) \dots$$

où $(v_i)_{i \geq 0}$ est une suite de valuations d'horloges telle que pour tout $x \in X$, $v_0(x) = 0$ et pour tout $i \geq 0$,

$$\begin{cases} v_{i+1} = [C_{i+1} \leftarrow 0](v_i + t_{i+1} - t_i) \\ v_i + t_{i+1} - t_i \models g_i \end{cases}$$

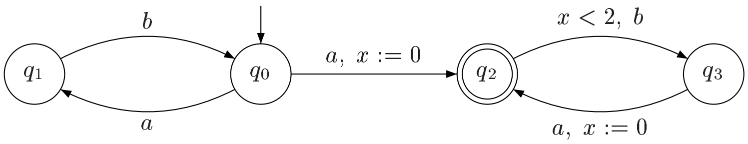
[Par convention, la date t_0 correspond à la date de début d'exécution du mot temporisé, nous supposons donc que $t_0 = 0$.]

Une telle exécution est dite *acceptante* pour u si q_0 est un état initial et
- soit le chemin est fini et se termine dans un état final,
- soit le chemin est infini et passe infiniment souvent par au moins un état répété.

Un mot u est *accepté* par l'automate $\mathcal{A}$ s'il existe une exécution acceptante pour u dans $\mathcal{A}$. Le langage accepté par $\mathcal{A}$ est l'ensemble des mots (finis ou infinis) acceptés par $\mathcal{A}$ et est noté $L(\mathcal{A})$.

Remarque : La condition d'acceptation des mots infinis est une condition de Büchi. Nous aurions pu considérer d'autres types de condition d'acceptation comme celles de Müller ou de Rabin [RS97b, PP01]. Fondamentalement, les résultats qui suivent seraient restés les mêmes.

***Exemple* 2.1** Considérons le langage $L = \{((ab)^\omega, (t_i)_{i\geq 1}) \mid \exists i,\ \forall j \geq i,\ (t_{2j} \leq t_{2j-1} + 2)\}$. Il est accepté par l'automate temporisé suivant.



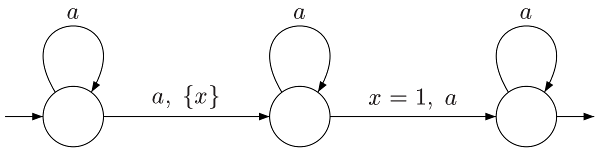
Automates temporisés avec ε-transitions. De la même manière, nous définissons un *automate temporisé avec ε-transitions* sur l'alphabet Σ comme étant un automate temporisé sur l'alphabet $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$. Les transitions de la forme $q \xrightarrow{g,\ \varepsilon,\ C} q'$ sont appelées des *ε-transitions*. Les mots acceptés par ces automates sont *a priori* des mots sur l'alphabet $(\Sigma_\varepsilon \times \mathbb{T})^\infty$, mais comme ε est une « action silencieuse », nous réduirons ces mots temporisés à des mots temporisés sur l'alphabet $(\Sigma \times \mathbb{T})^\infty$ en effaçant simplement les lettres de la forme (ε, t). Par exemple, le mot $(a, 0.2)(b, 1.7)(\varepsilon, 2)(\varepsilon, 2.3)(a, 2.5)(\varepsilon, 2.6)$ se réduit au mot $(a, 0.2)(b, 1.7)(a, 2.5)$. Le langage accepté par un automate temporisé avec ε-transitions est l'ensemble des mots temporisés de $(\Sigma \times \mathbb{T})^\infty$ qui sont obtenus de cette manière à partir d'un mot de $(\Sigma_\varepsilon \times \mathbb{T})^\infty$ lu dans l'automate.

2.2.2 Propriétés de clôture des automates temporisés

Les langages acceptés par les automates temporisés vérifient les propriétés de clôture suivantes :

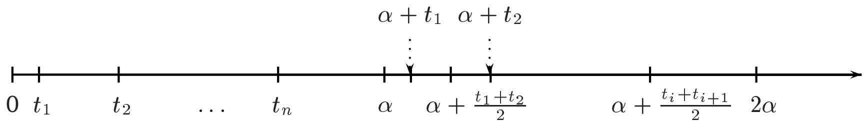
Proposition 2.2 ([AD90, AD94]) *L'ensemble des langages acceptés par des automates temporisés est clos par union et par intersection. Par contre, cet ensemble n'est pas clos par passage au complémentaire.*

***Exemple* 2.3** Le langage $\{(a, t_1) \ldots (a, t_n) \mid \exists i < j.\ t_j = t_i + 1\}$ sur l'alphabet $\Sigma = \{a\}$ est accepté par l'automate temporisé



Par contre, son complémentaire L n'est reconnu par aucun automate temporisé. La preuve de ce fait n'est pas décrite dans [AD94], nous en détaillons une ici.

Preuve : Supposons que L soit accepté par l'automate temporisé $\mathcal{A}$. Comme l'alphabet sur lequel nous travaillons est réduit au singleton $\{a\}$, nous pouvons supposer que $\mathcal{A}$ accepte non plus des mots temporisés, mais des séquences de dates. Soit c la plus petite constante non nulle apparaissant dans les contraintes de $\mathcal{A}$. Nous notons α le minimum entre 1 et c. Prenons une séquence de dates $0 < t_1 < t_2 \cdots < t_n < \alpha$ acceptée par $\mathcal{A}$.



La séquence de dates $\left(t_1, \cdots t_n, \alpha + \left(\frac{t_1+t_2}{2}\right), \cdots, \alpha + \left(\frac{t_{n-1}+t_n}{2}\right)\right)$ (voir le schéma ci-dessus) est aussi acceptée par $\mathcal{A}$ (par définition de $L = L(\mathcal{A})$). Lorsque nous considérons une exécution qui accepte cette séquence de dates, la contrainte qui correspond à la date $\alpha + \left(\frac{t_{i-1}+t_i}{2}\right)$ fait intervenir un certain nombre d'horloges, toutes ne participant pas à une contrainte « utile ». Celles qui font partie d'une contrainte utile sont des horloges qui ont été mises à zéro soit au tout début, soit lors d'une transition pour t_j car les horloges mises à zéro à la date $\alpha + \left(\frac{t_{j-1}+t_j}{2}\right)$ ne peuvent pas avoir d'influence : la contrainte qui leur correspond est obligatoirement triviale car $|\alpha + \left(\frac{t_{i-1}+t_i}{2}\right) - \alpha - \left(\frac{t_{j-1}+t_j}{2}\right)| < \alpha \leq c$ et c est la plus petite constante apparaissant dans l'automate. Les horloges utiles à la transition $\alpha + \left(\frac{t_{i-1}+t_i}{2}\right)$ ont donc leur valeur fixée par leur valeur en α. Ces horloges utiles définissent pour chaque transition intervenant dans l'exécution considérée un intervalle contenant chacun un $\alpha + \left(\frac{t_{j-1}+t_j}{2}\right)$. Notons A_j cet intervalle ; il doit être inclus dans l'intervalle $]\alpha + t_{j-1}; \alpha + t_j[$ car sinon, l'automate accepterait soit le mot $\left(t_1, \cdots, t_n, \alpha + \left(\frac{t_1+t_2}{2}\right), \cdots, \alpha + t_{j-1}\right)$

soit le mot $\left(t_1, \cdots, t_n, \alpha + \left(\frac{t_1+t_2}{2}\right), \cdots, \alpha + t_j\right)$, ce qui n'est pas possible. Chaque A_j est contenu dans un intervalle I_j avec les I_k deux à deux disjoints. Or, pour toute exécution, le nombre d'intervalles A_j est bornée par $2^x.t$ où x est le nombre d'horloges de l'automate et t son nombre de transitions. Donc, il suffit de choisir n plus grand que $2^x.t$ pour obtenir une contradiction (il y aura plus d'intervalles A_j que la borne supérieure autorisée, donc deux A_j seront égaux, ce qui n'est pas possible car les I_k sont deux à deux disjoints). ☐

2.2.3 Les automates temporisés déterministes

Un automate temporisé $\mathcal{A} = (Q, X, \Sigma, Q_0, F, R, T)$ est dit *déterministe* si l'ensemble Q_0 des états initiaux est réduit à un seul élément et si pour tout triplet $(q, v, a) \in Q \times \mathbb{T}^X \times \Sigma$, il existe au maximum un triplet $(g, C, q') \in \mathcal{C}(X) \times 2^X \times Q$ tel que $v \in g$ et $q \xrightarrow{g,a,C:=0} q'$ soit une transition de $\mathcal{A}$. Si $\mathcal{A}$ est un automate temporisé déterministe, pour tout mot temporisé u, il y a au maximum un chemin sur lequel il y a une exécution pour u.

La classe des automates temporisés déterministes a de meilleures propriétés de clôture que la classe originale :

Proposition 2.4 ([AD90, AD94]) *La classe des langages temporisés acceptés par un automate temporisé déterministe est close par union, intersection et par passage au complémentaire.*

Malheureusement, la classe des automates temporisés déterministes est beaucoup moins expressive que celle des automates temporisés classiques.

***Exemple* 2.5** Le langage $\{(a^\omega, (t_i)_{i\geq 0}) \mid \exists i < j.\, t_j = t_i + 1\}$ est reconnu par un automate temporisé classique (voir l'exemple 2.3), mais n'est reconnu par aucun automate temporisé déterministe.

2.3 Le problème du vide

2.3.1 Description du problème

Une fois que les systèmes sont modélisés, le problème de tester le vide du langage accepté par le modèle est fondamental. En effet, le problème de l'accessibilité d'un état (c'est-à-dire tester s'il existe une exécution dans le modèle qui permet d'atteindre un état donné) est strictement équivalent à la non-vacuité du langage accepté par ce même modèle en prenant comme état final l'état dont on cherche à tester l'accessibilité. Par exemple, pour assurer une propriété d'exclusion mutuelle, il faut tester que deux processus ne peuvent pas aller simultanément dans leur section critique, ce qui revient à tester l'accessibilité de l'un des états du système où deux processus sont simultanément en section critique. Le problème de tester le vide d'un langage accepté par un automate est appelé de manière plus synthétique le *problème du vide*. Nous disons alors qu'une classe de modèles est *décidable* si le problème du vide est décidable pour chacun de ces modèles.

Nous allons maintenant présenter l'étude de la décidabilité des automates temporisés faite par Alur et Dill dans [AD90, AD94].

2.3.2 Décidabilité des automates temporisés

Le résultat suivant est fondamental :

Théorème 2.6 ([AD90, AD94]) *La classe des automates temporisés est décidable.*

Étant donné un mot temporisé de $(\Sigma \times \mathbb{T})^\infty$, en effaçant toutes les dates, nous obtenons un mot de Σ^∞. De cette manière, nous pouvons associer à chaque langage temporisé L un langage classique sur l'alphabet Σ. Ce langage est noté $\mathsf{Untime}(L)$ et est défini formellement par :

$$\mathsf{Untime}(L) = \{a_1 \dots a_n \cdots \mid \exists t_1 \dots t_n \dots \text{ t.q. } (a_1, t_1) \dots (a_n, t_n) \cdots \in L\}.$$

La preuve du théorème 2.6 repose alors sur la construction, à partir d'un automate temporisé $\mathcal{A}$, d'un automate classique $\mathcal{B}$ qui reconnaît le Untime du langage accepté par l'automate $\mathcal{A}$.
Or, comme pour tout langage temporisé nous avons bien évidemment

$$L = \varnothing \iff \mathsf{Untime}(L) = \varnothing,$$

nous obtenons que :

« le langage accepté par $\mathcal{A}$ est vide si et seulement si le langage accepté par $\mathcal{B}$ est vide »

Ainsi, comme le test du vide est décidable pour les automates finis (ou de Büchi) [HU79], il est aussi décidable pour les automates temporisés.

Nous allons décrire le principe de la preuve de ce théorème. Tout d'abord, grâce au lemme suivant, nous pouvons nous restreindre aux constantes entières :

Lemme 2.7 *Soit $\mathcal{A}$ un automate temporisé. Si $\lambda \in \mathbb{Q}^+$, nous définissons l'automate $\lambda\mathcal{A}$ comme étant le même automate que $\mathcal{A}$ sauf que les constantes apparaissant sur les transitions (au niveau des contraintes) sont multipliées par λ. Soit $u = (a_1, t_1) \dots (a_n, t_n) \dots$ un mot temporisé. Nous avons alors que :*

$$u \in L(\mathcal{A}) \iff \lambda.u \in L(\lambda\mathcal{A})$$

si $\lambda.u$ représente le mot temporisé $(a_1, \lambda t_1) \dots (a_n, \lambda t_n) \dots$ (toutes les dates sont multipliées par λ).

Ainsi, pour tester le vide d'un automate temporisé, nous pouvons multiplier toutes les constantes apparaissant sur les transitions, sans changer le résultat du test du vide. En particulier, si nous prenons comme facteur multiplicatif le « plus petit commun multiple » (*i.e.* le ppcm) des dénominateurs de toutes les constantes (qui sont par hypothèse toutes rationnelles) apparaissant dans l'automate.

Nous pouvons aussi supposer que les contraintes utilisées sont non diagonales. En effet, nous avons le résultat suivant :

Lemme 2.8 *Soit $\mathcal{A}$ un automate temporisé. Il existe un automate temporisé $\mathcal{A}'$ avec uniquement des contraintes non diagonales tel que $L(\mathcal{A}) = L(\mathcal{A}')$.*

Une preuve complète de ce lemme peut par exemple être trouvée dans [BDGP98].

Grâce aux deux lemmes qui précèdent, nous restreignons notre étude aux automates temporisés avec contraintes non diagonales et constantes entières.

2.3.3 Le graphe des régions

Soit $\mathcal{A} = (Q, X, \Sigma, Q_0, F, R, T)$ un automate temporisé classique avec des constantes entières et des contraintes non diagonales. Pour toute horloge $x \in X$, nous notons $\max_x$ la plus grande constante c telle qu'une contrainte d'horloges $x \sim c$ apparaisse dans $\mathcal{A}$.

L'*équivalence des régions* est définie sur $\mathbb{T}^X$ par :

$$v \equiv v' \iff \begin{cases} \mathsf{Ent}(v(x)) = \mathsf{Ent}(v'(x)) \text{ ou } (v(x) > \max_x \text{ et } v'(x) > \max_x) \\ \text{si } (v(x) \leq \max_x \text{ et } v(y) \leq \max_y) \\ \qquad\qquad \text{alors } \Big[(\mathsf{frac}(v(x)) < \mathsf{frac}(v(y)) \iff \mathsf{frac}(v'(x)) < \mathsf{frac}(v'(y))) \Big] \end{cases}$$

où pour tout réel α, $\mathsf{Ent}(\alpha)$ représente la partie entière de α alors que $\mathsf{frac}(\alpha)$ représente la partie fractionnaire de α.

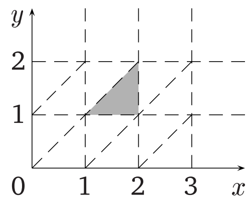
La relation $\equiv$ est une relation d'équivalence. Elle vérifie la propriété suivante :

$$v \equiv v' \implies \begin{cases} (i) \text{ pour toute contrainte } g \text{ de } \mathcal{A},\ v \models g \iff v' \models g \\ (ii)\ \forall t \in \mathbb{T},\ \exists t' \in \mathbb{T} \text{ tel que } v + t \equiv v' + t' \end{cases}$$

La première propriété ci-dessus indique que l'équivalence $\equiv$ est compatible avec les contraintes de l'automate $\mathcal{A}$ alors que la deuxième propriété indique que l'équivalence $\equiv$ est compatible avec l'écoulement du temps. Il est facile de voir que l'équivalence des régions est d'index fini. Une classe de $\mathbb{T}^X/_{\equiv}$ est appelée une *région*.

***Exemple* 2.9** Considérons un automate temporisé pour lequel $\max_x = 3$ et $\max_y = 2$. L'ensemble des régions associé à cet automate peut être décrit par le schéma de la figure de droite. La région grisée est définie par la contrainte

$$1 < x < 2 \wedge 1 < y < 2 \wedge x > y$$



Il serait aussi possible de construire un ensemble de régions directement pour un automate temporisé avec des contraintes générales, sans utiliser le lemme 2.8. La différence réside dans le fait qu'il faudrait rajouter des droites diagonales dans le dessin même au-delà des constantes maximales. Nous ne présentons pas de tels ensembles de régions ici, nous en verrons des exemples plus tard, dans le chapitre 3.

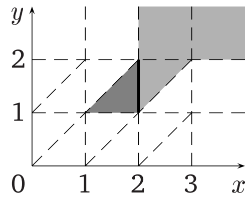
2.3.4 L'automate des régions

Nous avons vu que l'équivalence des régions est compatible avec l'écoulement du temps. Il est donc possible de définir une fonction successeur sur l'ensemble des régions. Si R est une région, nous notons $\mathsf{Succ}(R)$ l'ensemble des successeurs de R pour l'écoulement du temps, c'est-à-dire l'ensemble de régions $\mathsf{Succ}(R)$ vérifiant la propriété suivante :

$$R' \in \mathsf{Succ}(R) \iff \exists v \in R.\ \exists t \in \mathbb{T} \text{ tels que } v + t \in R'$$

***Exemple* 2.10** Reprenons l'exemple 2.9.

Le premier successeur de la région en gris foncé est la région dessinée par une ligne épaisse noire. L'ensemble des autres successeurs de cette même région est dessiné en gris clair.



Nous sommes maintenant en mesure de définir un automate fini $\mathcal{B} = (Q', \Sigma, q_0', F', R', T')$ de la manière suivante :

- $Q' = Q \times \mathbb{T}^X/_{\equiv}$, $q_0' = (q_0, \mathbf{0})$, $F' = F \times \mathbb{T}^X/_{\equiv}$, $R' = R \times \mathbb{T}^X/_{\equiv}$,
- $T' = \left\{ (q, R) \xrightarrow{a} (q', R') \;\middle|\; \begin{array}{l} \exists q \xrightarrow{g,a,C} q' \in T \text{ et } \exists R'' \in \mathsf{Succ}(R) \\ \text{tel que } R \subseteq g \text{ et } R' = R''[C \leftarrow 0] \end{array} \right\}$

Cet automate est appelé l'*automate des régions* associé à $\mathcal{A}$.

***Exemple* 2.11 ([AD94])** Nous traitons un exemple de construction de l'automate des régions. Considérons donc l'automate $\mathcal{A}$ dessiné sur la figure 2.1.

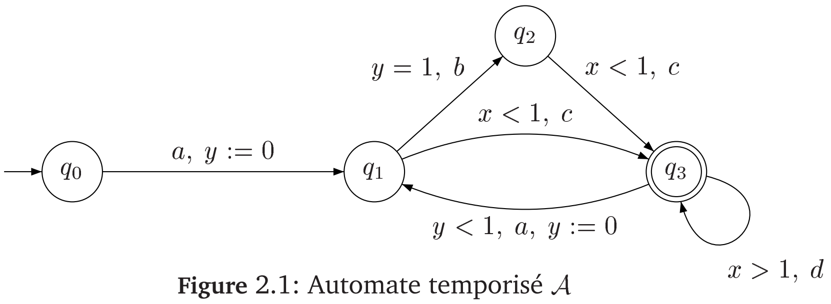


Figure 2.1: Automate temporisé $\mathcal{A}$

L'automate des régions que nous obtenons à partir de $\mathcal{A}$ en prenant comme constantes maximales 1 pour les deux horloges x et y est dessiné sur la figure 2.2.

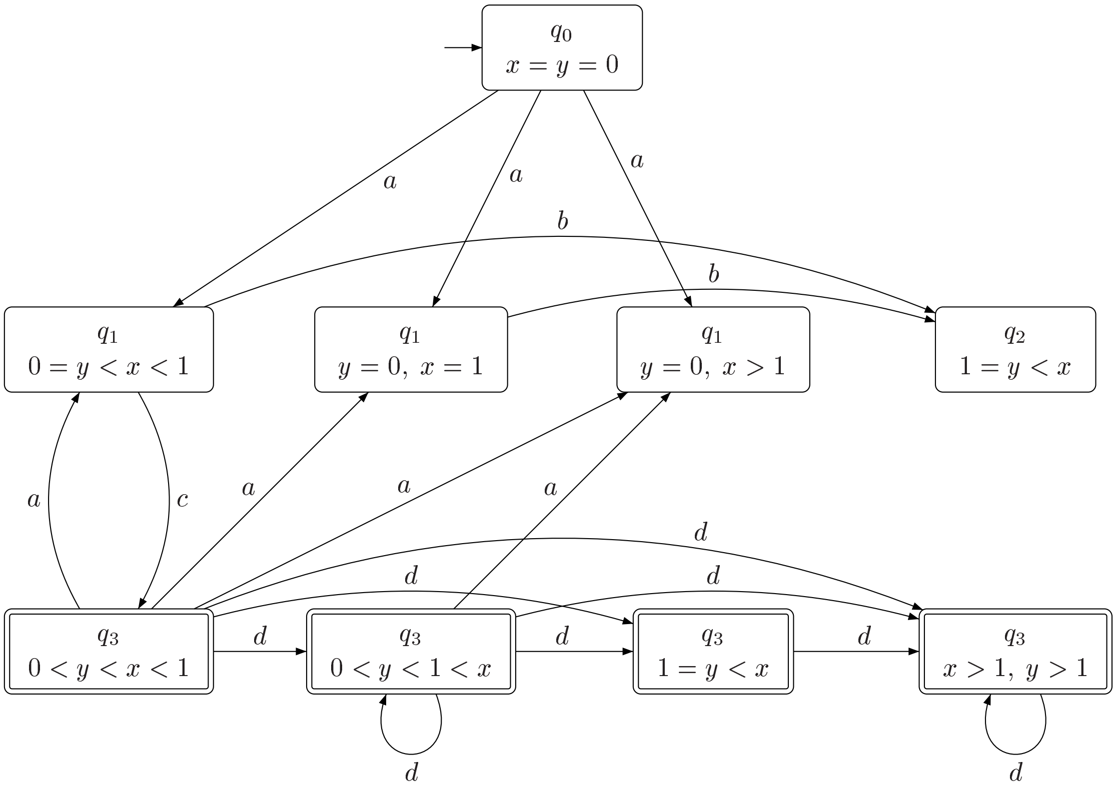


Figure 2.2: Automate des régions associé à $\mathcal{A}$ (figure 2.1)

L'automate des régions vérifie la propriété suivante :

Propriété 2.12 *Le langage accepté par $\mathcal{B}$ correspond au* Untime *du langage accepté par $\mathcal{A}$.*

Nous avons alors le théorème plus précis suivant :

Théorème 2.13 ([AD94]) *Tester le vide d'un langage accepté par un automate temporisé est un problème* PSPACE*-complet.*

2.4 Le problème universel

Ce problème consiste, étant donné un automate temporisé $\mathcal{A}$, à tester si $L(\mathcal{A}) = (\Sigma \times \mathbb{T})^\infty$. La non clôture par passage au complémentaire des langages reconnus par des automates temporisés fait que le problème universel ne peut pas se ramener au problème du vide. Il est alors d'une toute autre difficulté :

Théorème 2.14 ([AD94]) *Le problème universel est indécidable pour les automates temporisés.*

Ce théorème a des conséquences importantes, par exemple le corollaire suivant :

Corollaire 2.15 ([AD94]) *Étant donnés deux automates temporisés $\mathcal{A}$ et $\mathcal{B}$, le problème de savoir si $L(\mathcal{A}) \subseteq L(\mathcal{B})$ est un problème indécidable.*

PREUVE : Ce corollaire est facile à prouver en utilisant le théorème précédent. Nous allons montrer que si ce problème était décidable, alors le problème universel serait aussi décidable. En effet, considérons un automate temporisé $\mathcal{A}$ et $\mathcal{A}_u$ un automate qui accepte le langage universel, c'est-à-dire $(\Sigma \times \mathbb{T})^\infty$. Alors,

$$L(\mathcal{A}_u) \subseteq L(\mathcal{A}) \iff \mathcal{A} \text{ est universel } (\textit{i.e. } L(\mathcal{A}) = (\Sigma \times \mathbb{T})^\infty)$$

□

Remarque : Par contre, la classe des langages acceptés par des automates temporisés déterministes est close par passage au complémentaire. Pour ces langages, le problème d'inclusion de langages se ramène au problème du vide *via* la transformation

$$L \subseteq L' \iff L \cap L'^c = \varnothing,$$

il est donc décidable.

Le résultat négatif du corollaire 2.15 limite l'utilisation des automates temporisés pour des problèmes de vérification. Il montre qu'il n'est pas possible de tester si un modèle représenté par un automate temporisé vérifie une propriété donnée par un autre automate temporisé. L'utilisation des négations dans les logiques ne pourra également pas se traiter par passage au complémentaire. Comme nous l'avons déjà expliqué, le modèle des automates temporisés est tout de même largement utilisé pour la vérification des systèmes temporisés.

2.5 Un petit tour des travaux existants

Comme nous l'avons déjà expliqué, le modèle des automates temporisés est largement utilisé pour la vérification des systèmes temporisés.
Après avoir rappelé les résultats de base et fondamentaux de [AD90, AD94], nous allons décrire brièvement certains travaux qui ont été faits à partir du modèle des automates temporisés. Pour simplifier la présentation, nous les avons classés selon leur caractéristique principale. Nous n'avons pas la prétention de faire une bibliographie exhaustive, mais nous avons indiqué à chaque fois les points d'entrée principaux de cette bibliographie.

2.5.1 Travaux sur le modèle d'Alur et Dill

Comme nous l'avons dit dans ce qui précède, le modèle des automates temporisés a été défini par Rajeev Alur et David Dill dans leur article [AD90]. En 1994, ils ont publié une version longue de leur travail dans [AD94]. Depuis, Rajeev Alur a publié une version « tutorial » de tous ces travaux dans [Alu99]. Ces différents articles présentent le modèle des automates temporisés en exposant les différentes propriétés de clôture que cette classe possède, ainsi que les résultats de décidabilité dont nous avons parlé.

Rajeev Alur et David Dill ne sont pas les seuls à s'être intéressés à cette classe d'automates. Voici quelques travaux qui ont été réalisés sur le modèle des automates temporisés lui-même. Le pouvoir des horloges dans les automates temporisés a par exemple été étudié. Il a été montré dans [HKWT95] que le nombre d'horloges augmente le pouvoir d'expression des modèles. Ceci a été étudié. Dans [DY96], un algorithme est proposé pour minimiser le nombre d'horloges dans un automate donné. Ceci est fait *via* la notion d'*horloges actives*.

Cependant, le modèle des automates temporisés n'est pas accepté comme **le** modèle idéal (ou canonique) temporisé, car certains problèmes subsistent :

(i) les automates temporisés non déterministes sont nettement plus expressifs que les automates temporisés déterministes,

(ii) le problème universel est indécidable pour la classe générale des automates temporisés,

(iii) la classe d'automate temporisé générale n'est pas close par passage au complémentaire, ce qui rend plus difficile son utilisation en vérification.

C'est pour ces raisons que l'étude du modèle des automates temporisés ne s'arrête pas là, mais que des sous-classes sont étudiées pour ne plus être confrontés aux problèmes (ii) et (iii), tout en restant plus expressif que le modèle déterministe. Des extensions du modèle général sont aussi étudiées dans l'optique de conserver la décidabilité du vide, mais en augmentant l'expressivité ou bien la compacité de représentation (dans un but de vérification).

2.5.2 Étude de sous-classes intéressantes

Plusieurs travaux ont eu pour but d'étudier des sous-classes des automates temporisés qui auraient des propriétés de clôture et/ou de décidabilité plus robustes.

– L'article [AH92] introduit la notion d'automates temporisés à deux sens, bornés (« *two-way bounded timed automata* »). Ces classes d'automates sont des sous-classes des automates temporisés classiques. L'union des langages acceptés par des automates appartenant aux classes déterministes correspond à un ensemble de langages clos par toutes les opérations booléennes. En outre, le problème du vide et le problème universel sont tous les deux des problèmes PSPACE-complets pour cette nouvelle classe.

– Dans l'article [AFH94], d'autres classes d'automates sont définies, les « *event-clock automata* » avec horloges « *predicting* » ou « *recording* ». Les sous-classes déterministes de ces classes ne sont pas strictement moins expressives que la version non déterministe (il existe un algorithme de déterminisation). Elles sont également closes pour toutes les opérations booléennes et le problème du vide ainsi que le problème universel sont décidables (et même PSPACE). Ces classes d'automates sont de plus incluses dans l'ensemble des automates temporisés.

– Motivée par des aspects logiques, une variante des « event-clock automata », les « *state-clock automata* » est proposée dans [RS97a]. Ces automates vérifient à peu près les mêmes propriétés que le modèle des « event-clock automata ». Dans ce cadre, une logique, pour laquelle le model-checking et la satisfaisabilité sont décidables, a été proposée. En outre, toute formule de cette logique peut être transformée en un « state-clock automaton » qui reconnaît les mêmes modèles. Ces travaux ont été prolongés dans [HRS98] où plusieurs autres types de logiques ainsi qu'un modèle récursif d'« event-clock automata » sont présentés. Tous ces modèles sont comparés et classés selon leurs propriétés de décidabilité.

2.5.3 Extensions du modèle original

De nombreux travaux ont été réalisés sur des extensions des automates temporisés, parmi lesquels :

– l'étude des **actions silencieuses** : elles ont été étudiées en détails dans les articles [BGP96, DGP97, BDGP98]. Le modèle avec actions silencieuses est strictement plus expressif que celui sans action silencieuse. Par exemple, le langage accepté par l'automate temporisé avec ε-transitions de la figure 2.3 n'est accepté par aucun automate temporisé sans ε-transitions.

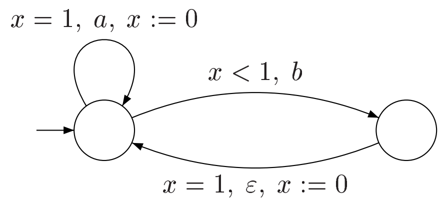


Figure 2.3: Aucun automate sans ε-transitions ne reconnaît le même langage

– l'ajout des **contraintes d'horloges du type** $x+y \sim c$: elles ont été étudiées dans [Duf97, BD00]. Le modèle d'automates avec au moins quatre horloges qui autorise ce type de contraintes est indécidable.

– l'ajout de **contraintes d'horloges périodiques** (du type $x \sim c \mod k$) : elles ont été étudiées dans [CG00]. Le modèle d'automates autorisant ces contraintes est strictement équivalent au modèle classique.

– l'ajout de **paramètres** dans les contraintes d'horloges : ce modèle a été étudié dans [AHV93]. Le modèle avec au moins trois paramètres est indécidable, celui avec un seul paramètre est décidable, nous ne savons rien pour le modèle avec deux paramètres. Un raffinement de ces travaux a été proposé dans [HRSF01].

– les **systèmes hybrides** : les variables utilisées ne sont plus uniquement des horloges mais peuvent avoir des pentes différentes de 1. Ces modèles ont été largement étudiés [ACHH93, HK94, HKPV95], il nous est impossible de faire une liste exhaustive de tous les travaux sur ce sujet. Des cas particuliers ont aussi été considérés, par exemple dans [DZ98], une sous-classe assez restreinte des automates hybrides, mais incluant les automates temporisés, a été montrée décidable. Dans [CL00b], il a été montré que les automates hybrides linéaires ne sont pas plus expressifs que les automates hybrides linéaires avec des pentes 0/1.

2.5.4 Aspects de modélisation

Comme nous l'avons dit dans l'introduction, le travail de vérification réclame d'avoir des modèles « fonctionnels », c'est-à-dire facilitant la phase de modélisation. Apparaît alors l'importance d'avoir des « macros » permettant de représenter aisément certains phénomènes. La partie précédente a traité en partie ce type de problématique. Cependant, cela n'est pas encore vraiment suffisant. Il est souvent pratique de décomposer le système que l'on cherche à modéliser en sous-systèmes plus petits. Le problème est ensuite de faire communiquer entre eux les différents sous-systèmes pour obtenir un modèle du système original. Alors que dans le cadre non temporisé, des notions de composition de systèmes semblent clairement définis, dans le cadre temporisé, la situation est *a priori* un peu plus floue. Dans ce cadre, plusieurs solutions ont été proposées.

– Dans [BST97, BS98], plusieurs modes de synchronisation sont proposés pour composer les systèmes. Tous ces modes de synchronisation sont d'un intérêt pratique pour la modélisation de systèmes réels. Grâce à l'ajout d'un « deadline » sur chaque transition, des notions d'*actions urgentes* peuvent être définies[1], ainsi que des notions de priorité entre actions pour éviter au maximum les choix non déterministes dans les modèles. Dans [BGS00, Goe01], une méthodologie complète de modélisation de systèmes temporisés est proposée en utilisant les notions précédemment décrites et en décrivant plusieurs propriétés qui sont préservées par la composition parallèle ainsi que par l'ajout de priorités.

– Une notion d'urgence différente est utilisée dans l'outil de model-checking UPPAAL [LPY97], celle-ci provient de la synchronisation utilisée dans l'algèbre de processus CCS [Mil89] étendue à TCCS [Yi90]. Dans le modèle d'UPPAAL, la notion d'urgence n'intervient que lors de la mise en parallèle des automates : lorsqu'une action est *urgente*, et qu'elle peut se synchroniser avec son « complémentaire » (rappelons que c'est une synchronisation binaire), alors il n'est pas possible d'attendre. Pour pallier ce mode de synchronisation simplifié et permettre par exemple les broadcasts, la notion d'état « committed » a été proposée : cela permet de forcer deux actions de s'effectuer séquentiellement de manière instantanée. Le problème réside peut-être dans le fait que cela peut entraîner des blocages car il est difficile de prévoir toutes les possibilités d'évolution de deux systèmes mis en parallèle.

2.5.5 Étude des langages temporisés

L'intérêt d'étudier les langages temporisés est de proposer un socle théorique solide aux travaux de vérification des systèmes temporisés, comme les langages formels en ont fourni un pour de nombreux travaux en informatique, en particulier pour des travaux de vérification de systèmes non temporisé.
Dans le cadre temporisé, des travaux sur ce thème ont commencé à voir le jour. Parmi ceux-ci, nous pouvons citer les travaux [ACM97, Asa98, ACM01] sur la définition d'expressions rationnelles temporisées permettant d'obtenir un théorème de Kleene.

Un autre aspect des langages formels qui a été aussi beaucoup étudié dans le cadre temporisé est l'aspect « caractérisation logique ». Nous pouvons par exemple citer les travaux de Thomas Wilke [Wil94] qui propose une logique monadique su second ordre avec distances pour caractériser les langages acceptés par des automates temporisés ; ou bien Jean-François Raskin, Pierre-Yves Schobbens puis Thomas Henzinger [RS97a, HRS98] qui proposent plusieurs logiques pour exprimer les comportements de différents types d'automates évoqués dans la partie 2.5.2 ; ou bien Deepak D'Souza [D'S00] qui a caractérisé les automates définis dans [AFH94] *via* une logique monadique du second-ordre sans restriction sur la négation.

Nous ne détaillons pas tous ces aspects, nous reviendrons sur certains d'entre eux dans la partie 4.6.2, mais nous dessinons un panorama de ce qui avait ou n'avait pas été fait dans le domaine « langages temporisés & langages formels » avant 1998, voir tableau 2.4.

[1]Par exemple, une action « eager » est une action qui doit être effectuée dès qu'il est possible de le faire alors qu'une action « *delayable* » est une action qui doit être réalisée avant qu'il ne soit plus possible de le faire.

Langages formels	**Langages temporisés**
Automates finis	Automates temporisés [AD90, AD94]
Expressions rationnelles	Une proposition [ACM97, Asa98]
Caractérisation logique MSO(<), LTL	Quelques logiques *ad hoc* [Wil94, HRS98]
Caractérisation algébrique (en utilisant des monoïdes finis)	?

Tableau 2.4: Comparaison entre les langages formels et les langages temporisés

2.6 Ce que nous allons faire

Les travaux sur les modèles temporisés que nous allons présenter s'inscrivent dans plusieurs des catégories dont nous venons de parler.

Les travaux sur les automates temporisés avec mises à jour que nous présentons au chapitre 3 s'inscrivent à la fois dans les extensions des automates temporisés et dans les aspects de modélisation. En effet, le modèle que nous étudions étend le modèle des automates temporisés classiques avec des opérations plus générales sur les horloges. De plus, même si ce modèle s'avère ne pas être plus expressif que les automates temporisés avec ε-transitions, nous verrons qu'il permet de représenter de manière plus concise des systèmes réels, ce qui est un véritable avantage pour les travaux de modélisation.

Les travaux que nous présentons dans les chapitres 4, 5 et 6 s'inscrivent principalement dans la recherche d'un socle théorique solide pour les langages temporisés. Après avoir proposé une notion d'expressions rationnelles temporisées permettant d'obtenir un théorème de Kleene comme dans le cadre non temporisé, ces travaux nous ont amenés à définir une nouvelle sémantique, les *langages de données*. Celle-ci généralise la notion de langages temporisés à des ensembles (de données) qui ne sont pas forcément des domaines de temps. Elle nous permet alors de proposer différents formalismes de description, de natures très diverses, permettant de définir, comme dans le cas des langages formels, des langages de données *reconnaissables*. Notons que le théorème de Kleene obtenu pour les langages temporisés nous paraît pouvoir s'adapter sans trop de difficultés à notre nouveau formalisme des langages de données, ce qui fait que nous obtenons un parallèle très intéressant entre les langages de données et les langages formels.

Chapitre 3

Les automates temporisés avec mises à jour

Le contenu de ce chapitre a fait l'objet des articles [BDFP00a] et [BDFP00b].

Le protocole ABR *Available Bit Rate*, qui fait partie de la norme ITU, est développé par France-Telecom R&D. Il est destiné à contrôler les débits d'émission de données par des usagers abonnés à un contrat de communication par un réseau ATM. Dans un tel réseau, le routage est réalisé par commutation de paquets, ce qui permet une plus grande flexibilité des communications, avec des débits qui peuvent varier suivant la charge du réseau. Parmi les contrats de connexion les plus récents, le mode ABR autorise un débit variable au cours d'une même session. Dans ce cas, le mécanisme d'offre et de contrôle des taux d'émission des usagers est lui-même dynamique. L'algorithme au coeur de ce mécanisme réalise à tout instant une mise à jour du taux courant, en fonction des informations provenant du réseau. Ces informations se présentent sous la forme de valeurs successives du débit, qui apparaissent sur les cellules RM (Resource Management cells) en transit du réseau vers l'utilisateur. Sa vérification est un problème difficile pour plusieurs raisons : fluctuation de la valeur du débit autorisé en fonction des disponibilités du réseau, ignorance du délai exact de transmission des informations entre le réseau et l'usager, présence de différents paramètres de configuration du réseau... On souhaite alors utiliser une variable prenant comme valeur le débit transporté par une cellule RM et modifier le taux courant en conséquence. Il semble difficile de modéliser ceci de manière naturelle avec de simples remises à zéro de variables, comme c'est le cas dans le modèle des automates temporisés, alors que des opérations plus générales sur les variables permettent de le faire (voir par exemple la modélisation de l'ABR faite dans [BF99, BFKM02], qui a été implémentée dans HyTech [HHWT97], un outil ayant implanté ce genre d'opérations plus générales).

Il apparaît alors naturel d'étendre le modèle des automates temporisés en autorisant des opérations plus générales sur les horloges. Nous formalisons ceci et définissons le modèle des *automates temporisés avec mises à jour*. Dans ce modèle, nous autorisons les opérations suivantes sur les horloges :

$$x :\sim c \;\mid\; x :\sim y + c, \text{ où } x, y \text{ sont des horloges, } c \in \mathbb{Q} \text{ et } \sim \in \{<, \leq, =, \geq, >\}$$

La mise à jour $x :< 2$ signifie que x reçoit une valeur plus petite que 2 et $x := y + 1$ signifie que x reçoit la valeur de y plus 1. Les mises à jour non déterministes (c'est-à-dire celles pour lesquelles $\sim$ n'est pas l'égalité) peuvent par exemple être interprétées comme une interaction avec un environnement, comme c'est le cas dans le protocole ABR : la variable va prendre une valeur « quelconque » qui sera fixée par l'environnement, composante que nous ne maîtrisons pas.

Nous définissons précisément ce modèle dans la première partie de ce chapitre.

Dans un second temps, nous étudions la décidabilité du problème du vide pour le nouveau modèle que nous avons défini. Il est facile de montrer que le modèle général est indécidable. Nous dessinons précisément la frontière entre les sous-classes décidables et les sous-classes indécidables de ce modèle. C'est ce que nous présentons dans les deux parties suivantes du chapitre. Nos principaux résultats à ce propos sont les suivants :
– Nous exhibons, grâce à une réduction au problème de l'arrêt d'une machine à deux compteurs, des classes de mises à jour pour lesquelles le test du vide est indécidable.
– Nous proposons une généralisation de la construction de l'automate des régions d'Alur et Dill présentée dans la partie 2.3.2. Celle-ci va permettre de prouver la décidabilité de larges sous-classes du modèle.

Un résultat surprenant tient dans le fait que les résultats de décidabilité dépendent de la nature des contraintes d'horloges (diagonales ou pas) utilisées. Ce point est très particulier car, dans le cadre des automates temporisés classiques, les contraintes diagonales ne changent nullement le modèle.

Dans un troisième temps, nous étudions l'expressivité du modèle, en nous restreignant plus particulièrement aux classes décidables. Nous répondons complètement à la question en montrant que tout langage accepté par un automate appartenant à une classe décidable est aussi accepté par un automate temporisé classique avec ε-transitions. Nous étudions même de manière plus précise ces transformations en termes de simulation. Cette étude fait l'objet de la quatrième et dernière partie de ce chapitre.

3.1 Les automates temporisés avec mises à jour

Comme nous l'avons annoncé, la différence principale entre le modèle que nous allons définir et les automates temporisés définis au chapitre 2 réside dans les opérations autorisées sur les horloges.

3.1.1 Les mises à jour

Une *mise à jour* d'un ensemble d'horloges X est une fonction de $\mathbb{T}^X$ dans $2^{\mathbb{T}^X}$ qui associe à toute valuation un ensemble de valuations. Nous n'allons bien évidemment pas considérer ces mises à jour de manière générale, mais nous allons nous restreindre à un sous-ensemble.
Une *mise à jour simple* d'une horloge $z \in X$ peut être représentée par l'une des formes

$$z :\sim c \;\mid\; z :\sim y + c$$

où $c \in \mathbb{Q}$, $y \in X$ et $\sim \in \{<, \leq, =, \geq, >\}$. Si up est l'une des mises à jour simples précédemment décrites et si v est une valuation sur X, une valuation v' est dans $up(v)$ si et seulement si $v'(x) = v(x)$ pour $x \neq z$ et $v'(z)$ vérifie :

$$\begin{cases} v'(z) \sim c \text{ si } up \text{ est } z :\sim c \\ v'(z) \sim v(y) + c \text{ si } up \text{ est } z :\sim y + c \end{cases}$$

Nous allons considérer les mises à jour qui sont la conjonction de telles mises à jour simples. Dans ce qui suit, une *mise à jour* sera donc de la forme

$$up = \bigwedge_{i=1}^{p} up_{x_i}$$

où chaque up_{x_i} est une mise à jour simple pour une horloge x_i (bien évidemment, nous pouvons avoir $x_i = x_j$ même si $i \neq j$). Si v et v' sont des valuations sur X, nous dirons que v' est dans $up(v)$ si et seulement si, pour tout $i = 1 \dots p$, la valuation v'_{x_i} définie par

$$\begin{cases} v'_{x_i}(x_i) = v'(x_i) \\ v'_{x_i}(x) = v(x) \text{ si } x \neq x_i \end{cases}$$

est dans $up_{x_i}(v)$.

***Exemple* 3.1** Considérons la mise à jour $up = x :> y \wedge x :< 7$. Si v est une valuation et si $v' \in up(v)$, cela signifie que $v'(x) > v(y)$ et $v'(x) < 7$. Nous remarquons ici qu'il est possible qu'un tel v' n'existe pas, par exemple si $v(y) = 8$.

Nous noterons $\mathcal{U}(X)$ l'ensemble de ces mises à jour. Dans la suite, nous distinguerons l'ensemble particulier suivant de mises à jour :

– $\mathcal{U}_0(X)$ est l'ensemble des mises à jour de la forme

$$\bigwedge_{x \in C} x := 0$$

où $C \subseteq X$. C'est donc l'ensemble des remises à zéro des horloges.

Une mise à jour simple est dite *déterministe* si elle est de l'une des formes

$$z := c \mid z := y + c$$

où $c \in \mathbb{Q}$ et $y \in X$. Par extension, une mise à jour up est *déterministe* si elle est la conjonction de mises à jour simples déterministes et si pour toute horloge x, il y a au maximum une mise à jour simple pour x dans up.

3.1.2 Définition du modèle

Un *automate temporisé avec mises à jour* est un 7-uplet $\mathcal{A} = (Q, X, \Sigma, Q_0, F, R, T)$ où

– Q est un ensemble fini d'états,
– X est un ensemble fini d'horloges,
– Σ est l'alphabet fini d'actions,
– $Q_0 \subseteq Q$, $F \subseteq Q$, $R \subseteq Q$ sont respectivement les états initiaux, les états finals et les états répétés,
– $T \subseteq Q \times \mathcal{C}(X) \times \Sigma \times \mathcal{U}(X) \times Q$ est l'ensemble des transitions.

La seule différence avec les automates temporisés classiques réside dans l'ensemble des transitions. Ici, les transitions permettent d'utiliser des mises à jour plus générales que les remises à zéro des horloges. Les notions qui suivent sont identiques à celles définies dans le cadre des automates temporisés d'Alur et Dill.

Un *chemin* dans l'automate temporisé avec mises à jour $\mathcal{A}$ est une suite finie ou infinie de transitions consécutives

$$q_0 \xrightarrow{g_1, a_1, up_1} q_1 \xrightarrow{g_2, a_2, up_2} \dots \xrightarrow{g_n, a_n, up_n} q_n \dots$$

Soit $u = (a_1, t_1) \dots (a_n, t_n) \dots$ un mot temporisé (fini ou infini) de $(\Sigma \times \mathbb{T})^\infty$. Une *exécution* sur le chemin précédent pour le mot u est de la forme

$$(q_0, v_0) \xrightarrow[t_1]{g_1, a_1, up_1} (q_1, v_1) \xrightarrow[t_2]{g_2, a_2, up_2} \dots \xrightarrow[t_n]{g_n, a_n, up_n} (q_n, v_n) \dots$$

où $(v_i)_{i \geq 0}$ est une suite de valuations telle que pour tout $x \in X$, $v_0(x) = 0$ et pour tout $i \geq 0$,

$$\begin{cases} v_{i+1} \in up_{i+1}(v_i + t_{i+1} - t_i) \\ v_i + t_{i+1} - t_i \models g_i \end{cases}$$

[Par convention, nous supposons que $t_0 = 0$.]
Une telle exécution est dite acceptante si $q_0 \in Q_0$ et

– soit elle est finie et termine dans un état final,
– soit elle est infinie et passe infiniment souvent par un état répété.

Un mot u est alors dit accepté par l'automate s'il existe une exécution acceptante sur ce mot. L'ensemble des mots acceptés par l'automate $\mathcal{A}$ est le langage accepté par $\mathcal{A}$ et est noté $L(\mathcal{A})$.

De la manière que nous avions défini les automates temporisés classiques avec ε-transitions (voir à la page 32), nous définissons les automates temporisés avec mises à jour et ε-transitions en étendant l'ensemble des actions possibles à l'ensemble Σ_ε au lieu de Σ.

Soient $\mathcal{C} \subseteq \mathcal{C}(X)$ et $\mathcal{U} \subseteq \mathcal{U}(X)$ des sous-ensembles de contraintes d'horloges et de mises à jour.
– L'ensemble *Aut*$(\mathcal{C},\mathcal{U})$ désigne l'ensemble des automates temporisés avec mises à jour pour lesquels nous restreignons les transitions à un sous-ensemble de $T \subseteq Q \times \mathcal{C} \times \Sigma \times \mathcal{U} \times Q$. Par exemple, l'ensemble des automates temporisés classiques correspond à l'ensemble *Aut*$(\mathcal{C}(X),\mathcal{U}_0(X))$.
– L'ensemble *Aut*$_\varepsilon(\mathcal{C},\mathcal{U})$ désigne l'ensemble des automates temporisés avec ε-transitions pour lesquels nous restreignons les transitions à un sous-ensemble de $T \subseteq Q \times \mathcal{C} \times \Sigma_\varepsilon \times \mathcal{U} \times Q$.

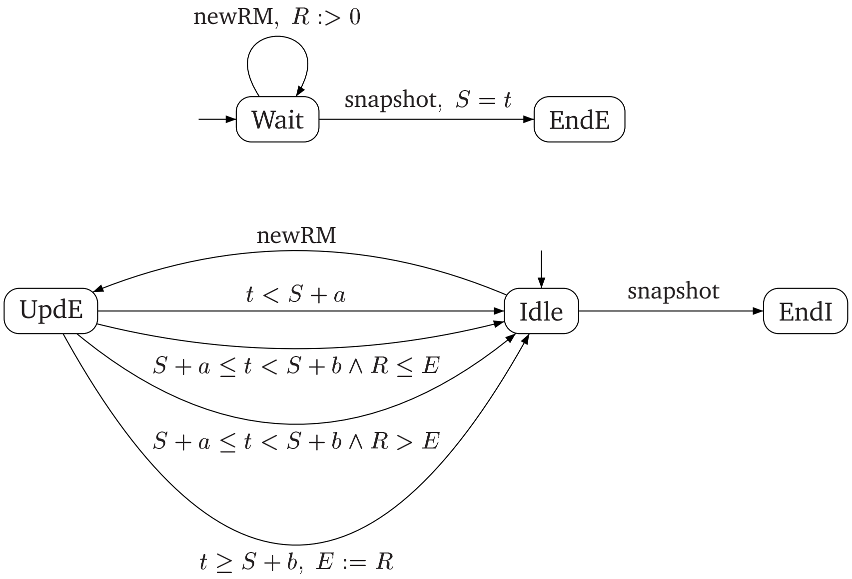


Figure 3.1: Une partie de la modélisation du protocole ABR

***Exemple* 3.2** Nous présentons dans notre formalisme deux des trois automates qui permettent de modéliser le protocole ABR [BF99] dont nous avons parlé dans le début de ce chapitre. Les paramètres t, a et b qui sont utilisés dans la modélisation sont ici supposés instanciés. L'horloge S est une horloge qui n'est jamais remise à zéro et qui représente donc le temps universel. L'horloge R représente le débit du réseau transporté par la cellule RM. L'horloge E représente le taux idéal du réseau qui sera proposé à l'utilisateur. Les automates représentant l'environnement et l'algorithme $\mathcal{I}$, légèrement simplifiés par rapport à la modélisation de [BF99], sont dessinés sur la figure 3.1.

Dans ce chapitre, et uniquement dans ce chapitre, nous dirons automates temporisés au lieu d'automates temporisés avec mises à jour pour éviter des lourdeurs dans l'écriture.

3.2 Indécidabilité

Nous commençons notre étude en étudiant le problème du vide évoqué dans la partie 2.3 et en exhibant des sous-classes indécidables. Pour ce faire, nous réduisons le problème de l'arrêt d'une machine à deux compteurs au problème du vide de ces sous-classes. Nous commençons donc par quelques rappels sur la machine à deux compteurs, ou machine de Minsky [Min67].

3.2.1 La machine à deux compteurs

Une *machine à deux compteurs* (ou aussi *machine de Minsky*) est un ensemble fini d'instructions. Ces instructions sont étiquetées et permettent de gérer deux compteurs, c_1 et c_2. Les instructions sont de deux types :

– une **instruction d'incrémentation** du compteur $x \in \{c_1, c_2\}$:

$$\texttt{p} : \; x := x + 1 \; ; \; \text{goto } \; \texttt{q} \quad \text{(où p et q sont des étiquettes d'instructions)}$$

– une **instruction de décrémentation avec test à zéro** du compteur $x \in \{c_1, c_2\}$:

$$\texttt{p} : \; \text{si} \; x > 0 \; \begin{cases} \text{alors} & x := x - 1 \; ; \; \text{goto } \texttt{q} \\ \text{sinon} & \text{goto } \texttt{q}' \end{cases}$$

La machine débute une exécution par une instruction initiale fixée avec les deux compteurs initialisés à zéro ($c_1 = c_2 = 0$) et s'arrête lorsqu'elle arrive à une instruction spéciale étiquetée par HALT. Le *problème de l'arrêt* pour une telle machine à deux compteurs consiste à décider si la machine atteint l'instruction étiquetée par HALT.

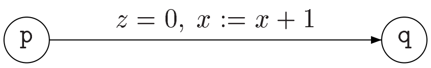
Le résultat suivant est à la base de tous nos résultats d'indécidabilité.

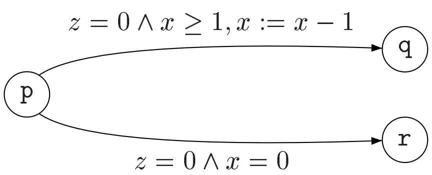
Théorème 3.3 *[Min67] Le problème de l'arrêt pour les machines à deux compteurs est indécidable.*

Ce résultat peut se montrer par réduction du problème de l'arrêt d'une machine de Turing au problème de l'arrêt d'une machine à deux compteurs. Une preuve claire et assez simple de cette réduction peut être trouvée dans [HU79].

3.2.2 Premiers résultats d'indécidabilité

De manière évidente, la classe générale *Aut*($\mathcal{C}(X)$,$\mathcal{U}(X)$) est indécidable. En effet, il est très facile de simuler une machine à deux compteurs avec un automate temporisé. Les deux instructions possibles d'une machine à deux compteurs peuvent être codées de la manière suivante (l'unique action a de l'alphabet Σ n'est pas représentée) :

– **Incrémentation du compteur** x **:** 

– **Décrémentation du compteur** x **:** 

où z est une nouvelle horloge qui contrôle l'écoulement du temps.

Ainsi, étant donnée une machine à deux compteurs $\mathcal{M}$, nous pouvons facilement construire un automate temporisé avec mises à jour $\mathcal{A}_{\mathcal{M}}$ qui vérifie :

$$\mathcal{M} \text{ s'arrête} \iff L(\mathcal{A}_{\mathcal{M}}) \neq \varnothing$$

Nous obtenons donc le résultat suivant :

Proposition 3.4 *Soit X un ensemble contenant au moins trois horloges. Alors, la classe d'automates temporisés Aut($\mathcal{C}_{df}(X)$,$\mathcal{U}(X)$) est indécidable.*

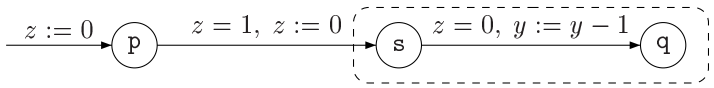
Comme toute classe contenant une sous-classe indécidable est elle-même indécidable, nous obtenons immédiatement le corollaire suivant :

Corollaire 3.5 *Soit X un ensemble contenant au moins trois horloges. Alors les classes d'automates Aut($\mathcal{C}(X)$,$\mathcal{U}(X)$), Aut$_\varepsilon$($\mathcal{C}_{df}(X)$,$\mathcal{U}(X)$) et Aut$_\varepsilon$($\mathcal{C}(X)$,$\mathcal{U}(X)$) sont indécidables.*

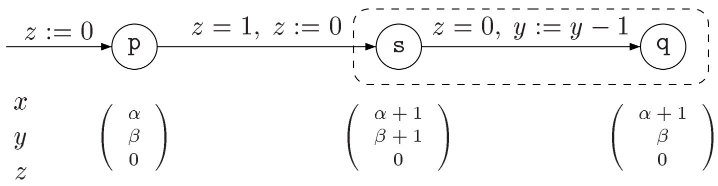
3.2.3 D'autres réductions

Les simulations précédentes utilisent des mises à jour de deux types, $x := x + 1$ et $x := x - 1$. Nous allons voir que si nous autorisons la remise à zéro des horloges, un seul des deux types précédents est suffisant pour obtenir des résultats d'indécidabilité. Nous commençons par considérer les mises à jour du type $x := x - 1$. L'incrémentation d'un compteur peut être simulée de la manière suivante :

Incrémentation du compteur x :

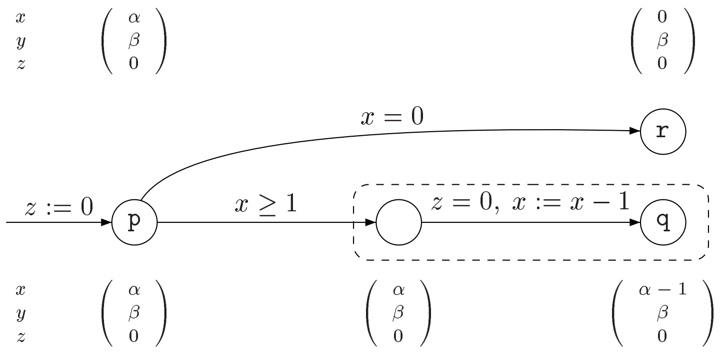


Nous montrons qu'une exécution sur ce chemin augmente la valeur de l'horloge x d'une unité de temps et laisse le valeur de l'horloge y inchangée. En effet, lors d'une telle exécution, les valeurs des horloges (dans l'ordre $x,\ y,\ z$ de gauche à droite) sont $(\alpha, \beta, 0)$ dans l'état p, $(\alpha + 1, \beta + 1, 0)$ dans l'état s et $(\alpha + 1, \beta, 0)$ dans l'état q. Dans tout ce qui suit, nous représenterons ceci par une figure simple du type suivant :



La simulation de l'instruction de décrémentation d'un compteur est identique à celle vue auparavant. Nous représentons cela de manière schématique :

Décrémentation du compteur x :

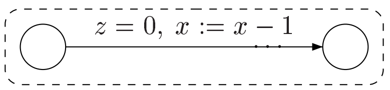


Grâce à cette nouvelle simulation, nous avons prouvé le résultat suivant :

Proposition 3.6 *Soit X un ensemble contenant au moins trois horloges. Soit $\mathcal{U}$ un ensemble de mises à jour contenant à la fois $\mathcal{U}_0(X)$ et $\{x := x - 1 \mid x \in X\}$. Alors la classe Aut($\mathcal{C}_{df}(X)$,$\mathcal{U}$) est indécidable.*

Jusqu'à maintenant, tous nos résultats d'indécidabilité sont vrais aussi bien pour les contraintes d'horloges non diagonales que pour les contraintes générales. Dans le reste de cette partie, nous allons obtenir de nouveaux résultats d'indécidabilité, mais uniquement lorsque nous considérons des contraintes d'horloges générales. Comme nous le verrons après, ces résultats ne seront pas vrais pour les contraintes non diagonales.

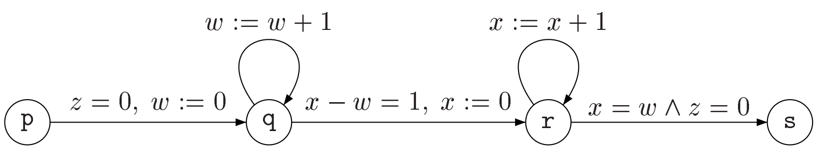
De la simulation précédemment décrite, il apparaît qu'il n'est plus nécessaire de simuler toute la machine à deux compteurs, mais qu'il est suffisant, pour obtenir de nouvelles classes indécidables, de simuler les transitions de la forme :



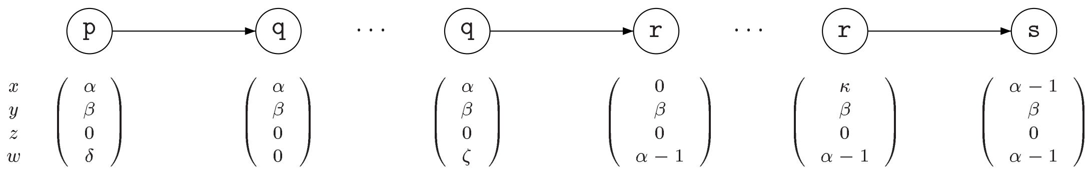
(3.1)

Bien sûr, nous supposons que la remise à zéro des horloges est toujours autorisée.

Nous commençons par montrer que ces transitions peuvent être simulées en utilisant seulement des remises à zéro et des incrémentations d'horloges. Cela peut se faire de la manière suivante :



En effet, de la même manière que précédemment, les valeurs d'horloges le long d'une exécution sur ce chemin peuvent être décrites par le schéma suivant :



Une exécution sur ce chemin simule donc bien une transition (3.1). Nous avons ainsi démontré qu'il est possible de simuler un machine à deux compteurs avec uniquement des remises à zéro et des incrémentations.

Proposition 3.7 *Soit X un ensemble contenant au moins quatre horloges. Soit $\mathcal{U}$ un ensemble de mises à jour contenant à la fois $\mathcal{U}_0(X)$ et $\{x := x + 1 \mid x \in X\}$. Alors la classe Aut($\mathcal{C}(X)$,$\mathcal{U}$) est indécidable.*

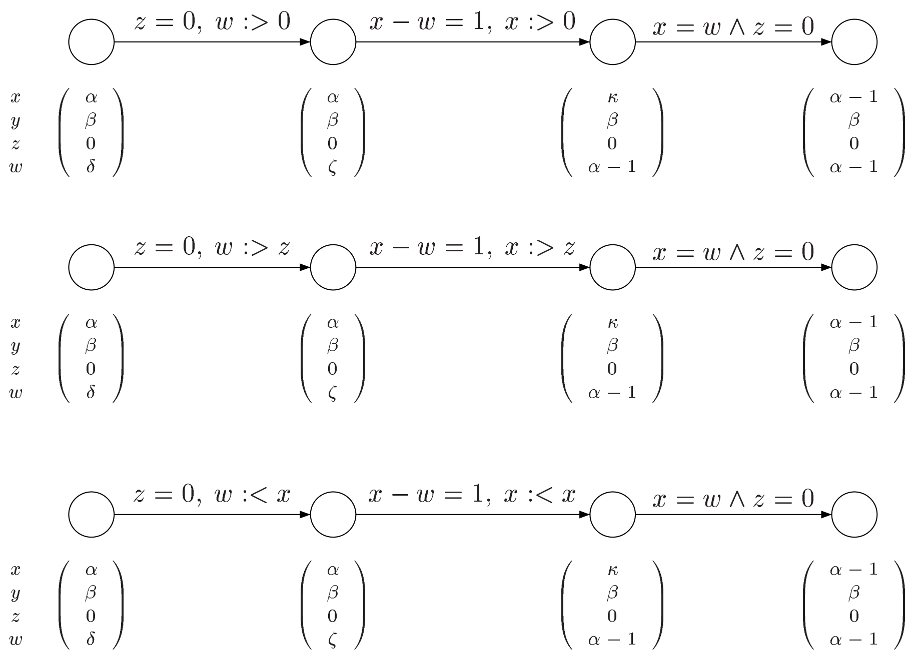
Les résultats d'indécidabilité qui vont suivre sont obtenus par des techniques très similaires.

Proposition 3.8 *Soit X un ensemble contenant au moins quatre horloges. Soit $\mathcal{U}$ un ensemble de mises à jour contenant à la fois $\mathcal{U}_0(X)$ et au moins l'un des ensembles suivants :*

- $\{x :> 0 \mid x \in X\}$,
- $\{x :> y \mid x, y \in X\}$,
- $\{x :< y \mid x, y \in X\}$.

Alors la classe Aut($\mathcal{C}(X)$,$\mathcal{U}$) est indécidable.

Preuve : Comme dans la preuve précédente, nous simulons une transition (3.1). Les trois schémas ci-dessous correspondent respectivement aux trois ensembles de mises à jour décrits dans l'énoncé de la proposition.



□

3.2.4 Conclusion

Nous avons maintenant prouvé nos principaux résultats d'indécidabilité. Quelques autres vont pouvoir être déduits de ceux-ci. Nous résumons l'ensemble de nos résultats d'indécidabilité dans le tableau 3.2.

Dans ce tableau, l'indécidabilité des lignes 2 et 4 correspond exactement aux résultats des propositions 3.6 et 3.7. La ligne 3 apparaît comme une généralisation de la ligne 2. Les lignes 6, 7 et 8 correspondent aux résultats de la proposition 3.8. La ligne 9 provient de la ligne 7 et du fait que toutes les contraintes diagonales peuvent être remplacées par une mise à jour qui ne peut être satisfaite que si la contrainte est vérifiée. La transformation est la suivante. Considérons une transition

<table>
<tr><th></th><th>$\mathcal{U}_0(X) \cup \ldots$</th><th>Contraintes non diagonales</th><th>Contraintes générales</th></tr>
<tr><td>1</td><td>$x := c, x := y$</td><td rowspan="3">?</td><td>?</td></tr>
<tr><td>2</td><td>$x := x + 1$</td><td rowspan="3">Indécidable</td></tr>
<tr><td>3</td><td>$x := y + c$</td></tr>
<tr><td>4</td><td>$x := x - 1$</td><td>Indécidable</td></tr>
<tr><td>5</td><td>$x :< c$</td><td rowspan="4">?</td><td>?</td></tr>
<tr><td>6</td><td>$x :> c$</td><td rowspan="4">Indécidable</td></tr>
<tr><td>7</td><td>$x :\sim y + c$</td></tr>
<tr><td>8</td><td>$y + c <: x :< y + d$</td></tr>
<tr><td>9</td><td>$y + c <: x :< z + d$</td><td>Indécidable</td></tr>
</table>

avec $\sim \in \{\leq, <, >, \geq\}$ et $c,\ d \in \mathbb{Q}^+$.

Tableau 3.2: Résultats d'indécidabilité

$$\bigcirc \xrightarrow{\boxed{x - y < c}\ \wedge\ g,\ a,\ up} \bigcirc$$

Elle peut être remplacée par la transition

$$\bigcirc \xrightarrow{g,\ a,\ \boxed{x <: z :< y + c}\ \wedge\ up} \bigcirc$$

où z est une nouvelle horloge. Cette transition ne peut être franchie que s'il est possible d'affecter une valeur entre x et $y + c$ à z, c'est-à-dire si $x < y + c$, ce qui est équivalent à $x - y < c$. La deuxième transition simule donc bien la première transition.

La partie suivante est consacrée à l'étude des classes marquées d'un point d'interrogation dans le tableau, ce sont des classes pour lesquelles nous n'avons pas réussi à prouver l'indécidabilité. Comme nous allons le voir, ces classes sont en fait décidables.

3.3 Décidabilité

Dans cette partie, nous allons exhiber des classes décidables d'automates temporisés avec mises à jour, étendant ainsi le théorème 2.6 concernant les automates temporisés d'Alur et Dill. La preuve de nos résultats de décidabilité repose sur une généralisation des techniques utilisées par Alur et Dill et rappelées dans la partie 2.3.2. Nous commençons par proposer des notions de régions et de graphe des régions généralisant celles d'Alur et Dill.

3.3.1 Régions et automate des régions

Soit X un ensemble fini d'horloges. Nous considérons une partition finie $\mathcal{R}$ de $\mathbb{T}^X$. Pour chaque valuation $v \in \mathbb{T}^X$, l'unique élément de $\mathcal{R}$ qui contient v est noté $[v]_{\mathcal{R}}$. Nous définissons les successeurs de R, $\mathsf{Succ}(R) \subseteq \mathcal{R}$ de manière naturelle :

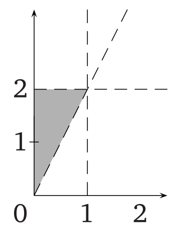
$$R' \in \mathsf{Succ}(R) \text{ s'il existe } v \in R, \text{ et } t \in \mathbb{T} \text{ t.q. } v + t \in R'$$

Nous disons qu'une telle partition finie est un *ensemble de régions* si la condition suivante est vérifiée :

$$R' \in \mathsf{Succ}(R) \iff \forall v \in R,\ \exists t \in \mathbb{T} \text{ t.q. } v + t \in R' \tag{3.2}$$

Cette condition est assez naturelle, elle exprime le fait que deux valuations d'une région sont équivalentes et indistinguables vis-à-vis de l'écoulement du temps. Notons que cette condition n'est pas vérifiée par n'importe quelle partition finie de $\mathbb{T}^X$.

***Contre-exemple* 3.9** Considérons la partition de $\mathbb{T}^2$ dessinée sur la figure à droite. La condition (3.2) n'est pas vérifiée par la région grisée. En effet, la valuation $(0,5;1,8)$ permet, en laissant le temps s'écouler, d'atteindre la valuation $(0,7;2)$ appartenant à la région définie par les contraintes $0 < x < 1 \wedge y = 2$, alors que la valuation $(0,5;1,1)$ ne permet pas d'atteindre cette région en laissant du temps s'écouler.



Soit $\mathcal{U} \subseteq \mathcal{U}(X)$ un ensemble fini de mises à jour. Chaque mise à jour $up \in \mathcal{U}$ induit naturellement une fonction $\widehat{up} : \mathcal{R} \to \mathcal{P}(\mathcal{R})$ qui envoie chaque région R sur l'ensemble $\{R' \in \mathcal{R} \mid up(R) \cap R' \neq \varnothing\}$, c'est-à-dire l'ensemble des régions qui intersectent l'image de R par up. L'ensemble de régions $\mathcal{R}$ est dit *compatible* avec $\mathcal{U}$ si la condition suivante est vérifiée :

$$R' \in \widehat{up}(R) \implies \forall v \in R,\ \exists v' \in R' \text{ t.q. } v' \in up(v) \tag{3.3}$$

Remarque : Si nous définissons les relations de transition $(\hookrightarrow_{up})_{up}$ sur $\mathbb{T}^X$ par

$$v \hookrightarrow_{up} v' \iff v' \in up(v)$$

et la relation $\rho_{\mathcal{R}}$ par

$$v\ \rho_{\mathcal{R}}\ v' \iff [v]_{\mathcal{R}} = [v']_{\mathcal{R}}$$

alors la condition (3.3) revient à dire que $\rho_{\mathcal{R}}$ est une bisimulation pour les relations $(\hookrightarrow_{up})_{up}$.

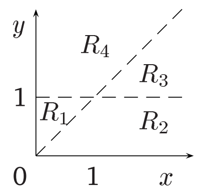
Si un ensemble de régions $\mathcal{R}$ est compatible avec un ensemble de mises à jour $\mathcal{U}$, nous définissons le *graphe des régions* associé à $\mathcal{R}$ et $\mathcal{U}$ comme le graphe dont les nœuds sont les éléments de $\mathcal{R}$ et les arêtes sont de deux types distincts :

$$\begin{cases} R \longrightarrow R' & \text{si } R' \in \mathsf{Succ}(R) \\ R \longrightarrow_{up} R' & \text{si } R' \in \widehat{up}(R) \end{cases}$$

***Exemple* 3.10** Considérons l'ensemble des régions $\mathcal{R}$ dessiné sur le dessin de droite. Il est constitué des quatre régions qui peuvent être définies par les équations suivantes :

R_1	R_2	R_3	R_4
$0 \le x < 1$	$x \ge 0$	$x > 1$	$x \ge 0$
$0 \le y < 1$	$0 \le y \le 1$	$y > 1$	$y > 1$
$x < y$	$x \ge y$	$x \ge y$	$x < y$

Alors, il est assez facile de voir que $\mathcal{R}$ est compatible avec l'ensemble des mises à jour $\mathcal{U} = \{x := 1, y := 0\}$. Pour cela, nous calculons le graphe des régions qui est associé à cet ensemble de régions et à $\mathcal{U}$. Il est dessiné sur la figure 3.3.



Soit $\mathcal{A} = (Q, X, \Sigma, I, F, \mathrm{Rep}, T)$ un automate temporisé appartenant à une certaine classe *Aut*$(\mathcal{C},\mathcal{U})$ et soit $\mathcal{R}$ une famille de régions compatible avec $\mathcal{U}$. Nous définissons l'*automate des régions* $\Gamma_{\mathcal{R}}(\mathcal{A})$ associé à $\mathcal{A}$ et $\mathcal{R}$ comme étant l'automate fini (de Büchi) tel que :

- son ensemble d'états est $Q \times \mathcal{R}$,

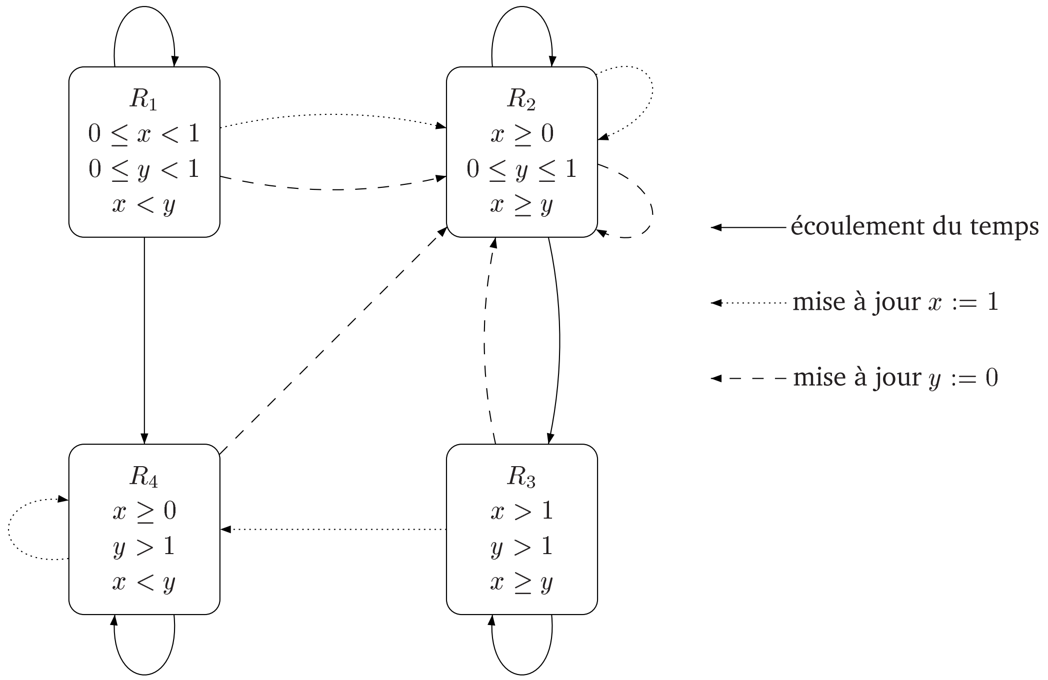


Figure 3.3: Un exemple simple de graphe des régions

- ses états initiaux sont $(q_0, \mathbf{0})$ où $q_0 \in I$ est initial et $\mathbf{0}$ est l'unique région contenant la valuation où toutes les horloges valent zéro,
- ses états finals sont (f, R) où f est final dans $\mathcal{A}$ et R est une région quelconque,
- ses états répétés sont (r, R) où r est répété dans $\mathcal{A}$ et R est une région quelconque.

- ses transitions sont définies par $(q, R) \xrightarrow{a} (q', R')$ s'il existe une région $\widehat{R}$ et une transition (q, g, a, up, q') dans $\mathcal{A}$ telle que :
  - $R \longrightarrow \widehat{R}$ est une transition du graphe des régions,
  - $\widehat{R} \subseteq g$,
  - $\widehat{R} \longrightarrow_{up} R'$ est une transition du graphe des régions.

Soit $\mathcal{C} \subseteq \mathcal{C}(X)$ un ensemble fini de contraintes d'horloges. L'ensemble des régions $\mathcal{R}$ est dit *compatible* avec $\mathcal{C}$ si pour tout $g \in \mathcal{C}$ et pour tout $R \in \mathcal{R}$, soit $R \subseteq g$, soit $R \subseteq \neg g$.

Sous les conditions (3.2) et (3.3), l'automate des régions abstrait l'automate initial de manière satisfaisante, dans le sens où nous obtenons un résultat similaire à celui de la proposition 2.12 :

Proposition 3.11 *Soit $\mathcal{A}$ un automate temporisé de Aut($\mathcal{C}$,$\mathcal{U}$) où $\mathcal{C}$ (respectivement $\mathcal{U}$) est un ensemble fini de contraintes d'horloges (respectivement de mises à jour). Soit $\mathcal{R}$ un ensemble de régions compatible avec $\mathcal{C}$ et $\mathcal{U}$. Alors l'automate fini $\Gamma_{\mathcal{R}}(\mathcal{A})$ accepte le langage* $\mathsf{Untime}(L(\mathcal{A}))$[1].

PREUVE : Supposons que $\mathcal{A} = (Q, X, \Sigma, I, F, R, T)$ et considérons une exécution dans $\mathcal{A}$

$$(q_0, v_0) \xrightarrow[t_1]{g_1,\ a_1,\ up_1} (q_1, v_1) \xrightarrow[t_2]{g_2,\ a_2,\ up_2} \cdots$$

[1]Rappelons (voir la partie 2.3.2) que si L est un langage temporisé

$$\mathsf{Untime}(L) = \{a_1 \dots a_n \cdots \mid \exists t_1 \dots t_n \dots\ (a_1, t_1) \dots (a_n, t_n) \cdots \in L\}.$$

Pour $i \geq 0$, définissons $R_i = [v_i]_{\mathcal{R}}$ et $\widehat{R_i} = [v_i + t_{i+1} - t_i]_{\mathcal{R}}$. Nous avons que $\widehat{R_i} \in \mathsf{Succ}(R_i)$ et, comme $v_{i+1} \in up_{i+1}(v_i + t_{i+1})$, $R_{i+1} \in \widehat{up_i}(\widehat{R_i})$. De plus, $v_i + t_{i+1} \models g_{i+1}$ et comme $\mathcal{R}$ est compatible avec $\mathcal{C}$, nous en déduisons que $\widehat{R_i} \subseteq g_{i+1}$. Ainsi, de par la définition,

$$(q_0, R_0) \xrightarrow{a_1} (q_1, R_1) \xrightarrow{a_2} \cdots$$

est un chemin acceptant de $\Gamma_{\mathcal{R}}(\mathcal{A})$. Nous obtenons donc que $\mathsf{Untime}(L(\mathcal{A})) \subseteq L(\Gamma_{\mathcal{R}}(\mathcal{A}))$.

Réciproquement, considérons un chemin

$$(q_0, R_0) \xrightarrow{a_1} (q_1, R_1) \xrightarrow{a_2} \cdots$$

dans $\Gamma_{\mathcal{R}}(\mathcal{A})$. Nous posons $v_0 = \mathbf{0}$ et supposons que nous avons déjà construit des suites $(v_i)_{0 \leq i < n}$ et $(t_i)_{1 \leq i < n}$ telles que $v_i \in R_i$ et

$$(q_0, v_0) \xrightarrow[t_1]{g_1, a_1, up_1} (q_1, v_1) \cdots \xrightarrow[t_{i-1}]{g_{i-1}, a_{i-1}, up_{i-1}} (q_{i-1}, v_{i-1})$$

est une exécution de $\mathcal{A}$. Comme $(q_{i-1}, R_{i-1}) \xrightarrow{a_i} (q_i, R_i)$ est une transition de $\Gamma_{\mathcal{R}}(\mathcal{A})$, il existe, par définition, une région $\widehat{R}$ et une transition $(q_{i-1}, g_i, a_i, up_i, q_i)$ dans $\mathcal{A}$ telle que

- $R_{i-1} \longrightarrow \widehat{R}$ est une transition du graphe des régions,
- $\widehat{R} \subseteq g_i$,
- $\widehat{R} \longrightarrow_{up_i} R_i$ est une transition du graphe des régions.

Comme $v_{i-1} \in R_{i-1}$ et l'ensemble des régions $\mathcal{R}$ vérifie (3.2), il existe un $t_i \in \mathbb{T}$ tel que $v_{i-1} + t_i - t_{i-1} \in \widehat{R}$. Maintenant, comme $\mathcal{R}$ est compatible avec up_i (donc vérifie la condition (3.3)), nous déduisons qu'il existe une valuation v_i telle que $v_i \in up_i(v_{i-1} + t_i - t_{i-1})$. Ainsi,

$$(q_0, v_0) \xrightarrow[t_1]{g_1, a_1, up_1} (q_1, v_1) \cdots \xrightarrow[t_{i-1}]{g_{i-1}, a_{i-1}, up_{i-1}} (q_{i-1}, v_{i-1}) \xrightarrow{g_i, a_i, up_i} (q_i, v_i)$$

est une exécution dans $\mathcal{A}$. Ainsi, nous construisons par induction une exécution

$$(q_0, v_0) \xrightarrow[t_1]{g_1, a_1, up_1} (q_1, v_1) \cdots (q_{i-1}, v_{i-1}) \xrightarrow[t_i]{g_i, a_i, up_i} (q_i, v_i) \cdots$$

dans $\mathcal{A}$. Nous avons donc montré que $L(\Gamma_{\mathcal{R}}(\mathcal{A})) \subseteq \mathsf{Untime}(L(\mathcal{A}))$, ce qui conclut la preuve de cette proposition. □

Comme le problème de tester le vide d'un langage accepté par un automate fini (ou de Büchi) est décidable [HU79] et comme le vide de $L(\mathcal{A})$ est évidemment équivalent à celui de $\mathsf{Untime}(L(\mathcal{A}))$, nous obtenons immédiatement le théorème suivant :

Théorème 3.12 *Soit $\mathcal{C}$ (respectivement $\mathcal{U}$) un ensemble fini de contraintes d'horloges (respectivement de mises à jour). Supposons qu'il existe un ensemble de régions $\mathcal{R}$ tel que $\mathcal{R}$ soit compatible avec $\mathcal{C}$ et $\mathcal{U}$. Alors la classe $Aut(\mathcal{C},\mathcal{U})$ est décidable.*

Ce théorème est bien sûr fondamental mais il ne permet pas d'exhiber des classes décidables d'automates temporisés avec mises à jour. En fait, nous devons trouver ou construire des ensembles de contraintes d'horloges, de mises à jour et de régions qui sont compatibles. Les régions utilisées par Alur et Dill (voir la partie 2.3.3) sont adaptées à leur modèle dans le sens où elles sont compatibles avec les contraintes d'horloges et les remises à zéro. En ce qui nous concerne, nous commençons par distinguer le type des contraintes, non diagonales ou générales, qui sont utilisées.

Nous rappelons alors le lemme suivant, vrai dans le cas des automates temporisés classiques (voir lemme 4.1 page 15 de [AD94]). Il s'étend trivialement à notre modèle.

Lemme 3.13 *Soit $\mathcal{A}$ un automate temporisé et soit λ un rationnel strictement positif. Soit $\lambda\mathcal{A}$ l'automate temporisé obtenu en remplaçant toutes les constantes μ apparaissant dans $\mathcal{A}$ par le produit $\lambda\mu$. Alors $L(\lambda\mathcal{A}) = \lambda L(\mathcal{A})$ où $\lambda L(\mathcal{A}) = \{(a_i, \lambda t_i)_{i\geq 0} \mid (a_i, t_i)_{i\geq 0} \in L(\mathcal{A})\}$.*

Ainsi, étant donné un automate temporisé $\mathcal{A}$, pour tout $\lambda \in \mathbb{Q}^+$, la vacuité de $L(\mathcal{A})$ est équivalente à celle de $L(\lambda\mathcal{A})$. En considérant le ppcm, m, de toutes les constantes apparaissant dans $\mathcal{A}$, l'automate $m\mathcal{A}$ n'a que des constantes entières. Dans toute la suite de cette partie, nous supposerons que les constantes qui apparaissent dans les automates temporisés sont des entiers.

3.3.2 Classes décidables - Automates avec contraintes non diagonales

Dans cette partie, nous considérons uniquement des contraintes d'horloges non diagonales sur un ensemble d'horloges X. Nous construisons un ensemble de régions conviendra pour ce type de contraintes.

Définition des régions

Pour chaque horloge $x \in X$, nous considérons une constante entière $\max_x$ et nous définissons l'ensemble d'intervalles :

$$\mathcal{I}_x = \{[c] \mid 0 \leq c \leq \max_x\} \cup \{]c; c+1[\mid 0 \leq c < \max_x\} \cup \{]\max_x; +\infty[\}$$

Soit R un uplet $((I_x)_{x\in X}, \prec)$ où :
- pour tout $x \in X$, $I_x \in \mathcal{I}_x$,
- $\prec$ est un préordre total[2] sur $X_0 = \{x \in X \mid I_x$ est un intervalle de la forme $]c; c+1[\}$.

La région associée à R est définie comme l'ensemble de valuations :

$$\left\{ v \in \mathbb{T}^X \;\middle|\; \begin{array}{l} \forall x \in X,\, v(x) \in I_x \text{ et} \\ \forall x, y \in X_0,\ x \prec y \iff \mathsf{frac}(v(x)) \leq \mathsf{frac}(v(y)) \end{array} \right\}$$

Dans ce qui suit, nous confondrons l'uplet R avec la région qu'il définit.

L'ensemble fini $\mathcal{R}_{(\max_x)_{x\in X}}$ de tous ces ensembles de $\mathbb{T}^X$ forme une partition de $\mathbb{T}^X$. Nous allons montrer que cet ensemble vérifie aussi la condition (3.2), c'est-à-dire que le lemme suivant est vérifié :

Lemme 3.14 *L'ensemble $\mathcal{R}_{(\max_x)_{x\in X}}$ est un ensemble de régions.*

PREUVE : Supposons que $R = ((I_x)_{x\in X}, \prec)$. Si pour tout $x \in X$, $I_x =]\max_x, +\infty[$, alors de manière évidente

$$\forall v \in R, \forall t \in \mathbb{T}, v + t \in R$$

et donc $\mathsf{Succ}(R) = \{R\}$. Sinon, il existe au moins une région $R' \neq R$ telle que $R' \in \mathsf{Succ}(R)$. Parmi toutes ces régions, nous définissons la « plus proche » de R, c'est-à-dire la région R_{succ} telle que

- $R_{\mathsf{succ}} \in \mathsf{Succ}(R)$,
- $\forall v \in R, \forall t \in \mathbb{T}$, si $v + t \notin R$, alors $\exists t' \leq t$ t.q. $v + t' \in R_{\mathsf{succ}}$

La région $R_{\mathsf{succ}} = ((I'_x)_{x\in X}, \prec')$ peut être caractérisée comme suit. Soit

$$Z = \{x \in X \mid I_x \text{ est de la forme } [c]\}$$

Nous distinguons deux cas :

[2]Rappelons qu'un *préordre* est une relation symétrique et transitive. Si, en plus, elle est anti-réflexive, alors, c'est un ordre.

1. si $Z \neq \varnothing$, alors
   - $I'_x = \begin{cases} I_x & \text{si } x \notin Z \\ ]c, c+1[& \text{si } x \in Z \text{ et } I_x = [c] \text{ avec } 0 \leq c < \max_x \\ ]\max_x, +\infty[& \text{si } x \in Z \text{ et } I_x = [\max_x] \end{cases}$
   - $x \prec' y$ si $\begin{cases} x \prec y \text{ ou bien} \\ I_x = [c] \text{ avec } 0 \leq c < \max_x \text{ et } I'_y \text{ est de la forme }]d, d+1[\end{cases}$
2. si $Z = \varnothing$, soit M l'ensemble des éléments maximaux pour $\prec$, *i.e.*

$$M = \{x \in X_0 \mid \forall z \in X_0,\ x \prec z \implies z \prec x\}$$

   Alors,
   - $I'_x = \begin{cases} I_x & \text{si } x \notin M \\ [c+1] & \text{si } x \in M \text{ et } I_x =]c, c+1[\text{ avec } 0 \leq c < \max_x \end{cases}$
   - $\prec'$ est la restriction de $\prec$ à $\{x \in X \mid I'_x$ est de la forme $]d, d+1[\}$.

Il est maintenant facile de vérifier que :

$$\forall v \in R,\ \exists t \in \mathbb{T} \text{ t.q. } v + t \in R_{\mathsf{succ}}$$

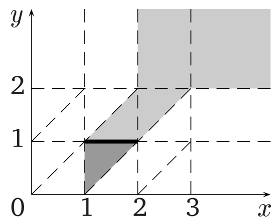
En effet, prenons une valuation v dans R.

1. si $Z \neq \varnothing$, alors soit $\tau = \min\left(\{1 - \mathsf{frac}(v(x)) \mid I_x \text{ est de la forme}]c; c+1[\}\right)$. Alors la valuation $v + \frac{1}{2}\tau$ est dans la région R_{succ}.
2. si $Z = \varnothing$, alors soit $\tau = 1 - \mathsf{frac}(v(x))$ pour tout $x \in M$. Alors la valuation $v + \tau$ est dans R_{succ}.

Nous obtenons donc que l'ensemble $\mathcal{R}_{(\max_x)_{x \in X}}$ vérifie la condition (3.2), ce qui termine la preuve du lemme. □

***Exemple* 3.15** Supposons que nous ayons seulement deux horloges x et y avec comme constantes $\max_x = 3$ et $\max_y = 2$. Alors l'ensemble de régions associé à ces constantes est décrit sur la figure à droite. La région en gris foncé est définie par $I_x =]1; 2[$, $I_y =]0; 1[$ et le préordre $\prec$ est défini par $x \prec y$ et $y \not\prec x$.

Le successeur immédiat de la région en gris foncé est définie par $I_x =]1; 2[$ et $I_y = [1]$ (dessinée par une ligne plus épaisse). Les autres successeurs sont dessinés en gris clair.



Remarque : Les régions que nous venons de définir correspondent aux régions d'Alur et Dill dans leur article [AD94] (voir aussi la partie 2.3.2), le formalisme utilisé étant cependant légèrement différent.

Compatibilité avec les contraintes non diagonales

Les ensembles de régions que nous considérons étant définis, le résultat suivant sur leur compatibilité avec les contraintes non diagonales est immédiat.

Proposition 3.16 *Soit $\mathcal{C} \subseteq \mathcal{C}_{df}(X)$ un ensemble de contraintes tel que pour chaque contrainte $x \sim c$ de $\mathcal{C}$, nous avons $c \leq \max_x$. Alors l'ensemble de régions $\mathcal{R}_{(\max_x)_{x \in X}}$ est compatible avec $\mathcal{C}$.*

Remarquons bien que le résultat n'est plus vrai pour un ensemble quelconque de contraintes incluses dans $\mathcal{C}(X)$. Dans l'exemple précédent 3.15, la région $((]3; +\infty[,]2; +\infty[), \varnothing)$ n'est incluse ni dans $x - y \leq 1$, ni dans $x - y > 1$.

Compatibilité avec les mises à jour simples

Nous étudions maintenant la compatibilité de $\mathcal{R}_{(\max_x)_{x\in X}}$ avec des ensembles de mises à jour. Nous considérons tout d'abord des mises à jour **simples**. Rappelons (voir partie 3.1.1) qu'une mise à jour simple est une mise à jour de la forme $z :\sim c$ ou $z :\sim y + c$ avec y, z des horloges, $\sim \in \{<, \leq, =, \geq, >\}$ et c une constante (entière).

Lemme 3.17 *Soient* $(\max_x)_{x\in X}$ *des constantes entières. L'ensemble de régions* $\mathcal{R}_{(\max_x)_{x\in X}}$ *est compatible avec*

- *toute mise à jour simple de la forme* $z :\sim c$ *si* $\max_z \geq c$*,*
- *toute mise à jour simple de la forme* $z :\sim y + c$ *si* $\max_z \leq \max_y + c$*.*

PREUVE : Supposons que $R = ((I_x)_{x\in X}, \prec)$ soit une région de $\mathcal{R}_{(\max_x)_{x\in X}}$. Rappelons que $\prec$ est un préordre total sur $X_0 = \{x \in X \mid I_x$ est un intervalle de la forme$]c; c+1[\}$. Soit up une mise à jour simple pour z. Nous commençons par caractériser les régions de $\widehat{up}(R)$.

Soit $R' = ((I'_x)_{x\in X}, \prec')$ (où $\prec'$ est un préordre total sur X'_0). Alors R' est dans $\widehat{up}(R)$ si $I'_x = I_x$ pour tout $x \neq z$ et :

si *up* est *z* : $\sim$ c : I'_z peut être n'importe quel intervalle de $\mathcal{I}_z$ qui intersecte $\{\gamma \in \mathbb{T} \mid \gamma \sim c\}$ et

- soit I'_z est de la forme $[d]$ ou $]\max_z; +\infty[$ et donc
  - $X'_0 = X_0 \setminus \{z\}$,
  - $\prec' = \prec \cap (X'_0 \times X'_0)$.
- soit I'_z est de la forme $]d; d+1[$ et donc
  - $X'_0 = X_0 \cup \{z\}$,
  - $\prec'$ est n'importe quel préordre total sur X'_0 qui coïncide avec $\prec$ sur $X'_0 \setminus \{z\}$.

si *up* est *z* : $\sim$ y+c avec c $\in \mathbb{Z}$: I'_z peut être n'importe quel intervalle de $\mathcal{I}_z$ tel qu'il existe $\alpha \in I'_z$, $\beta \in I_y$ avec $\alpha \sim \beta + c$ et

- soit I'_z est de la forme $[d]$ ou $]\max_z; +\infty[$
  - $X'_0 = X_0 \setminus \{z\}$,
  - $\prec' = \prec \cap (X'_0 \times X'_0)$.
- soit I'_z est de la forme $]d; d+1[$,
  - $X'_0 = X_0 \cup \{z\}$,
    - · si $y \notin X_0$, $\prec'$ est un préordre total quelconque sur X'_0 qui coïncide avec $\prec$ sur $X'_0 \setminus \{z\}$,
    - · si $y \in X_0$, alors nous devons faire attention aux valeurs relatives de $\mathsf{frac}(v'(y))$ et $\mathsf{frac}(v'(z))$ lorsque $(I_y + c) \cap I'_z \neq \varnothing$:
      - → soit $(I_y + c) \cap I'_z = \varnothing$ et $\prec'$ est un préordre total quelconque sur X'_0 qui coïncide avec $\prec$ sur $X_0 \setminus \{z\}$,
      - → soit $(I_y + c) \cap I'_z \neq \varnothing$.

        Remarquons que la condition $\max_z \leq \max_y + c$ implique que $I_y + c \subseteq I'_z$. Dans ce cas, $\prec'$ est un préordre total quelconque sur X'_0 qui coïncide avec $\prec$ sur $X'_0 \setminus \{z\}$ et vérifie :

· $z \prec' y$ et $y \prec' z$	si $\sim$ est $=$
· $z \prec' y$ et $y \not\prec' z$	si $\sim$ est $<$
· $z \prec' y$	si $\sim$ est $\leq$
· $y \prec' z$	si $\sim$ est $\geq$
· $z \not\prec' y$ et $y \prec' z$	si $\sim$ est $>$
· $(z \prec' y$ et $y \not\prec' z)$ ou $(z \not\prec' y$ et $y \prec' z)$	si $\sim$ est $\neq$

De cette construction, il est facile de vérifier que la condition (3.3) est vérifiée, *i.e.* que pour toute valuation $v \in R$ et pour toute région $R' \in \widehat{up}(R)$, il existe $v' \in R' \cap up(v)$. En effet, comme up est une mise à jour simple pour z, $v'(x) = v(x)$ pour tout $x \neq z$ et nous avons juste à définir $v'(z)$.

1. si $z \notin X'_0$, alors
   (a) si $I'_z = [c]$, $v'(z)$ vaut bien sûr c.
   (b) si $I'_z =]\max_z, \infty[$, comme $v(y) + c \in I'_z$ et que I'_z est un intervalle ouvert, il existe $\alpha \in I'_z$ tel que $\alpha \sim v(y) + c$ (quel que soit $\sim \in \{=, \neq, <, \leq, \geq, >\}$). Il suffit alors de poser $v'(z) = \alpha$.
2. si $z \in X'_0$, alors
   (a) si $x \prec' z$ et $z \prec' x$ pour un x, alors $v'(z) = d + \mathsf{frac}(v'(x))$ lorsque $I'_z =]d; d+1[$
   (b) si, pour toute horloge x, soit $x \not\prec' z$, soit $z \not\prec' x$, alors $v'(z) = d + \tau$ lorsque $I'_z =]d; d+1[$ et
   $$\max\{\mathsf{frac}(v'(x)) \mid x \prec' z\} < \tau < \min\{\mathsf{frac}(v'(x)) \mid z \prec' x\}$$
   Remarquons que comme le domaine de temps est dense, il existe toujours une infinité de tels τ.

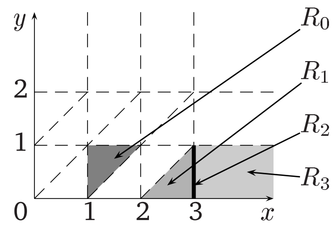
Dans tous les cas, nous avons que $v' \in R' \cap up(R)$ et le lemme est montré. □

***Exemple* 3.18** Supposons que nous ayons deux horloges, x et y, que $\max_x = 3$ et que $\max_y = 2$. L'ensemble de régions $\mathcal{R}_{(\max_x, \max_y)}$ est représenté sur la figure à droite. L'image de la région R_0, définie par $I_x =]1, 2[$, $I_y =]0, 1[$, $x \prec y$, $y \not\prec x$, par la mise à jour $x :> y + 2$ est constituée des trois régions suivantes :

– Région R_1 : $I'_x =]2; 3[$, $I'_y =]0; 1[$, $y \prec' x$ et $x \not\prec' y$

– Région R_2 : $I'_x = [3]$ et $I'_y =]0; 1[$

– Région R_3 : $I'_x =]3; +\infty[$ et $I'_y =]0; 1[$



Considérons maintenant une mise à jour $up = (up_i)_{1 \leq i \leq k}$ où chaque up_i est une mise à jour simple pour une horloge x_i. Soit $\mathcal{R}_{(\max_x)_{x \in X}}$ un ensemble de régions comme défini ci-avant. Il peut arriver que chaque mise à jour up_i soit compatible avec l'ensemble des régions alors que up ne l'est pas, comme l'illustre le contre-exemple suivant.

***Contre-exemple* 3.19** Définissons l'ensemble d'horloges $X = \{x, y, z\}$, les constantes maximales $\max_x = 2, \max_y = \max_z = 1$ et les régions $R = ((]2, \infty[,]1, \infty[, \{1\}), \varnothing)$ et $R' = ((]2, \infty[,]1, \infty[,]1, \infty[), \varnothing)$. Finalement, soit up_1 la mise à jour $z :< x$ et up_2 la mise à jour $z :> y$. Nous avons bien évidemment que

$$\forall v' \in R', \exists v_1, v_2 \in R \text{ t.q. } v_1 \in up_1(v) \text{ et } v_2 \in up_2(v)$$

Mais les deux valuations $(2.3, 1.1, 1)$ et $(2.3, 3.4, 1)$ appartiennent toutes les deux à R et $(2.3, 1.1, 1.8)$ est dans $R' \cap up((2.3, 1.1, 1))$ alors que $up((2.3, 3.4, 1)) = \varnothing$.

Pour que les mises à jour soient compatibles avec les ensembles de régions de la forme $\mathcal{R}_{(\max_x)_{x \in X}}$, nous devons restreindre celles que nous considérons. Du contre-exemple ci-dessus, il apparaît qu'une horloge ne peut pas être remise à jour dans un intervalle dont les bornes inférieures et supérieures dépendent de deux horloges distinctes. De plus, à cause du lemme 3.17, nous devons restreindre les constantes que nous utilisons dans les mises à jour simples. Ces remarques nous conduisent naturellement à la définition qui suit.

Compatibilité avec les mises à jour

Définition 3.20 *Soient* $(\max_x)_{x\in X}$ *des constantes entières. L'ensemble* $\mathcal{U}_{(\max_x)_{x\in X}}$ *est constitué des mises à jour de la forme* $up = \bigwedge_{x\in X} up_x$ *où, pour chaque horloge* $x \in X$, up_x *est une mise à jour de l'horloge* x *définie par l'une des quatre grammaires abstraites suivantes :*

- $det_x \quad ::= \quad x := c \mid x := z + d$
  avec $c, d \in \mathbb{Z}$, $c \leq \max_x$, $z \in X$, *et* $\max_x \leq \max_z + d$
- $inf_x \quad ::= \quad x :\lhd\, c \mid x :< z + d \mid inf_x \wedge inf_x$
  avec $c, d \in \mathbb{Z}$, $\lhd \in \{<, \leq\}$, $c \leq \max_x$, $z \in X$ *et* $\max_x \leq \max_z + d$
- $sup_x \quad ::= \quad x :\rhd\, c \mid x :> z + d \mid sup_x \wedge sup_x$
  avec $c, d \in \mathbb{Z}$, $\rhd \in \{>, \geq\}$, $c \leq \max_x$, $z \in X$ *et* $\max_x \leq \max_z + d$
- $int_x \quad ::= \quad x :\in (c; d) \mid x :\in (c; z + d') \mid x :\in (z + c'; d) \mid x :\in (z + c'; z + d')$
  où (et) sont soit [soit], $c, c', d, d' \in \mathbb{Z}$, $c, d \leq \max_x$, $z \in X$,
  $c, c' \leq \max_x$, $\max_x \leq \max_z + d'$ *et* $\max_x \leq \max_z + c'$

La *base* d'une mise à jour $up = \bigwedge_{x\in X} up_x$ de $\mathcal{U}_{(\max_x)_{x\in X}}$ est intuitivement l'ensemble Y des horloges qui peuvent être modifiées par up. Formellement, cet ensemble Y est défini par son complémentaire :

$$X \setminus Y = \{z \in X \mid up_z \text{ est égal à } z := z\}$$

La première étape pour prouver la compatibilité de $\mathcal{R}_{(\max_x)_{x\in X}}$ et de $\mathcal{U}_{(\max_x)_{x\in X}}$ est donnée par le lemme suivant. Sa preuve est similaire à celle du lemme 3.17 et n'est donc pas écrite.

Lemme 3.21 *Soient* $(\max_x)_{x\in X}$ *des constantes entières. L'ensemble de régions* $\mathcal{R}_{(\max_x)_{x\in X}}$ *est compatible avec toute mise à jour de* $\mathcal{U}_{(\max_x)_{x\in X}}$ *ayant une base réduite à un singleton.*

Nous pouvons maintenant établir notre résultat principal concernant la compatibilité des ensembles de régions et des ensembles de mises à jour, dans le cas des automates temporisés avec contraintes non diagonales.

Proposition 3.22 *Soit* $(\max_x)_{x\in X}$ *des constantes entières. Alors l'ensemble de régions* $\mathcal{R}_{(\max_x)_{x\in X}}$ *est compatible avec l'ensemble de mises à jour* $\mathcal{U}_{(\max_x)_{x\in X}}$.

PREUVE : Soient $R = ((I_y)_{y\in X}, \prec)$, $R' = ((I'_y)_{y\in X}, \prec')$ deux régions de $\mathcal{R}_{(\max_x)_{x\in X}}$ et up une mise à jour de $\mathcal{U}_{(\max_x)_{x\in X}}$ telle que $R' \in \widehat{up}(R)$ *i.e.* il existe des valuations $v \in R$ et $v' \in R'$ telles que $v' \in up(v)$. Pour chaque horloge x, soit v_x la valuation définie par :

$$v_x(y) = \begin{cases} v(y) & \text{si } y \neq x \\ v'(x) & \text{si } y = x \end{cases}$$

et soit $R_x = ((I_y^{(x)})_{y\in X}, \prec^{(x)})$ l'unique région de $\mathcal{R}_{(\max_x)_{x\in X}}$ contenant v_x.

Soit maintenant w une valuation dans R. Par le lemme 3.21, $\mathcal{R}_{(\max_x)_{x\in X}}$ est compatible avec up_x, donc, pour chaque horloge x, il existe une valuation $w_x \in up_x(w) \cap R_x$. Nous définissons maintenant la valuation w' en posant

$$w'(y) = w_y(y) \text{ pour chaque horloge } y.$$

De par la définition d'une mise à jour, il vient tout de suite que $w' \in up(w)$. Nous montrons aussi que $w' \in R'$. En effet, pour chaque horloge y, $w'(y) = w_y(y) \in I_y^{(y)} = I'_y$. Il reste à montrer que la suite $(\mathsf{frac}(w'(x)))_{x\in X}$ vérifie les conditions données par le préordre $\prec'$. Dans ce but, il est suffisant de montrer que le préordre $\prec'$ (qui est *a priori* donné par la valuation v') peut être défini à partir de $\prec$ et de la suite $(\prec^{(x)})_{x\in X}$.

À partir des constructions données dans le lemme 3.17, qui peuvent être étendues pour prouver le lemme 3.21, il est facile de vérifier que le préordre $\prec'$ peut être calculé comme suit.

Soit X' une copie disjointe de l'ensemble d'horloges X. Nous commençons par définir une suite $(\overline{\prec}^{(x)})_{x\in X}$ de préordres sur l'ensemble $X \cup X'$. Intuitivement, $\overline{\prec}^{(x)}$ est obtenu à partir de $\prec^{(x)}$ en remplaçant simplement l'horloge x par sa copie x'. Formellement,

$$\begin{array}{llll} \forall y,z \in X\setminus\{x\}, & y \,\overline{\prec}^{(x)}\, z & \text{si} & y \prec^{(x)} z \\ \forall y \in X\setminus\{x\}, & y \,\overline{\prec}^{(x)}\, x' & \text{si} & y \prec^{(x)} x \\ \forall y \in X\setminus\{x\}, & x' \,\overline{\prec}^{(x)}\, y & \text{si} & x \prec^{(x)} y \end{array}$$

Nous définissons ensuite $\overline{\prec}$ comme l'union de tous les $\overline{\prec}^{(x)}$. Il est clair que $\overline{\prec}$ est encore un préordre sur $X \cup X'$. Maintenant, $\prec'$ peut être obtenu à partir de $\overline{\prec}$ en commençant par le restreindre à $X' \times X'$ puis en transformant chaque horloge x' en sa copie x. Le préordre $\prec'$ peut donc être calculé à partir des préordres $\prec$ et $(\prec^{(x)})_{x\in X}$, donc comme $(\mathsf{frac}(w(x)))_{x\in X}$ vérifie le préordre $\prec$ et chaque $(\mathsf{frac}(w_x(y)))_{y\in X}$ vérifie le préordre $\prec^{(x)}$, nous obtenons que $(\mathsf{frac}(w'(x)))_{x\in X}$ vérifie le préordre $\prec'$. Nous en concluons que la valuation w' est dans R'.

Nous avons donc montré que si $R' \in \widehat{up}(R)$, alors pour toute valuation $w \in R$, il existe une valuation $w' \in up(w) \cap R'$, ce qui correspond à la condition (3.3). $\square$

À partir du théorème 3.12 et des propositions 3.16 et 3.22, nous obtenons immédiatement le théorème qui suit, qui est notre principal résultat (effectif) concernant la décidabilité des automates temporisés avec mises à jour lorsque nous nous restreignons à des contraintes non diagonales.

Théorème 3.23 *Soient :*

- $\mathcal{C} \subseteq \mathcal{C}_{df}(X)$ *un ensemble fini de contraintes d'horloges non diagonales,*
- *pour chaque horloge* x*, une constante* $\max_x$ *telle que pour toute contrainte* $x \sim c$ *de* $\mathcal{C}$*, nous avons* $c \leq \max_x$*,*
- $\mathcal{U} \subseteq \mathcal{U}_{(\max_x)_{x\in X}}$ *un ensemble fini de mises à jour.*

Alors la classe Aut($\mathcal{C},\mathcal{U}$) *est décidable.*

En pratique, étant donné un automate temporisé $\mathcal{A}$, il serait intéressant de pouvoir savoir s'il existe des constantes $(\max_x)_{x\in X}$ telles que l'ensemble des mises à jour de $\mathcal{A}$ soient dans $\mathcal{U}_{(\max_x)_{x\in X}}$ et pour toute contrainte $x \sim c$ de $\mathcal{A}$, $c \leq \max_x$. Nous allons décrire une procédure permettant de répondre à cette question.

Comment savoir si un automate appartient à une classe décidable ?

Soient $\mathcal{C} \subseteq \mathcal{C}_{df}(X)$ un ensemble de contraintes non diagonales et $\mathcal{U} \subseteq \mathcal{U}(X)$ un ensemble de mises à jour tels que

$$up = \bigwedge_{x\in X} up_x \in \mathcal{U} \implies \text{pour tout } x,\ up_x \in \{det_x, inf_x, sup_x, int_x\} \text{ où :} \qquad (\Diamond_{df})$$

$$\begin{cases} det_x ::= x := c \ \mid \ x := z + d \text{ avec } c \in \mathbb{N},\ d \in \mathbb{Z} \text{ et } z \in X \\ inf_x ::= x :\lhd c \ \mid \ x :< z + d \ \mid \ inf_x \wedge inf_x \text{ avec } \lhd \in \{<, \leq\},\ c \in \mathbb{N},\ d \in \mathbb{Z} \text{ et } z \in X \\ sup_x ::= x :\rhd c \ \mid \ x :> z + d \ \mid \ sup_x \wedge sup_x \text{ avec } \rhd \in \{>, \geq\},\ c \in \mathbb{N},\ d \in \mathbb{Z} \text{ et } z \in X \\ int_x ::= x :\in (c; d) \ \mid \ x :\in (c; z + d) \ \mid \ x :\in (z + c; d) \ \mid \ x :\in (z + c; z + d) \\ \qquad\qquad\qquad \text{où (et) sont soit [soit], } z \text{ est une horloge et } c,\ d \text{ sont dans } \mathbb{Z}. \end{cases}$$

Nous souhaitons déterminer à quelles conditions il existe des constantes $(\max_x)_{x \in X}$ telles que $\mathcal{U} \subseteq \mathcal{U}_{(\max_x)_{x\in X}}$ et $\mathcal{C}$ compatible avec $\mathcal{R}_{(\max_x)_{x\in X}}$.

Si le système Diophantien d'inéquations linéaires sur les variables $(\max_x)_{x \in X}$

$$\{c \leq \max_x \mid x \sim c \in \mathcal{C} \text{ ou } x :\sim c \in \mathcal{U}\} \cup \{\max_z \leq \max_y + c \mid z :\sim y + c \in \mathcal{U}\} \qquad (\mathcal{S}_{df})$$

a une solution, alors $\mathcal{U} \subseteq \mathcal{U}_{(\max_x)_{x\in X}}$ et $\mathcal{C}$ est compatible avec $\mathcal{R}_{(\max_x)_{x\in X}}$, donc, dans ce cas, la classe d'automates *Aut*($\mathcal{C}$,$\mathcal{U}$) est décidable (d'après le théorème 3.23). De plus, si $(\max_x)_{x \in X}$ est solution du système, et si $\mathcal{A} \in$ *Aut*($\mathcal{C}$,$\mathcal{U}$), alors nous pouvons décider du vide de $L(\mathcal{A})$ grâce à l'automate des régions $\Gamma_{\mathcal{R}_{(\max_x)_{x\in X}}}(\mathcal{A})$ (proposition 3.11).

Si toutes les constantes c apparaissant dans les mises à jour $x :\sim y + c$ sont positives ou nulles, le système ($\mathcal{S}_{df}$) admet toujours une solution. Sinon, grâce aux résultats de [Dom91], il est possible de décider de l'existence d'une telle solution.

Ce qui précède donne donc une procédure permettant de décider si un automate appartient à une classe décidable ou pas. Si c'est le cas, cela nous permet de calculer un ensemble de régions $\mathcal{R}$ « adapté » à l'automate, c'est-à-dire tel que $L(\Gamma_{\mathcal{R}}(\mathcal{A})) = \mathsf{Untime}(L(\mathcal{A}))$. Cet ensemble de régions nous sera utile dans le chapitre 8, dans lequel nous proposerons un algorithme implémentable pour tester l'accessibilité dans ces automates.

Remarque : Nous avons montré dans la partie 3.2 qu'utiliser des mises à jour de la forme $z := z - 1$ même avec des contraintes non diagonales conduit à l'indécidabilité du problème du vide. Nous devons donc noter que, heureusement, ce n'est pas en contradiction avec notre construction car il nous faudrait alors résoudre un système d'équations comprenant l'équation $\max_z \leq \max_z - 1$. Bien évidemment, un tel système ne peut pas avoir de solution.

Complexité. Une région comme définie à la page 55 peut être codée en espace polynômial. En effet, pour chaque horloge, il suffit de stocker au maximum deux entiers (les bornes de l'intervalle où se trouve cette horloge), et pour chaque couple d'horloges, leur ordre relatif pour le préordre définissant la région. Le test du vide pour l'automate des régions est donc PSPACE, donc celui de l'automate initial aussi car l'automate des régions peut être construit à la volée. Comme cette classe d'automates temporisés contient tous les automates temporisés classiques, nous avons que le problème du vide pour les automates, qui appartiennent à une classe décidable décrite ci-avant, est PSPACE-dur (voir la partie 2.3.2) et donc PSPACE-complet.

3.3.3 Classes décidables - Automates avec contraintes générales

Nous nous intéressons maintenant à la classe d'automates temporisés avec des contraintes d'horloges générales. Comme nous l'avons remarqué juste après la proposition 3.16, les contraintes diagonales ne sont pas compatibles avec les ensembles de régions que nous avons définis dans la partie précédente. En effet, si nous revenons à l'exemple 3.15 (page 56), la région correspondant aux équations $x > 3 \wedge y > 2$ n'est incluse ni dans $x - y \leq 1$, ni dans $x - y > 1$. Il nous faut donc définir de nouveaux ensembles de régions.

Définition des régions

Nous considérons pour chaque couple d'horloges de X, (x, y), une constante entière $\max(y, z)$ et nous définissons l'ensemble

$$\begin{array}{rcl} \mathcal{J}_{y,z} & = & \{]-\infty; -\max_{z,y}[\} \\ & & \cup \{[d] \mid -\max_{z,y} \le d \le \max_{y,z}\} \\ & & \cup \{]d; d+1[\mid -\max_{z,y} \le d < \max_{y,z}\} \\ & & \cup \{]\max_{y,z}; +\infty[\} \end{array}$$

La région définie par l'uplet $R = ((I_x)_{x \in X}, (J_{x,y})_{x,y \in X}, \prec)$ où

- $\forall x \in X$, $I_x \in \mathcal{I}_x$,
- $\forall (y, z) \in X_\infty$, $J_{y,z} \in \mathcal{J}_{y,z}$ où $X_\infty = \{(y, z) \in X^2 \mid I_y$ ou I_z est non borné$\}$,
- $\prec$ est un préordre total sur $X_0 = \{x \in X \mid I_x$ est un intervalle de la forme $]c, c+1[\}$.

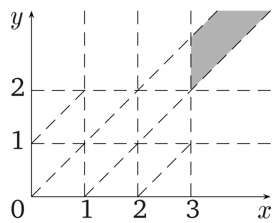
est le sous-ensemble de $\mathbb{T}^X$ tel que :

$$\left\{ v \in \mathbb{T}^X \;\middle|\; \begin{array}{l} \forall x \in X,\ v(x) \in I_x, \\ \forall x,\ y \in X_0,\ x \prec y \iff \mathsf{frac}(v(x)) \le \mathsf{frac}(v(y)), \\ \forall (y, z) \in X_\infty,\ v(y) - v(z) \in J_{y,z} \end{array} \right\}$$

L'ensemble fini $\mathcal{R}_{(\max_x)_{x \in X}, (\max_{y,z})_{y,z \in X}}$ de tous ces sous-ensembles de $\mathbb{T}^X$ forme une partition de $\mathbb{T}^X$. Par une preuve similaire a celle du lemme 3.14, il est facile de vérifier que cet ensemble vérifie aussi la condition (3.2), *i.e.* que le lemme suivant est vérifié :

Lemme 3.24 *L'ensemble $\mathcal{R}_{(\max_x)_{x \in X}, (\max_{y,z})_{y,z \in X}}$ est un ensemble de régions.*

***Exemple* 3.25** Supposons que nous ayons uniquement deux horloges, x et y, et que les constantes maximales soient $\max_x = 3$, $\max_y = 2$, $\max_{x,y} = 1$ et $\max_{y,x} = 0$. Alors l'ensemble des régions associées à ces constantes est décrit sur le figure à droite. La région grise est définie par $I_x =]3; +\infty[$, $I_y =]2; +\infty[$ et $-1 < y - x < 0$ (*i.e.* $J_{y,x}$ est $]-1; 0[$).



Notons que les ensembles de régions dans le cas des contraintes non diagonales peuvent apparaître comme des cas particuliers des ensembles de régions que nous venons de décrire en étendant la définition précédente à des constantes $\max_{x,y}$ qui valent $-\infty$. Nous appellerons ces ensembles des ensembles de régions *non diagonaux*.

Compatibilité avec les contraintes générales

Encore une fois, la compatibilité de cet ensemble de régions avec des ensembles de contraintes est facile et immédiate à obtenir.

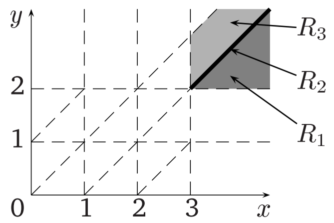
Proposition 3.26 *Soit $\mathcal{C} \subseteq \mathcal{C}(X)$ un ensemble de contraintes tel que pour toute contrainte $x \sim c$ de $\mathcal{C}$, $c \le \max_x$ et pour toute contrainte $x - y \sim c$ de $\mathcal{C}$, $-\max_{y,x} \le c \le \max_{x,y}$. Alors l'ensemble de régions $\mathcal{R}_{(\max_x)_{x \in X}, (\max_{y,z})_{y,z \in X}}$ est compatible avec $\mathcal{C}$.*

Compatibilité avec les mises à jour

Comme dans le cas des contraintes non diagonales, nous introduisons maintenant un ensemble de mises à jour, qui dépend des constantes $(\max_x)_{x\in X}$ et $(\max_{y,z})_{y,z\in X}$ et qui sera compatible avec l'ensemble des régions que nous venons juste de définir.

Notons que, de par les résultats d'indécidabilité de la partie 3.2, nous devons restreindre de manière drastique l'ensemble des mises à jour que nous utilisons si nous voulons préserver la décidabilité du modèle.

***Exemple* 3.27** Si nous considérons la mise à jour $y := y + 1$ et l'ensemble de régions décrit sur la figure à droite, les images de la région R_1 par $y := y + 1$ sont les régions R_1, R_2 et R_3. Mais nous ne pouvons pas atteindre la région R_1 (respectivement R_2, respectivement R_3) de n'importe quel point de la région R_1. Ainsi, cet ensemble de régions n'est pas compatible avec la mise à jour $y := y + 1$.



Définition 3.28 *Soient* $(\max_x)_{x\in X}, (\max_{y,z})_{y,z\in X}$ *des constantes entières. L'ensemble*

$$\mathcal{U}_{(\max_x)_{x\in X},(\max_{y,z})_{y,z\in X}}$$

est constitué des mises à jour de la forme $up = \bigwedge_{x\in X} up_x$ *où, pour chaque horloge* $x \in X$*,* up_x *est une mise à jour locale de l'une des formes suivantes :*

- $x :\lhd c$ *avec* $\lhd \in \{=, <, \leq\}$*,* $c \in \mathbb{N}$*,* $c \leq \max_x$
  et, pour toute horloge y, $\max_y \geq c + \max_{y,x}$
- $x := y$ *avec* $y \in X$*, et* $\max_x \leq \max_y$
  et, pour toute horloge z, $\max_{z,x} \leq \max_{z,y}$*,* $\max_{x,z} \leq \max_{y,z}$

Dans le cadre des automates temporisés avec contraintes générales, la compatibilité entre les mises à jour et les régions est donnée par le résultat suivant.

Proposition 3.29 *Soient* $(\max_x)_{x\in X}, (\max_{y,z})_{y,z\in X}$ *des constantes entières. L'ensemble des régions* $\mathcal{R}_{(\max_x)_{x\in X},(\max_{y,z})_{y,z\in X}}$ *est compatible avec l'ensemble des mises à jour* $\mathcal{U}_{(\max_x)_{x\in X},(\max_{y,z})_{y,z\in X}}$.

PREUVE : Comme dans le cas des automates temporisés avec contraintes non diagonales, nous allons commencer par étudier le cas particulier des mises à jour **simples**.

Supposons que $R = ((I_x)_{x\in X}, (J_{x,y})_{x,y\in X}, \prec)$ où $\prec$ est un préordre total sur X_0 et que up est une mise à jour simple pour z. Alors la région $R' = ((I'_x)_{x\in X}, (J'_{x,y})_{x,y\in X}, \prec')$ (où $\prec'$ est un préordre total sur X'_0) est dans $\widehat{up}(R)$ 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 *up* est *z* : $\sim$ c : I'_z peut être n'importe quel intervalle de $\mathcal{J}_z$ qui intersecte $\{\gamma \in \mathbb{T} \mid \gamma \sim c\}$ et

- soit I'_z est de la forme $[d]$ et alors
  - $X'_0 = X_0 \setminus \{z\}$,
  - $\prec' = \prec \cap (X'_0 \times X'_0)$,
  - $X'_\infty = \quad \{(x,y) \in X_\infty \mid (x \neq z \wedge y \neq z)\} \cup \{(z,y) \mid I_y =]\max_y, \infty[\}$
    $\cup \{(x,z) \mid I_x =]\max_x, \infty[)\}$

  et pour tout couple $(x,y) \in X'_\infty$:

· $J'_{x,y} = J_{x,y}$ si $x \neq z$ et $y \neq z$

· $J'_{x,z} =]\max_{x,z}, \infty[$.

Remarque : Si v est une valuation telle que $\max_x < v(x)$ et $v(z) \lhd c$ avec $\lhd \in \{=, <, \leq\}$, alors nous avons que $\max_x - c < v(x) - v(z)$. Ainsi, de l'hypothèse $\max_x \geq c + \max_{x,z}$, nous obtenons $\max_{x,z} < v(x) - v(z)$.

· $J'_{z,y} =]-\infty, -\max_{z,y}[$.

Remarque : Si v est une valuation telle que $\max_y < v(y)$ et $v(z) \lhd c$ avec $\lhd \in \{=, <, \leq\}$, nous avons que $v(z) - v(y) < c - c_y$. Ainsi, de l'hypothèse $\max_y \geq c + \max_{z,y}$, nous obtenons que $v(z) - v(y) < -\max_{z,y}$.

- soit I'_z est de la forme $]d; d+1[$ et donc
  - $X'_0 = X_0 \cup \{z\}$,
  - $\prec'$ est un préordre total quelconque sur X'_0 qui coïncide avec $\prec$ sur $X'_0 \setminus \{z\}$,
  - $X'_\infty = \{(x,y) \in X_\infty \mid (x \neq z \wedge y \neq z)\} \cup \{(z,y) \mid I_y =]\max_y, \infty[\}$ $\cup \{(x,z) \mid I_x =]\max_x, \infty[\}$

    et pour tout couple $(x,y) \in X'_\infty$,

    · $J'_{x,y} = J_{x,y}$ si $x \neq z$ et $y \neq z$,

    · $J'_{x,z} =]\max_{x,z}, \infty[$,

    · $J'_{z,y} =]-\infty, -\max_{z,y}[$.

si *up* est *z* : $= y$: Définissons tout d'abord I'_z.

- si $I_y = [d]$, $I'_z = [d]$ où $d \leq \max_z$, $I'_z =]\max_z, \infty[$ sinon
- si $I_y =]d, d+1[$, $I'_z =]d, d+1[$ où $d < c_z$, $I'_z =]\max_z, \infty[$ sinon
- si $I_y =]\max_y, \infty[$, $I'_z =]\max_z, \infty[$ (par hypothèse, $\max_z \leq \max_y$)

Maintenant,

- soit I'_z est de la forme $[d]$ (et donc $I_y = [d]$ par ce qui précède)
  - $X'_0 = X_0 \cup \{z\}$,
  - $\prec' = \prec \cap (X'_0 \times X'_0)$,
  - $X'_\infty = \{(x,x') \in X_\infty \mid (x \neq z \wedge x' \neq z)\} \cup \{(z,x') \mid I_{x'} =]\max_{x'}, \infty[\}$ $\cup \{(x,z) \mid I_x =]\max_x, \infty[\}$

    et pour tout couple $(x,x') \in X'_\infty$,

    · $J'_{x,x'} = J_{x,x'}$ si $x \neq z$ et $x' \neq z$

    · $J'_{z,x'}$ est l'unique intervalle de $\mathcal{J}_{z,x'}$ qui contient $J_{z,x'}$.

    Remarquons que l'unicité provient de l'hypothèse que $\max_{z,x'} \leq \max_{y,x'}$

    · $J'_{x,z}$ est l'unique intervalle de $\mathcal{J}_{x,z}$ qui contient $J_{x,z}$.

    Remarquons que l'unicité provient de l'hypothèse que $\max_{x,z} \leq \max_{x,y}$
- soit I'_z est de la forme $]d; d+1[$ (et donc $I_y =]d, d+1[$)
  - $X'_0 = X_0 \cup \{z\}$,
  - $\prec'$ est n'importe quel préordre total sur X'_0 qui coïncide avec $\prec$ sur $X'_0 \setminus \{z\}$ et tel que $z \prec' y$ et $y \prec' z$,
  - L'ensemble X'_∞ et les intervalles $J'_{x,x'}$ sont définis comme dans le cas précédent $I'_z = [d]$.
- soit I'_z est de la forme $]\max_z, \infty[$
  - $X'_0 = X_0 \setminus \{z\}$,
  - $\prec' = \prec \cap (X'_0 \times X'_0)$,
  - $X'_\infty = X_\infty \cup \{(x,z),(z,x) \mid x \in X\}$ et $J'_{x,x'} = J_{x,x'}$ si $x \neq z$ et $x' \neq z$. 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 $I_x = [f]$, $I_y = [g]$, alors
$$J'_{z,x} = \begin{cases} [g-f] & \text{si } -\max_{x,z} \leq g-f \leq \max_{z,x} \\ ]\max_{z,x}, \infty[& \text{si } \max_{z,x} < g-f \\ ]-\infty, -\max_{x,z}[& \text{si } g-f < -\max_{x,z} \end{cases}$$
2. si $I_x = [f]$, $I_y =]g; g+1[$, alors
$$J'_{z,x} = \begin{cases}]g-f-1; g-f[& \text{si } -\max_{x,z} \leq g-f-1 < \max_{z,x} \\ ]\max_{z,x}, \infty[& \text{si } \max_{z,x} \leq g-f-1 \\ ]-\infty, -\max_{x,z}[& \text{si } g-f-1 < -\max_{x,z} \end{cases}$$
3. si $I_x = [f]$, $I_y =]\max_y, \infty[$, alors
$J'_{z,x}$ est l'unique intervalle de $\mathcal{J}_{z,x}$ qui contient $J_{y,x}$.
Remarque : L'unicité provient de l'hypothèse que $\max_{z,x} \leq \max_{y,x}$ et $\max_{x,z} \leq \max_{x,y}$.
4. si $I_x =]f; f+1[$, $I_y = [g]$, ce cas est identique aux deux cas ci-dessus.
5. si $I_x =]f; f+1[$, $I_y =]g; g+1[$, alors
   (a) si $x \prec z \wedge y \prec x$, alors
$$J'_{z,x} = \begin{cases} [g-f] & \text{si } -\max_{x,z} \leq g-f \leq \max_{z,x} \\ ]\max_{z,x}, \infty & \text{si } \max_{z,x} < g-f \\ ]-\infty, -\max_{x,z}[& \text{si } g-f < -\max_{x,z} \end{cases}$$
   (b) si $x \prec z \wedge z \not\prec x$, alors
$$J'_{z,x} = \begin{cases}]g-f; g-f+1[& \text{si } -\max_{x,z} \leq g-f < \max_{z,x} \\ ]\max_{z,x}, \infty[& \text{si } \max_{z,x} \leq g-f \\ ]-\infty, -\max_{x,z}[& \text{si } g-f < -\max_{x,z} \end{cases}$$
   (c) si $x \not\prec z \wedge z \prec x$, alors
$$J'_{z,x} = \begin{cases}]g-f-1; g-f[& \text{si } -\max_{x,z} \leq g-f-1 < \max_{z,x} \\ ]\max_{z,x}, \infty[& \text{si } \max_{z,x} \leq g-f-1 \\ ]-\infty, -\max_{x,z}[& \text{si } g-f-1 < -\max_{x,z} \end{cases}$$
6. si $I_x =]f; f+1[$, $I_y =]\max_y; \infty[$, ce cas est identique aux trois cas précédents.
7. si $I_x =]\max_x; \infty[$, ce cas est identique aux trois cas précédents.

En utilisant cette construction, il est facile de montrer, d'une manière similaire à la preuve du lemme 3.17, que la condition (3.3) est vérifiée lorsque nous considérons des mises à jour simples. L'extension aux mises à jour de $\mathcal{U} \subseteq \mathcal{U}_{(\max_x)_{x \in X}, (\max_{y,z})_{y,z \in X}}$ (sous l'hypothèse de la proposition) est obtenue en utilisant la même technique que celle de la proposition 3.22. □

Notre résultat principal effectif concernant la décidabilité du modèle des automates temporisés avec contraintes générales est donné par le théorème suivant. Sa preuve provient directement du théorème 3.12 et des propositions 3.26 et 3.29.

Théorème 3.30 *Soient :*

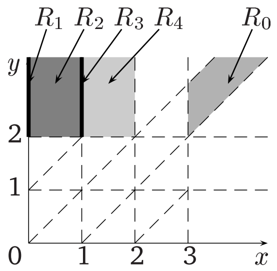
– $\mathcal{C} \subseteq \mathcal{C}(X)$ *un ensemble fini de contraintes d'horloges générales,*

– *pour chaque horloge* x*, une constante* $\max_x$ *telle que pour toute contrainte* $x \sim c$ *de* $\mathcal{C}$*, nous avons* $c \leq \max_x$*,*

– *pour chaque couple d'horloges* (x, y) *une constante* $\max_{x,y}$ *telle que pour toute contrainte* $x - y \sim c$ *de* $\mathcal{C}$*, nous avons* $c \leq \max_{x,y}$*,*

– $\mathcal{U} \subseteq \mathcal{U}_{(\max_x)_{x \in X}, (\max_{x,y})_{x,y \in X}}$ *un ensemble fini de mises à jour.*

Alors la classe Aut$(\mathcal{C}, \mathcal{U})$ *est décidable.*



***Exemple* 3.31** Considérons les régions dessinées sur la figure de gauche. Nous souhaitons calculer les successeurs par la mise à jour $x :< 2$ de la région R_0. Les quatre successeurs sont dessinés sur la figure. Leurs équations sont :

– Région R_1 : $I'_x = [0]$ et $I'_y =]2; +\infty[$

– Région R_2 : $I'_x =]0; 1[$, $I'_y =]2; +\infty[$ et $J_{y,x} =]1; +\infty[$

– Région R_3 : $I'_x = [1]$ et $I'_y =]2; +\infty[$

– Région R_4 : $I'_x =]1; 2[$, $I'_y =]2; +\infty[$ et $J_{y,x} =]1; +\infty[$

La condition que nous imposons sur les constantes maximales revient à dire que les droites diagonales de l'ensemble de régions n'intersectent pas l'union des régions R_1, R_2, R_3 et R_4.

En pratique, comme dans le cas des automates temporisés avec contraintes non diagonales, il serait utile de pouvoir décider, étant donné un automate temporisé, s'il appartient ou pas à une des classes décidables décrites dans le théorème précédent. Nous allons décrire une procédure qui permet de répondre à cette question.

Comment savoir si un automate appartient à une classe décidable ?

Soient $\mathcal{C} \subseteq \mathcal{C}(X)$ un ensemble fini de contraintes générales et $\mathcal{U} \subseteq \mathcal{U}(X)$ un ensemble fini de mises à jour telles que :

$$up = \bigwedge_{x \in X} up_x \in \mathcal{U} \quad \Longrightarrow \quad \forall x \in X,\ up_x \in \begin{array}{l} \{x := c, x :< c, x :\leq c \mid c \in \mathbb{N}\} \\ \cup\ \{x := y \mid y \in X\} \end{array} \qquad (\diamondsuit_{gen})$$

Nous cherchons à savoir s'il existe des constantes $(\max_x)_{x \in X}$ et $(\max_{x,y})_{x,y \in X}$ telles que

$$\mathcal{U} \subseteq \mathcal{U}_{(\max_x)_{x \in X}, (\max_{x,y})_{x,y \in X}}$$

et $\mathcal{C}$ compatible avec $\mathcal{R}_{(\max_x)_{x \in X}, (\max_{x,y})_{x,y \in X}}$.

Si le système Diophantien d'inéquations linéaires suivant (les variables de ce système sont $\alpha = ((\max_x)_{x \in X}; (\max_{x,y})_{x,y \in X})$) :

$$\begin{array}{ll} & \{c \leq \max_x \mid x \sim c \in \mathcal{C}\} \\ \cup & \{c \leq \max_{x,y} \mid x - y \sim c \in \mathcal{C}\} \\ \cup & \{c \leq \max_x,\ \max_z \geq c + \max_{z,x} \mid x :< c \text{ ou } x :\leq c \text{ ou } x := c\ \in \mathcal{U},\ \text{et } z \in X\} \\ \cup & \{\max_x \leq \max_y,\ \max_{z,y} \geq \max_{z,x},\ \max_{x,z} \leq \max_{y,z} \mid x := y \in \mathcal{U} \text{ et } z \in X\} \end{array} \qquad (\mathcal{S}_{gen})$$

a une solution, alors $\mathcal{U} \subseteq \mathcal{U}_\alpha$ et $\mathcal{C}$ est compatible avec $\mathcal{R}_\alpha$. La classe *Aut*($\mathcal{C}$,$\mathcal{U}$) est alors décidable (théorème 3.30). De plus, si α est solution de ce système, et si $\mathcal{A} \in$ *Aut*($\mathcal{C}$,$\mathcal{U}$), nous pouvons décider du vide de $L(\mathcal{A})$ grâce à l'automate des régions $\Gamma_{\mathcal{R}_\alpha}(\mathcal{A})$.

Il est facile de constater que le système $(\mathcal{S}_{gen})$ a toujours une solution. Nous obtenons donc le théorème suivant :

Théorème 3.32 *Soient $\mathcal{C} \subseteq \mathcal{C}(X)$ un ensemble de contraintes générales et $\mathcal{U}$ un ensemble de mises à jour défini comme dans $(\diamondsuit_{gen})$. Alors la classe d'automates temporisés Aut($\mathcal{C}$,$\mathcal{U}$) est décidable.*

En outre, résoudre le système $(\mathcal{S}_{gen})$ permet de calculer un ensemble de régions « adapté » à l'automate, ceci nous sera utile dans le chapitre 8 lorsque nous proposerons un algorithme implémentable pour tester l' accessibilité des automates temporisés qui font partie d'une classe décidable.

Complexité. Comme dans le cas non diagonal, une région peut être codée en espace polynômial (deux entiers pour chaque horloge, puis deux autres entiers pour chaque couple d'horloges). Donc, par un raisonnement similaire à celui fait dans le cas non diagonal, le problème du vide pour les classes décidables décrites ci-avant d'automates temporisés avec contraintes générales est un problème PSPACE-complet.

3.3.4 Conclusion et discussion

Lors de cette étude, nous avons délimité de manière assez fine les classes décidables et indécidables. Nous résumons ces résultats dans le tableau 3.4.

<table>
<tr><th></th><th>Contraintes non diagonales</th><th>Contraintes générales</th></tr>
<tr><td>$x := c,\, x := y$</td><td rowspan="3">PSPACE-complet</td><td>PSPACE-complet</td></tr>
<tr><td>$x := x + 1$</td><td rowspan="3">Indécidable</td></tr>
<tr><td>$x := y + c$</td></tr>
<tr><td>$x := x - 1$</td><td>Indécidable</td></tr>
<tr><td>$x :< c$</td><td rowspan="4">PSPACE-complet</td><td>PSPACE-complet</td></tr>
<tr><td>$x :> c$</td><td rowspan="4">Indécidable</td></tr>
<tr><td>$x :\sim y + c$</td></tr>
<tr><td>$y + c <: x :< y + d$</td></tr>
<tr><td>$y + c <: x :< z + d$</td><td>Indécidable</td></tr>
</table>

avec $\sim \in \{\leq, <, >, \geq\}$ et $c,\ d \in \mathbb{Q}^+$.

Tableau 3.4: Résultats de décidabilité

Ce tableau donne la limite entre les sous-classes décidables et les sous-classes indécidables des automates temporisés avec mises à jour. Nous n'avons pas précisé les conditions dont nous avons parlé auparavant pour ne pas surcharger l'écriture, nous avons plutôt voulu donner une vision globale des résultats de décidabilité.

Remarque : Soient $\alpha = ((\max_x)_{x\in X},\, (\max_{y,z})_{y,z\in X})$ des constantes entières. Il existe des mises à jour qui n'appartiennent pas à $\mathcal{U}_\alpha$ mais qui sont cependant compatibles avec l'ensemble de régions $\mathcal{R}_\alpha$. C'est par exemple le cas pour les mises à jour suivantes :

– $z := y + c$ lorsque $\max_z \leq \max_y + c$ et pour chaque horloge x, $\max_{x,z} \leq \max_{x,y} - c$ et $\max_{z,x} \leq \max_{y,x} + c$,
– $z :> c$ lorsque $c \leq \max_z$ et pour chaque horloge $x,\ \max_z - \max_x \geq \max_{z,x}$.

Nous n'avons pas considéré ces mises à jour dans notre étude, car les conditions sous lesquelles nous obtenons la décidabilité sont assez restrictives. En outre, il serait toujours possible de raffiner encore et encore les classes pour obtenir d'autres sous-classes décidables. La solution est alors, lorsque nous avons un modèle qui utilise des mises à jour non répertoriées dans « nos » classes décidables, de calculer *à la main* un ensemble de régions qui convient.

Discussion. Les points suivants nous semblent devoir être particulièrement soulignés car ils contredisent ce qu'il est possible et légitime de penser avant une étude plus précise.
– La nature des contraintes (diagonales ou pas) joue un rôle fondamental, ce qui constitue une différence majeure avec les automates temporisés classiques, dans lesquels les contraintes diagonales n'apportent pas d'expressivité au modèle.
– La décrémentation et l'incrémentation ne jouent pas du tout un rôle semblable.

– De même, les mises à jour $y + c <: z :< y + d$ et $x + c :< z :< y + d$ conduisent à des résultats différents.

3.4 Expressivité

Maintenant que nous avons terminé l'étude de la décidabilité de notre modèle, il apparaît assez naturel et intéressant d'étudier l'expressivité des sous-classes décidables que nous avons dégagées. Nous allons comparer ces sous-classes décidables aux automates temporisés classiques avec et sans ε-transitions.

Afin de pouvoir comparer ces différentes modèles, nous avons besoin d'outils de comparaison. Nous présentons ceux qui nous seront utiles dans la partie qui suit.

3.4.1 Plusieurs équivalences sur les automates temporisés

Équivalence de langages.

Deux automates temporisés sont dits *équivalents* (pour les langages) s'ils acceptent les mêmes langages temporisés. Nous étendons cette définition aux familles d'automates en disant que deux familles d'automates Aut_1 et Aut_2 sont équivalentes si tout automate d'une des classes est équivalent à un automate de l'autre classe. Nous écrivons dans ce cas $Aut_1 \equiv_\ell Aut_2$.
Par exemple, nous avons déjà vu que pour les automates temporisés classiques, les contraintes diagonales peuvent être remplacées par des contraintes non diagonales. Avec le formalisme proposé ici, cela s'écrit simplement

$$Aut(\mathcal{C}_{df}(X), \mathcal{U}_0(X)) \equiv_\ell Aut(\mathcal{C}(X), \mathcal{U}_0(X))$$

où nous rappelons que $\mathcal{U}_0(X)$ est l'ensemble des remises à zéro des horloges.

Systèmes de transitions et similarité.

Nous présentons en toute généralité les systèmes de transitions étiquetés, puis nous définirons la notion de similarité avant de décrire les systèmes de transitions temporisés.

Définition 3.33 *Un* système de transitions *est un 4-uplet $\mathcal{T} = (S, \Gamma, s_0, \longrightarrow)$ où S est un ensemble d'états, Γ est un alphabet, $s_0 \in S$ est l'état initial et $\longrightarrow \subseteq S \times \Gamma \times S$ est un ensemble de transitions.*

La *similarité* [Par81, Mil89] est une notion plus forte que l'inclusion de langages. Elle définit pas à pas une correspondance entre deux systèmes de transitions. Un système de transitions $\mathcal{T} = (S, \Gamma, s_0, \longrightarrow)$ *simule* un système de transitions $\mathcal{T}' = (S', \Gamma, s'_0, \longrightarrow')$ s'il existe une relation $\succcurlyeq \subseteq S \times S'$ telle que :

INITIALISATION : $\forall s_0 \in S_0,\ \exists s'_0 \in S'_0$ t.q. $s_0 \succcurlyeq s'_0$

PROPAGATION : (TRANSFERT) si $s_1 \succcurlyeq s'_1$ et $s_1 \xrightarrow{e} s_2$ alors il existe $s'_2 \in S'$ t.q. $s'_1 \xrightarrow{e} s'_2$ et $s_2 \succcurlyeq s'_2$

Une telle relation $\succcurlyeq$ est appelée une (relation de) *simulation*. Si la relation $\succcurlyeq^{-1}$ définie par

$$x \succcurlyeq^{-1} y \iff y \succcurlyeq x$$

est aussi une simulation, alors nous disons que $\succcurlyeq$ est une (relation de) *bisimulation*.

Systèmes de transitions temporisés.

Les systèmes de transitions temporisés sont alors vus comme des cas particuliers de systèmes de transitions.

Définition 3.34 *Un* système de transitions temporisé *sur l'alphabet* Σ *et le domaine de temps* $\mathbb{T}$ *est un système de transitions* $\mathcal{T} = (S, \Gamma, s_0, \longrightarrow)$ *où* Γ *est l'ensemble* $\Sigma_\varepsilon \cup \{\epsilon(d) \mid d \in \mathbb{T}\}$ *et la relation de transitions* $\longrightarrow$ *vérifie les propriétés suivantes :*

– DÉTERMINISME TEMPOREL : *pour tous les états* s, s', s'' *de* S *et pour tout* $d \in \mathbb{T}$*, si* $s \xrightarrow{\epsilon(d)} s'$ *et* $s \xrightarrow{\epsilon(d)} s''$*, alors* $s' = s''$*.*
– ADDITIVITÉ DU TEMPS : *pour tous les états* s, s'' *de* S *et pour tous* $d_1, d_2 \in \mathbb{T}$*, si* $s \xrightarrow{\epsilon(d_1+d_2)} s''$*, alors il existe* $s' \in S$ *tel que* $s \xrightarrow{\epsilon(d_1)} s'$ *et* $s' \xrightarrow{\epsilon(d_2)} s''$*.*
– ATTENTE DE 0 : *pour tous les états* $s, s' \in S$*,* $s \xrightarrow{\epsilon(0)} s'$ *si et seulement si* $s = s'$*.*

Les trois conditions que nous venons de décrire sont classiques lorsque l'on considère des algèbres de processus comme TCCS [Yi90, Yi91]. Notons que la définition précédente n'impose pas que les systèmes de transitions temporisés aient un nombre fini d'états.

Si $\mathcal{T}$ est un tel système de transitions temporisé, une *exécution* dans $\mathcal{T}$ est une suite de transitions consécutives

$$s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \ldots$$

où pour tout $i > 0$, $s_{i-1} \xrightarrow{\alpha_i} s_i$ est une transition de $\mathcal{T}$. Une *exécution d'attente* est une suite de transitions

$$s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \ldots \xrightarrow{\alpha_n} s_n$$

telle que $n \geq 0$, pour tout $1 \leq i \leq n$, $\alpha_i = \varepsilon$ ou bien $\alpha_i = \epsilon(d_i)$ pour un certain $d_i \in \mathbb{T}$.

***Exemple* 3.35** Considérons le système de transitions temporisé $(\{s_0, s_1\}, \{a, \varepsilon\} \cup \{\epsilon(d) \mid d \in \mathbb{R}^+\}, s_0, \longrightarrow)$ où les transitions sont $s_0 \xrightarrow{\epsilon(0)} s_0$, $s_0 \xrightarrow{a} s_1$, $s_1 \xrightarrow{\epsilon(0)} s_1$, $s_1 \xrightarrow{\epsilon(1)} s_1$ et $s_1 \xrightarrow{\varepsilon} s_0$. Il peut être représenté schématiquement comme sur la figure 3.5. Ce système de transitions permet par exemple de faire l'exécution suivante :

$$s_0 \xrightarrow{\epsilon(0)} s_0 \xrightarrow{a} s_1 \xrightarrow{\epsilon(1)} s_1 \xrightarrow{\epsilon(0)} s_1 \xrightarrow{\epsilon(1)} s_1 \xrightarrow{\varepsilon} s_0$$

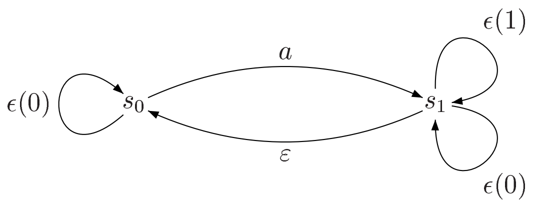


Figure 3.5: Exemple de système de transitions temporisé

Si $\mathcal{T} = (S, \Gamma, s_0, \longrightarrow)$ est un système de transitions temporisé, nous définissons le *système de transitions abstrait* associé à $\mathcal{T}$ par $\mathcal{T}_{abs} = (S, \Gamma, s_0, \Longrightarrow)$ où

$$\begin{cases} s \overset{a}{\Longrightarrow} s' & \iff \quad a \neq \varepsilon \text{ et il existe } s'' \in S,\ s \xrightarrow{\varepsilon}{}^{*} s'' \xrightarrow{a} s' \\ s \overset{\epsilon(d)}{\Longrightarrow} s' & \iff \quad \begin{cases} \text{il existe une exécution d'attente} \\ s = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \ldots \xrightarrow{\alpha_n} s_n = s' \\ \text{telle que } d = \sum\{d_i \mid \alpha_i = \epsilon(d_i)\} \end{cases} \end{cases}$$

où la relation $\xrightarrow{\varepsilon}^*$ représente la clôture réflexive et transitive de $\xrightarrow{\varepsilon}$. Le système de transitions $\mathcal{T}_{abs}$ fait abstraction des actions silencieuses de $\mathcal{T}$. La relation $\xrightarrow{\varepsilon}^*$ correspond donc à $\xRightarrow{\epsilon(0)}$. Remarquons aussi que la relation $\xRightarrow{a}$ abstrait uniquement les actions silencieuses qu'il est possible de faire avant l'action a.

Comme un système de transitions temporisés n'est qu'un cas particulier d'un système de transitions, la notion de similarité définie ci-dessus s'applique.

Similarité forte et faible.

Un automate temporisé avec mises à jour $\mathcal{A} = (Q, X, \Sigma_\varepsilon, I, F, R, T)$, définit deux systèmes de transitions temporisés :

– le système de transitions $\mathcal{T}(\mathcal{A}) = (Q \times \mathbb{T}^X, \Sigma_\varepsilon, \mathbb{T}, (q_0, \mathbf{0}), \longrightarrow)$ où la relation de transitions $\longrightarrow$ est définie par :

$$\begin{cases} (q, v) \xrightarrow{\epsilon(d)} (q, v + d) \\ (q, v) \xrightarrow{a} (q', v') \text{ s'il existe } q \xrightarrow{g,a,up} q' \in T \text{ t.q. } v \models g \text{ et } v' \in up(v) \end{cases}$$

– le système de transitions abstrait $\mathcal{T}_{abs}(\mathcal{A})$ défini comme précédemment à partir de $\mathcal{T}(\mathcal{A})$.

Bien sûr, si $\mathcal{A}$ est un automate temporisé sans ε-transitions, $\mathcal{T}(\mathcal{A})$ et $\mathcal{T}_{abs}(\mathcal{A})$ sont identiques.

Un automate temporisé avec mises à jour $\mathcal{A}$ *simule fortement* un autre automate temporisé avec mises à jour $\mathcal{B}$, ce que nous notons $\mathcal{A} \succcurlyeq_s \mathcal{B}$, si $\mathcal{T}(\mathcal{A})$ simule $\mathcal{T}(\mathcal{B})$. Nous disons que $\mathcal{A}$ et $\mathcal{B}$ sont *fortement bisimilaires* (ce que nous notons $\mathcal{A} \equiv_s \mathcal{B}$) s'il existe une relation de bisimulation $\equiv$ telle que $\mathcal{T}(\mathcal{A}) \equiv \mathcal{T}(\mathcal{B})$.

Un automate temporisé avec mises à jour $\mathcal{A}$ *simule faiblement*[3] un autre automate temporisé avec mises à jour $\mathcal{B}$, ce que nous notons $\mathcal{A} \succcurlyeq_w \mathcal{B}$, si $\mathcal{T}_{abs}(\mathcal{A})$ simule $\mathcal{T}_{abs}(\mathcal{B})$. Nous disons que $\mathcal{A}$ et $\mathcal{B}$ sont *faiblement bisimilaires* (ce que nous notons $\mathcal{A} \equiv_w \mathcal{B}$) s'il existe une relation de bisimulation $\equiv$ telle que $\mathcal{T}_{abs}(\mathcal{A}) \equiv \mathcal{T}_{abs}(\mathcal{B})$.

Remarque : Bien évidemment, deux automates fortement bisimilaires sont aussi faiblement bisimilaires. Si une bisimulation préserve les états finals et répétés, alors deux automates fortement ou faiblement bisimilaires sont équivalents pour les langages.

Soient $\mathcal{A}$ un automate temporisé et λ une constante. Nous notons $\lambda\mathcal{A}$ l'automate temporisé dans lequel toutes les constantes qui apparaissent sont multipliées par λ. La preuve du lemme suivant est immédiate et est semblable à celle du lemme 4.1 page 15 dans [AD94].

Lemme 3.36 *Soient $\mathcal{A}$ et $\mathcal{B}$ deux automates temporisés et $\lambda \in \mathbb{Q}^+$ une constante. Alors*

$$\mathcal{A} \succcurlyeq_w \mathcal{B} \iff \lambda\mathcal{A} \succcurlyeq_w \lambda\mathcal{B} \quad \textit{et} \quad \mathcal{A} \succcurlyeq_s \mathcal{B} \iff \lambda\mathcal{A} \succcurlyeq_s \lambda\mathcal{B}$$

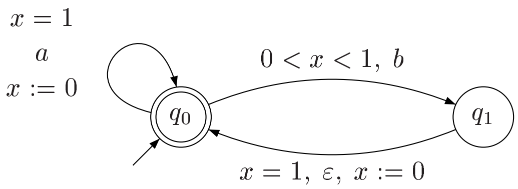
Ce lemme permet, comme dans le cas du test du vide, de nous restreindre aux automates temporisés où toutes les constantes qui apparaissent sont entières.

Maintenant que nous nous sommes donnés des outils de comparaison pour les automates temporisés, nous allons comparer les classes décidables d'automates temporisés avec la classe des automates temporisés classiques. Nous commençons par montrer que les classes décidables d'automates temporisés sont strictement plus expressives que les automates temporisés classiques. Nous montrerons ensuite qu'elles ne sont en fait pas beaucoup plus expressives que les automates temporisés, mais qu'elles permettent une représentation plus **compacte** de certains systèmes.

[3] Notons que cette définition de simulation faible diffère de la définition usuelle car, comme nous l'avons dit au-dessus, la relation de transition $\xRightarrow{a}$ abstrait uniquement les actions silencieuses qui peuvent être effectuées avant les actions, alors que, dans la définition classique, la relation de transition utilisée abstrait toutes les actions silencieuses, c'est-à-dire celles qui peuvent être faites avant ou après une action.

3.4.2 Notre modèle est plus expressif

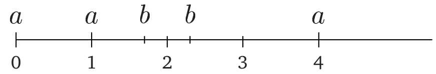
Considérons l'automate temporisé $\mathcal{A}$ avec ε-transitions suivant :



Il est possible de montrer, voir [BDGP98], qu'aucun automate temporisé classique sans ε-transition ne reconnaît le langage temporisé L accepté par $\mathcal{A}$. Ce langage peut être décrit par :

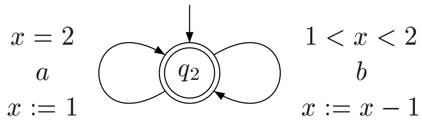
$$(a_i, t_i)_{i\geq 1} \in L \quad \Longleftrightarrow \quad \forall i \geq 1, \begin{cases} t_i = i \text{ et } a_i = a \\ \text{ou} \\ t_i \in]i-1; i[\text{ et } a_i = b \end{cases}$$

Nous pouvons représenter une exécution de l'automate comme sur le schéma qui suit :



Des a peuvent être réalisés à chaque unité de temps sauf si un b a eu lieu dans l'unité de temps qui précède.

Ce langage est reconnu par l'automate temporisé $\mathcal{B}$ avec mises à jour suivant :

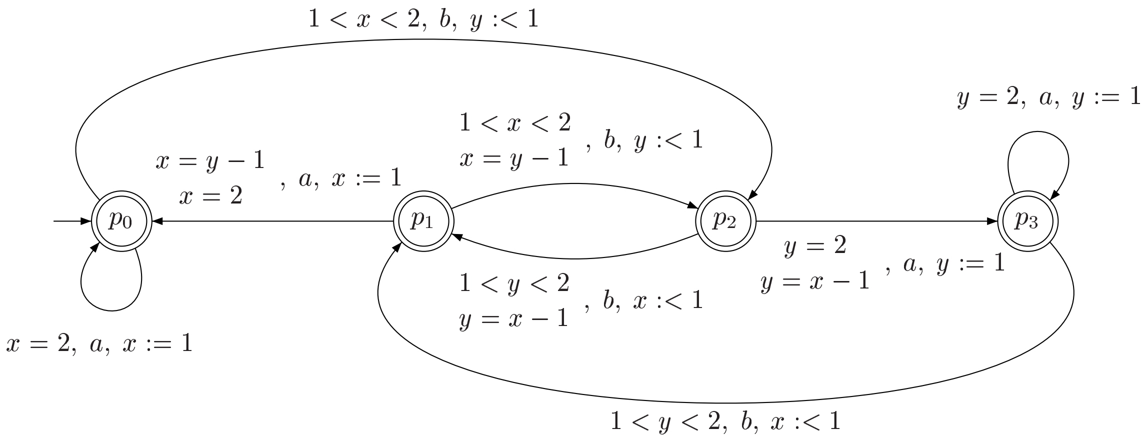


Le résultat est même plus fort que cela : $\mathcal{A}$ et $\mathcal{B}$ sont faiblement bisimilaires. Considérons par exemple la relation

$$\mathcal{R} = \{((q_0, v), (q_2, v+1)) \mid v \in \mathbb{T}^{\{x\}}\} \cup \{((q_1, v), (q_2, v)) \mid v \in \mathbb{T}^{\{x\}}\}$$

Il est très aisé de voir que $\mathcal{R}$ est une bisimulation.

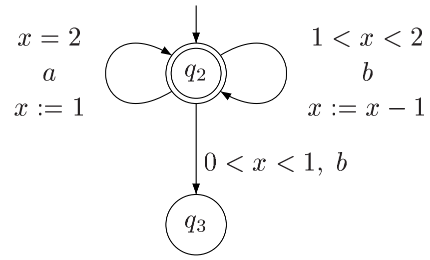
Nous avons vu précédemment que la décrémentation nous entraîne dans des classes indécidables. Cependant, dans ce cas précis, l'horloge x est bornée par 1, donc nous allons pouvoir transformer l'automate $\mathcal{B}$ en un automate appartenant à une classe décidable. Considérons donc l'automate $\mathcal{D}$ suivant :



Nous allons montrer que $\mathcal{D}$ accepte bien le langage L.

PREUVE : Nous commençons par décrire « informellement » ce que fait l'automate $\mathcal{D}$. Nous ne pouvons arriver dans les états p_0 et p_3 qu'après avoir lu un a et nous ne pouvons arriver dans les états p_1 et p_2 qu'après avoir lu un b. Lorsque nous arrivons dans l'un des états p_0 ou p_3, les valeurs de x et de y valent toutes les deux 1 (c'est facile à vérifier en regardant les transitions d'entrée dans ces états). Rester dans ces états permet donc bien de faire des a à un rythme de un toutes les unités de temps. En outre, il est possible de quitter ces états pour aller soit dans l'état p_1, soit dans l'état p_2 en faisant une action b avant la fin de l'unité de temps courante.

Revenons maintenant à notre preuve. Nous transformons l'automate $\mathcal{B}$ de la manière suivante. Nous lui rajoutons un état-puits q_3 avec comme unique transition menant à q_3, la transition $q_2 \xrightarrow{0<x<1,\ b} q_3$. Nous notons $\mathcal{B}_m$ l'automate résultant de cette transformation. Il peut être dessiné comme sur la figure qui suit.



Nous définissons alors la relation $\mathcal{R}$ par :

$$\begin{aligned}\mathcal{R} \quad = \quad & \{(q_2,\alpha),(p_0,(\alpha+1,\alpha+1)) \mid 0 \leq \alpha \leq 1\} \cup \{(q_2,\alpha),(p_3,(\alpha+1,\alpha+1)) \mid 0 \leq \alpha \leq 1\} \\ & \cup \{((q_2,\alpha),(p_2,(\alpha+1,\alpha))) \mid 0 < \alpha \leq 1\} \cup \{((q_2,\alpha),(p_1,(\alpha,\alpha+1))) \mid 0 < \alpha \leq 1\} \\ & \cup \{((q_3,\alpha),(p_2,(\beta,\alpha))) \mid \alpha > 0 \text{ et } \beta \neq \alpha+1\} \cup \{((q_3,\alpha),(p_2,(\alpha,\beta))) \mid \alpha > 0 \text{ et } \beta \neq \alpha+1\}\end{aligned}$$

Les rôles de p_0 et p_3 sont symétriques (il suffit d'inverser x et y). Les rôles de p_1 et p_2 sont symétriques aussi.
La propriété de transfert est vérifiée de manière évidente au niveau de p_0 (donc aussi au niveau de p_3). Elle n'est pas beaucoup plus compliquée au niveau de p_1 et de p_2. La relation $\mathcal{R}$ est donc bien une relation de bisimulation qui rend les automates $\mathcal{D}$ et $\mathcal{B}_m$ bisimilaires. En outre, $\mathcal{B}$ et $\mathcal{B}_m$ reconnaissent bien évidemment le même langage. $\square$

3.4.3 Expressivité des mises à jour déterministes

Les mises à jour simples déterministes ont été définies dans la partie 3.1.1. Nous ne les rappelons pas ici. Si U est un ensemble de mises à jour simples déterministes, nous notons $\ell u(U)$ l'ensemble des mises à jour qui s'écrivent $\bigwedge_{x \in X} up_x$ avec $up_x \in U$ pour tout x.
Nous nous intéressons tout d'abord aux automates temporisés avec ce type de mises à jour. Le résultat suivant est souvent considéré comme un résultat « folklorique » connu de tous. Cependant, il est difficile d'en trouver une preuve dans la littérature. Nous en proposons une ici.

Théorème 3.37 *Soit $\mathcal{C} \subseteq \mathcal{C}(X)$ un ensemble de contraintes d'horloges et soit*

$$\mathcal{U} \subseteq \ell u\left(\{x := d \mid x \in X \text{ et } d \in \mathbb{N}\} \cup \{x := y \mid x,\ y \in X\}\right)$$

Soit $\mathcal{A} \in Aut_\varepsilon(\mathcal{C},\mathcal{U})$. Alors il existe $\mathcal{B}$ dans $Aut_\varepsilon(\mathcal{C}(X),\mathcal{U}_0(X))$ tel que $\mathcal{A} \equiv_s \mathcal{B}$.

PREUVE : Nous décomposons la preuve en plusieurs étapes en commençant par éliminer les mises à jour du type $x := y$, puis nous éliminerons les mises à jour du type $x := d$.

Première étape, éliminer les mises à jour du type $x := y$.

Soit $\mathcal{A} = (Q, X, \Sigma_\varepsilon, I, F, R, T)$ un automate temporisé dans *Aut*$(\mathcal{C}, \mathcal{U})$. Nous construisons un automate temporisé $\mathcal{B} = (Q', X, \Sigma_\varepsilon, I', F', R', T')$ dans *Aut*$(\mathcal{C}(X), \mathcal{U}')$ (où $\mathcal{U}'$ est égal à $\ell u\,(\{x := d \mid x \in X \text{ et } d \in \mathbb{N}\})$) tel que $\mathcal{A} \equiv_s \mathcal{B}$.

Supposons que $X = \{x_1, \ldots, x_n\}$. Nous avons alors :

- $Q' = Q \times X^X$,
- $I' = I \times \{\mathsf{Id}\}$ où Id est la fonction identité de X,
- $F' = F \times X^X$
- $R' = R \times X^X$.

De manière intuitive, dans un état (q, σ) (avec $q \in Q$ et $\sigma \in X^X$), la valeur de l'horloge x est mémorisée par l'horloge $\sigma(x)$. Il nous reste à définir l'ensemble T' des transitions de $\mathcal{B}$.

Nous considérons une transition $q \xrightarrow{g,a,up} q'$ de $\mathcal{A}$ et un état (q, σ) de $\mathcal{B}$. Nous associons à up la fonction $\overline{up}$ définie par $\overline{up}(x)$ vaut d si $x := d$ fait partie de la mise à jour up, vaut y si $x := y$ fait partie de la mise à jour up, vaut x dans tous les autres cas.

Dans $\mathcal{B}$, il y aura une transition

$$(q, \sigma) \xrightarrow{g',a,up'} (q', \sigma')$$

telle que :

- Si $\overline{up}(x) \in X$, alors $\sigma'(x) = \sigma \circ \overline{up}(x)$. Si $\overline{up}(x) \notin X$, cela est un peu plus compliqué. Il y a certaines horloges non utilisées (c'est-à-dire qu'elles ne correspondent pas à un $\sigma'(x)$ déjà défini). Nous piochons alors parmi ces horloges non utilisées pour définir les $\sigma'(x)$ qui nous restent à définir, c'est-à-dire les $\sigma'(x)$ pour lesquels $\overline{up}(x) \notin X$. Plus formellement, nous pouvons écrire que

  $$\#\{x \in X \mid \overline{up}(x) \in X\} \geq \#\{\overline{up}(x) \mid x \in X \text{ et } \overline{up}(x) \in X\}$$

  donc nous pouvons considérer une injection ι de l'ensemble $\{x \in X \mid \overline{up}(x) \notin X\}$ dans l'ensemble $X \setminus \{\overline{up}(x) \mid x \in X \text{ et } \overline{up}(x) \in X\}$ et nous pouvons alors poser $\sigma'(x) = \iota(x)$ si $\overline{up}(x) \notin X$.
- g' est définie par $g[x \leftarrow \sigma(x)]$
- up' est défini par $\bigwedge_{x \in X \text{ et } \overline{up}(x) \notin X} \sigma'(x) := \overline{up}(x)$

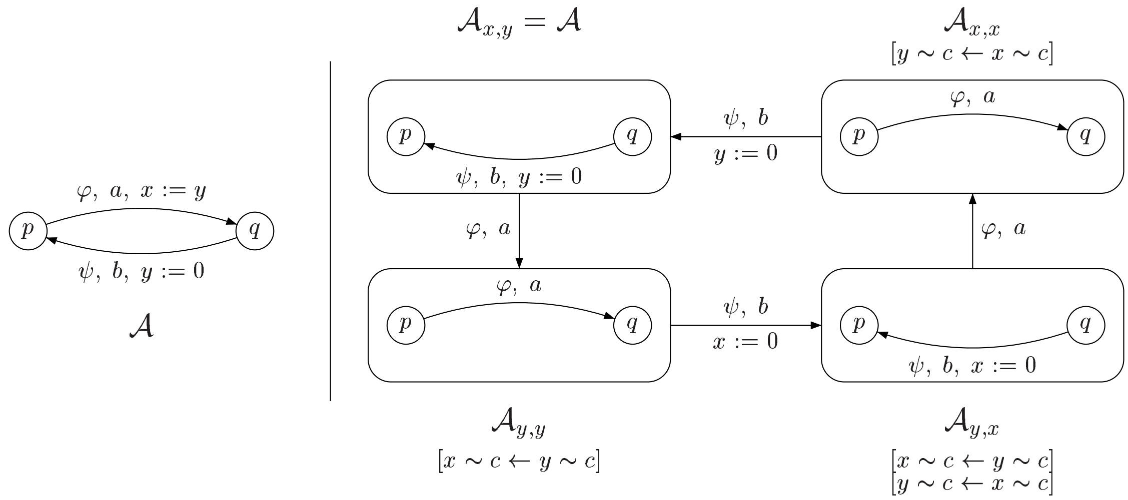
Nous définissons la relation $\mathcal{R}$ sur $(Q \times \mathbb{T}^X) \times ((Q \times X^X) \times \mathbb{T}^X)$ par

$$\{((q, \widehat{v}), ((q, \sigma), v)) \mid q \in Q,\ \sigma \in Y^X,\ v \in \mathbb{T}^Y,\ \widehat{v} \in \mathbb{T}^X \text{ et } \widehat{v} = v \circ \sigma\}$$

La construction a été faite pour que $\mathcal{R}$ soit bien une bisimulation.

Nous allons illustrer la construction précédente par un exemple de transformation d'un automate en éliminant les mises à jour de la forme $x := y$.

***Exemple* 3.38** Considérons par exemple l'automate sur la figure de gauche ci-dessous :



La construction décrite dans la preuve précédente pour l'automate $\mathcal{A}$ est dessinée sur la figure de droite. Ces deux automates sont fortement bisimilaires.

Deuxième étape : éliminer les mises à jour du type $x := d$.
Soit $\mathcal{A}$ un automate temporisé de *Aut*$(\mathcal{C},\mathcal{U})$ où $\mathcal{U} \subseteq \ell u(U')$ avec $U' = \{x := d \mid d \in \mathbb{N}\}$. Nous construisons un automate $\mathcal{B}$ dans *Aut*$(\mathcal{C},\mathcal{U}_0(X))$ fortement bisimilaire à $\mathcal{A}$.
Pour tout x, nous notons c_x la plus grande constante c apparaissant dans $\mathcal{A}$ dans une mise à jour $x := c$. Pour chaque uplet $\alpha = (\alpha_x)_{x\in X}$ tel que pour tout x, $\alpha_x \leq c_x$, nous construisons une copie de l'automate $\mathcal{A}$ que nous notons $\mathcal{A}_\alpha$. Intuitivement, dans l'automate $\mathcal{A}_\alpha$, la valeur de l'horloge x vaudra ce qu'elle devrait valoir dans $\mathcal{A}$ moins la quantité α_x (α apparaît comme un décalage des horloges par rapport à ce qu'elles devraient valoir dans l'automate initial).
Si $q \xrightarrow{g,a,up} q'$ est une transition de $\mathcal{A}$, alors pour chaque α, il y aura une transition $q_\alpha \xrightarrow{g_\alpha,a,up_\alpha} q'_{\alpha'}$ où :

- $g_\alpha = g[x \leftarrow x + \alpha_x]$,
- $up_\alpha = up[x := 0$ à la place de $x := c]$,
- $\alpha'_x = c$ si $x := c$ mise à jour de up, $\alpha'_x = \alpha_x$ sinon.

Il y a un nombre fini d'uplets $\alpha = (\alpha_x)_{x\in X}$ tels que pour tout x, $\alpha_x \leq c_x$. Donc nous ne construisons qu'un nombre fini de copies de l'automate initial. Nous notons $\mathcal{B}$ l'union de tous ces automates $\mathcal{A}_\alpha$. De manière évident, $\mathcal{B}$ est dans *Aut*$(\mathcal{C},\mathcal{U}_0(X))$.

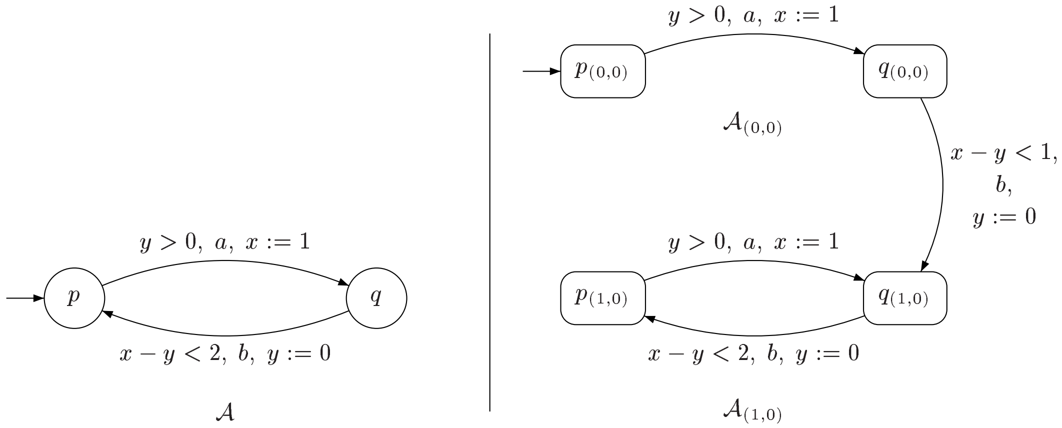
Nous définissons la relation $\mathcal{R}$ entre les états du système de transitions associé à $\mathcal{A}$ et ceux du système de transitions associé à $\mathcal{B}$ comme suit :

$$(q, v)\mathcal{R}(q_\alpha, v_\alpha) \iff v = v_\alpha + \alpha$$

La relation $\mathcal{R}$ est trivialement une bisimulation. Ceci conclut la preuve. $\square$

Nous allons donner un exemple de transformation pour la deuxième étape de la preuve.

***Exemple* 3.39** Considérons l'automate $\mathcal{A}$ dessiné ci-dessous, à gauche. La construction précédente donne l'automate dessiné ci-dessous, à droite (ici, nous n'avons besoin que de deux copies de l'automate car la constante maximale pour x est 1 alors que celle pour y est 0).



Nous allons maintenant considérer les mises à jour déterministes sans restriction et généraliser le résultat précédent à ces mises à jour. Contrairement à ce qui précède et à ce que les résultats folkloriques peuvent laisser penser, nous allons devoir nous restreindre à des contraintes non diagonales. Nous verrons pourquoi après. Notons que le résultat suivant n'est plus ce que nous pourrions appeler un « résultat folklorique ». D'ailleurs, sa preuve est plus compliquée que les preuves précédentes.

Théorème 3.40 *Soient $\mathcal{C} \subseteq \mathcal{C}_{df}(X)$ un ensemble de contraintes non diagonales et*

$$\mathcal{U} \subseteq \ell u\left(\{x := d \mid x \in X \text{ et } d \in \mathbb{N}\} \cup \{x := y + d \mid x,\ y \in X \text{ et } d \in \mathbb{Z}\}\right)$$

un ensemble de mises à jour déterministes tels que le système ($\mathcal{S}_{df}$) associé à $\mathcal{C}$ et à $\mathcal{U}$ ait une solution. Soit $\mathcal{A} \in Aut(\mathcal{C},\mathcal{U})$. Alors il existe $\mathcal{B} \in Aut(\mathcal{C}_{df}(X),\mathcal{U}_0(X))$ tel que $\mathcal{A} \equiv_s \mathcal{B}$.

PREUVE : Soit $\mathcal{A}$ un automate temporisé de $Aut(\mathcal{C},\mathcal{U})$. Nous construisons un automate temporisé $\mathcal{B}$ dans $Aut(\mathcal{C}(X),\mathcal{U}')$ où $\mathcal{U}' \subseteq \ell u\left(\{x := d \mid x \in X \text{ et } d \in \mathbb{N}\} \cup \{x := y \mid x,\ y \in X\}\right)$ qui sera fortement bisimilaire à $\mathcal{A}$. Il suffira alors d'appliquer le théorème 3.37 pour conclure la preuve.

Nous considérons les constantes $(\max_x)_{x \in X}$ solutions du système ($\mathcal{S}_{df}$) de la page 61 pour l'automate $\mathcal{A}$. Pour chaque $\alpha = (\alpha_x)_{x \in X} \in \mathbb{Z}^X$ tel que pour tout x, $\alpha_x \leq \max_x + 1$, pour chaque état q de $\mathcal{A}$, nous construisons une copie q_α de q. Intuitivement, dans l'état q_α, la valeur de l'horloge x vaudra ce qu'elle devrait valoir dans l'état q moins la quantité α_x (α apparaît comme un décalage des horloges par rapport à ce qu'elles devraient valoir dans l'automate initial).

Si $q \xrightarrow{g,a,up} q'$ est une transition de $\mathcal{A}$, alors, nous créons une transition $q_\alpha \xrightarrow{g_\alpha,a,up_\alpha} q'_{\alpha'}$, pour chaque α, avec :

- $g_\alpha = g[x \leftarrow x + \alpha_x]$,
- $up_\alpha = up[x := y$ à la place de $x := y + c]$,
- $\alpha'_x = \begin{cases} \alpha_y + c & \text{si } x := y + c \text{ mise à jour de } up \\ 0 & \text{si } x := c \text{ mise à jour de } up \end{cases}$

  Si la valeur de α'_x ainsi calculée vérifie $\alpha'_x > \max_x$, alors réaffecter $\max_x + 1$ à α'_x. Nous disons que α' est obtenu à partir de α **de manière élémentaire** *via* la mise à jour up.

Le nombre d'uplets $\alpha = (\alpha_x)_{x \in X} \in \mathbb{Z}^X$ tels que pour tout x, $\alpha_x \leq \max_x + 1$ est infini. Nous avons donc construit pour chaque état q, un nombre infini de copies. Cependant, nous montrons que, à partir des états initiaux indicés par $(0, \ldots, 0)$, seulement un nombre fini de tels états est accessible.

Il suffit de montrer que l'ensemble des α tels qu'un état q_α est accessible est minoré.

Supposons que ce ne soit pas le cas. Alors il existe une suite d'uplets $(\alpha^{(i)})_{i\geq 0}$ telle que $\alpha^{(0)} = (0,\ldots,0)$, pour tout i, $\alpha^{(i+1)}$ soit obtenue à partir de $\alpha^{(i)}$ de manière élémentaire à partir d'une mise à jour up_i et la suite $(\alpha_x^{(i)})_{i\geq 0}$ tende vers $-\infty$ (pour un x donné). Par définition de $\mathcal{U}$, chaque up_i peut s'écrire sous la forme :

$$\bigwedge_{x\in X_1} x := d_x \wedge \bigwedge_{\substack{x\in X_2\\ c_x<0}} x := y_x + c_x \wedge \bigwedge_{\substack{x\in X_3\\ c_x\geq 0}} x := y_x + c_x$$

avec X_1, X_2 et X_3 des ensembles disjoints. Nous posons alors

$$up'_i := \bigwedge_{\substack{x\in X_2\\ c_x<0}} x := y_x + c_x$$

et nous définissons la suite $(\beta^{(i)})_{i\geq 0}$ par :

$$\begin{cases} \beta_0 = \alpha_0 \\ \beta^{(i+1)} \text{ est obtenu de manière élémentaire à partir de } \beta^{(i)} \text{ en utilisant } up'_i \end{cases}$$

Il est alors facile de vérifier que la suite $(\beta^{(i)})_{i\geq 0}$ est décroissante non stationnaire (pour l'ordre naturel sur les uplets) car $(\alpha_x^{(i)})_{i\geq 0}$ tend vers $-\infty$ pour un x donné. Notons alors $\mathcal{U} = \bigcup_{i\geq 0} \mathcal{U}_i$.

Soit z_1 une horloge telle que la suite $(\beta_{z_1}^{(i)})_{i\geq 0}$ tende vers $-\infty$. Il existe au moins une mise à jour de la forme $z_1 := z_2 + c_1$ dans $\mathcal{U}$ (donc avec $c_1 < 0$) telle que la suite $(\beta_{z_2}^{(i)})_{i\geq 0}$ tende aussi vers $-\infty$. De cette manière, nous pouvons construire une suite d'horloges $(z_p)_{p\geq 1}$ telle que :

- il y a une mise à jour $z_p := z_{p+1} + c_p$ dans $\mathcal{U}$ (avec $c_p < 0$),
- pour tout $p \geq 1$, la suite $(\beta_{z_p}^{(i)})_{i\geq 0}$ tend vers $-\infty$.

L'ensemble des horloges étant fini, il existe $p < q$ tels que $z_p = z_q$. Cependant, les constantes $(\max_x)_{x\in X}$ sont solutions du système $(\mathcal{S}_{df})$ de la page 61 et ce système comprend en particulier les inéquations

$$\begin{array}{cl} \max_{z_p} \leq \max_{z_{p+1}} + c_p & \text{avec } c_p < 0 \\ \vdots & \\ \max_{z_{q-1}} \leq \max_{z_q} + c_{q-1} & \text{avec } c_{q-1} < 0 \end{array}$$

Nous obtenons donc en particulier que la constante $\max_{z_p} = \max_{z_q}$ doit vérifier $\max_{z_p} < \max_{z_p}$, ce qui est bien entendu impossible.

Nous en déduisons donc que l'ensemble des états q_α accessibles est fini. Nous notons alors $\mathcal{B}$ l'automate que nous venons de construire. Cet automate appartient à $Aut(\mathcal{C}(X), \mathcal{U}')$.

Nous définissons la relation $\mathcal{R}$ comme suit entre les états du système de transitions associé à $\mathcal{A}$ et ceux du système de transitions associé à $\mathcal{B}$:

$$(q,v)\mathcal{R}(q_\alpha, v_\alpha) \iff \begin{cases} v \text{ et } v_\alpha + \alpha \text{ sont équivalentes pour l'équivalence des régions } \mathcal{R}_{(\max_x)_{x\in X}} \\ v(x) \leq \max_x \implies v(x) = v_\alpha(x) + \alpha_x \text{ pour tout } x \in X \end{cases}$$

Nous allons montrer que $\mathcal{R}$ est une bisimulation.

Supposons que $(q,v)\mathcal{R}(q_\alpha, v_\alpha)$ et que $(q,v) \xrightarrow{a} (q', v')$. Cela signifie qu'il existe une transition $q \xrightarrow{g,a,up} q'$ dans $\mathcal{A}$ telle que $v \in g$ et $v' = up(v)$. Dans l'automate que nous avons construit, il y a une transition $q_\alpha \xrightarrow{g_\alpha, a, up_\alpha} q'_{\alpha'}$. Posons $v'_{\alpha'} = up_\alpha(v_\alpha)$ et montrons que $(q', v')\mathcal{R}(q'_{\alpha'}, v')$.

- si x est telle que $x := c$ fait partie de up, alors $x := c$ fait aussi partie de up_α, donc $v'_{\alpha'}(x) = c = v'(x)$ et $\alpha'_x = 0$.
- si x est telle que $x := y + c$ fait partie de up, alors $x := y$ fait partie de up_α,
  - Supposons que $v'(x) \in I_x$ avec $I_x \leq \max_x$ (c'est-à-dire que $I_x =]d-1; d[$ ou $[d]$ avec $d \leq \max_x$). Nous voulons montrer que $v'(x) = v'_{\alpha'}(x) + \alpha'_x$. Pour cela, calculons

$$v'_{\alpha'}(x) + \alpha'_x = v_\alpha(y) + \alpha'_x \text{ car } x := y \text{ fait partie de } up_\alpha$$

  Nous distinguons deux cas :

  1. Si $\alpha'_x \leq \max_x$, alors nous obtenons que

$$v'_{\alpha'}(x) + \alpha'_x = v_\alpha(y) + \alpha_y + c$$

     Or, $(q, v)\mathcal{R}(q_\alpha, v_\alpha)$ et $v(y) \leq \max_y$ (car $v'(x) = v(y) + c \leq \max_x$ et $\max_x \leq \max_y + c$), donc

$$v'_{\alpha'}(x) + \alpha'_x = v(y) + c = v'(x)$$

  2. Si $\alpha'_x > \max_x$, alors cela signifie que $\alpha_y + c > \max_x$. Or,

$$v'(x) = v(y) + c = v_\alpha(y) + \alpha_y + c > \max_x$$

     Ceci n'est bien entendu pas possible car nous avons supposé que $v'(x) \leq \max_x$.

  - Supposons que $v'(x) > \max_x$. Nous distinguons deux cas :
    1. si $\alpha'_x > \max_x$, alors $v'_{\alpha'}(x) + \alpha'_x > \max_x$
    2. si $\alpha'_x \leq \max_x$, alors $v'_{\alpha'}(x) + \alpha'_x = v_\alpha(y) + \alpha_y + c$. Nous distinguons encore une fois deux cas :

       (i) si $v_\alpha(y) + \alpha_y \leq \max_y$, alors

$$v'_{\alpha'}(x) + \alpha'_x = v(y) + c = v'(x) > \max_x$$

       (ii) si $v_\alpha(y) + \alpha_y > \max_y$, alors comme $\max_x \leq \max_y + c$, nous obtenons que $v'_{\alpha'}(x) + \alpha'_x > \max_x$.

    Dans tous les cas, nous avons que $v'_{\alpha'}(x) + \alpha'_x > \max_x$, ce que nous voulions.
  - le changement entre up et up_α conserve l'ordre relatifs des parties fractionnaires

Tout cela permet de montrer que $(q', v')\mathcal{R}(q'_{\alpha'}, v'_{\alpha'})$. La réciproque est identique.

Nous avons donc bien exhibé une relation de bisimulation entre $\mathcal{A}$ et $\mathcal{B}$. □

Remarque : Considérant les automates du théorème 3.40 dans lesquels nous autorisons aussi les contraintes diagonales, nous avons vu dans la partie 3.2 que nous tombions alors dans une classe indécidable. Il est intéressant de se poser la question de savoir à quel endroit de la preuve précédente, l'hypothèse sur les contraintes non diagonales est fondamentale. Pour pouvoir n'avoir qu'un nombre fini de copies de chaque état, nous réaffectons la valeur $\max_x + 1$ à α'_x lorsque la valeur calculée est plus grande que $\max_x + 1$. Cette réaffectation ne change en rien la véracité des contraintes non-diagonales, mais peut changer celle de certaines contraintes diagonales.

***Exemple* 3.41** Là encore, nous considérons un exemple simple. Les deux automates de la figure 3.6 sont fortement bisimilaires, celui de droite résulte de la construction décrite dans la preuve précédente en prenant comme automate de départ celui de gauche et les constantes maximales $\max_x = 0$ et $\max_y = 1$.

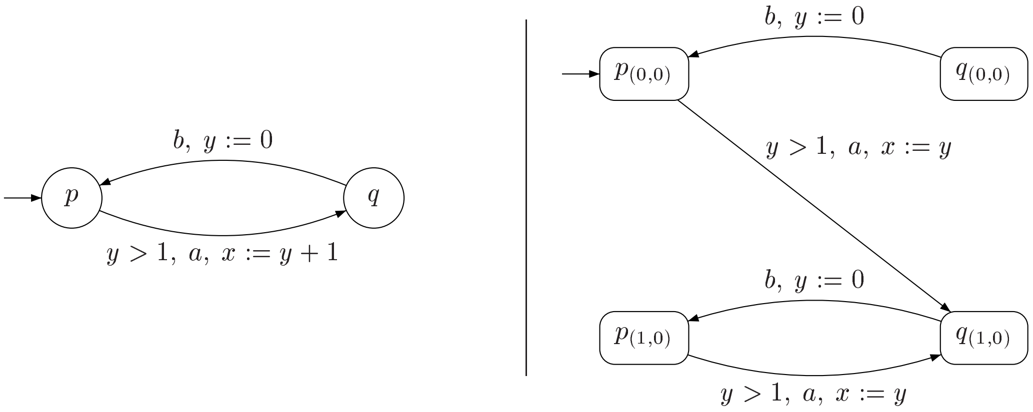


Figure 3.6: Deux automates fortement bisimilaires

3.4.4 Expressivité des mises à jour non déterministes

Pour le cas des mises à jour non déterministes, nous commençons par rappeler le langage considéré dans la partie 3.4.2. Ce langage n'est reconnu par aucun automate temporisé classique. Il est donc vain d'espérer obtenir une bisimulation forte entre les automates temporisés avec mises à jour non déterministes et les automates temporisés classiques.

Nous nous focalisons sur la similarité faible. Comme il va apparaître, les constructions que nous faisons ne sont pas très simples et assez techniques. Nous commençons par traiter le cas des automates temporisés avec contraintes non diagonales. Nous étudierons le cas général plus tard.

Construction pour des contraintes non diagonales.

Nous proposons une forme normale pour les automates temporisés avec contraintes non diagonales. Soient $(\max_x)_{x\in X}$ des constantes entières. Dans ce qui suit, nous nous restreignons aux contraintes $x \sim c$ telles que $c \leq \max_x$. Comme dans la partie 3.3.2, nous posons :

$$\mathcal{I}_x = \{[c] \mid 0 \leq c \leq \max_x\} \cup \{]c; c+1[\mid 0 \leq c < \max_x\} \cup \{]\max_x, \infty[\}$$

Une contrainte d'horloges g est dite *totale* si φ est une conjonction $\bigwedge_{x\in X}(x \in I_x)$ où pour toute horloge x, I_x est dans $\mathcal{I}_x$. Chaque contrainte bornée par les constantes $(c_x)_{x\in X}$ est équivalente à une disjonction de contraintes totales.

Nous définissons aussi

$$\mathcal{I}'_x = \{]c; c+1[\mid 0 \leq c < \max_x\} \cup \{]\max_x, \infty[\}$$

Une mise à jour up_x pour l'horloge x est dite *élémentaire* si elle est de l'une des formes suivantes :
- $x :\in I_x$ avec $I_x \in \mathcal{I}_x$,
- $x := y + c \wedge x :\in I'_x$ avec $I'_x \in \mathcal{I}'_x$ et $\max_x \leq \max_y + c$,
- $\left(\bigwedge_{y\in H} x :< y + c \wedge x :\in I'_x\right)$ avec $H \subseteq X$, $I'_x \in \mathcal{I}'_x$ et $\forall y \in H$, $\max_x \leq \max_y + c$,
- $\left(\bigwedge_{y\in H} x :> y + c \wedge x :\in I'_x\right)$ avec $H \subseteq X$, $I'_x \in \mathcal{I}'_x$ et $\forall y \in H$, $\max_x \leq \max_y + c$.

Une mise à jour élémentaire up_x est *compatible* avec une contrainte totale $\bigwedge_{x\in X}(x \in I_x)$ si :
- $I_y + c \subseteq I'_x$ si up_x est $x := y + c \wedge x :\in I'_x$,
- pour tout $y \in H$, $I_y + c \subseteq I'_x$ si up_x est $((\bigwedge_{y\in H} x :\sim y + c) \wedge x :\in I'_x)$ et $I'_x = I_x$.

Définition 3.42 *Soient* $(\max_x)_{x \in X}$ *des constantes entières et* $\mathcal{A}$ *un automate temporisé de la classe* *Aut*$(\mathcal{C}_{df}(X), \mathcal{U}(X))$. *Nous disons que* $\mathcal{A}$ *est sous* forme normale *pour les constantes* $(\max_x)_{x \in X}$ *si pour chaque transition* $q \xrightarrow{g,a,up} q'$ *de* $\mathcal{A}$*, nous avons :*
– g *est une contrainte totale,*
– $up = \bigwedge_{x \in X} up_x$ *avec pour tout horloge* x*,* up_x *est une mise à jour élémentaire compatible avec* g.

En appliquant les règles classiques du calcul propositionnel et en découpant les transitions, nous obtenons la forme normale suivante pour les automates temporisés avec contraintes non diagonales (rappelons que nous nous restreignons aux mises à jour définies par ($\diamondsuit_{df}$), page 60).

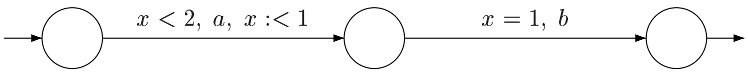
Proposition 3.43 *Soient* $\mathcal{C}$ *un ensemble de contraintes non diagonales et* $\mathcal{U}$ *un ensemble de mises à jour engendré par la grammaire (*$\diamondsuit_{df}$*). Nous supposons que le système (*$\mathcal{S}_{df}$*) admet comme solution* $(\max_x)_{x \in X}$. *Alors tout automate temporisé de* Aut$(\mathcal{C}, \mathcal{U})$ *est fortement bisimilaire à un automate de* *Aut*$(\mathcal{C}_{df}(X), \mathcal{U}(X))$ *sous forme normale pour les constantes* $(\max_x)_{x \in X}$.

Nous pouvons maintenant établir notre principal résultat concernant l'expressivité des automates temporisés avec contraintes non diagonales :

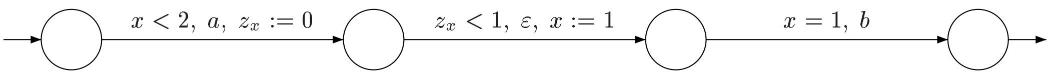
Théorème 3.44 *Soient* $\mathcal{C}$ *un ensemble de contraintes non diagonales et* $\mathcal{U}$ *un ensemble de mises à jour engendré par la grammaire (*$\diamondsuit_{df}$*). Nous supposons que le système (*$\mathcal{S}_{df}$*) a une solution. Soit* $\mathcal{A} \in$ *Aut*$(\mathcal{C}, \mathcal{U})$. *Il existe un automate* $\mathcal{B} \in$ *Aut*$_\varepsilon(\mathcal{C}_{df}(X), \mathcal{U}_0(X))$ *tel que* $\mathcal{B} \succcurlyeq_w \mathcal{A}$ *et* $\mathcal{A} \equiv_\ell \mathcal{B}$.

Avant d'écrire la preuve de ce théorème, nous considérons deux petits exemples pour permettre de mieux comprendre la construction qui sera faite dans la preuve.

***Exemple* 3.45** Considérons l'automate suivant :

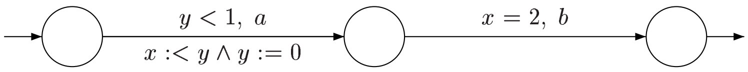


Le langage reconnu par cet automate est $\{(a,t)(b,t') \mid 0 \leq t < 2 \text{ et } 0 < t' - t < 1\}$.
L'automate précédent peut être faiblement simulé par l'automate, qui ne contient plus que des mises à jour déterministes, suivant :

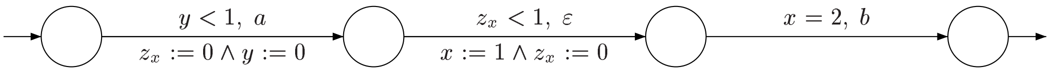


La mise à jour non déterministe du premier automate a été remplacée ici par une ε-transition. L'horloge z_x qui a été rajoutée représente la partie fractionnaire de x et veille donc à ce que celle-ci ne dépasse pas 1.

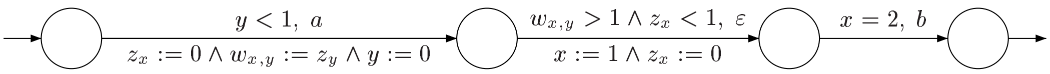
***Exemple* 3.46** Considérons maintenant l'automate suivant :



Le langage reconnu par cet automate est $\{(a,t)(b,t') \mid t < 1 \text{ et } t' > 2\}$.
Une première idée de preuve, fausse, serait de faire la même chose que précédemment, à savoir :



Cet automate n'est pas correct, il reconnaît par exemple le mot $(a, 0.5)(b, 1.8)$ alors que ce mot n'est pas accepté par le premier automate.
L'idée est alors de rajouter une nouvelle horloge, $w_{x,y}$ qui aura comme rôle de se souvenir que, lorsque x a été initialisée, elle était plus petite que y. Nous construisons alors l'automate suivant :



Il est facile de vérifier que cet automate reconnaît le même langage que le premier automate.

PREUVE : Grâce au lemme 3.36 et à la proposition 3.43, nous pouvons supposer que toutes les constantes qui apparaissent dans $\mathcal{A}$ sont entières et que $\mathcal{A}$ est sous forme normale pour les constantes $(\max_x)_{x \in X}$.
Une horloge x est dite *fixée* si la dernière mise à jour qui a changé la valeur de x était de la forme $x := c$ ou $(x := y + c \wedge x :\in I'_x)$ où y était une horloge fixée, Une horloge est dite *flottante* si elle n'est pas fixée. La terminologie « flottante » provient du fait qu'une horloge flottante a sa valeur dans un intervalle connu $]d; d+1[$, mais nous ne connaissons pas sa valeur précise.
La transformation que nous allons faire consiste à construire plusieurs copies de l'automate initial $\mathcal{A}$, en ajoutant quelques horloges, en transformant les transitions et enfin en rajoutant quelques ε-transitions entre les différentes copies des automates.

Horloges supplémentaires.
Pour chaque horloge x de X, nous définissons une horloge z_x qui va intuitivement représenter la partie fractionnaire de x.
Pour chaque couple d'horloges (x, y) de X^2, nous définissons également deux horloges $w_{x,y}$ et $w'_{x,y}$ qui vont servir à comparer les parties fractionnaires de x et y.
Soit $\overline{X}$ l'ensemble de ces $2|X|^2$ horloges supplémentaires. Nous verrons leurs rôles précis le long de la construction.

Duplication de l'automate original.
Considérons un sous-ensemble Y de X, correspondant intuitivement aux horloges flottantes et un ordre partiel $\prec$ sur Y, représentant les positions relatives des parties fractionnaires des horloges de Y.
De plus, pour chaque horloge y de Y, nous définissons l'intervalle I_y, de la forme $]d; d+1[$ avec $0 \leq d < \max_y$ auquel la valeur de y va appartenir.
Enfin, nous considérons un sous-ensemble Z de $\overline{X}$, dont le rôle sera expliqué ultérieurement.

Pour chaque triplet $\tau = ((I_y)_{y \in Y}, \prec, Z)$, nous construisons alors une copie $\mathcal{A}_\tau$ de l'automate $\mathcal{A}$, dans laquelle est ajoutée à la contrainte de chaque transition la contrainte

$$\bigwedge_{y \in Y} y \in I_y \wedge \bigwedge_{x \in X} z_x < 1$$

Un certain nombre de contraintes de l'automate ainsi construit peuvent être trivialement équivalentes à « Faux ». Les transitions correspondantes peuvent alors être effacées.
Nous notons T l'ensemble de tous les triplets τ.

Horloges fixées.
Lorsque la partie fractionnaire d'une horloge fixée atteint la valeur 1, le calcul ne change pas de copie d'automate. Pour cela, dans chaque copie $\mathcal{A}_\tau$ avec $\tau = ((I_y)_{y \in Y}, \prec, Z)$, nous ajoutons sur chaque état, et pour chaque horloge $x \in X \setminus Y$, une boucle étiquetée par $z_x = 1,\ \varepsilon,\ z_x := 0$.

Horloges flottantes.

Il est possible de fixer certaines horloges flottantes en utilisant une ε-transition. Bien entendu, cela ne peut se faire que pour les horloges flottantes maximales pour le préordre de la copie considérée (les horloges maximales doivent atteindre la borne supérieure de leur intervalle avant les autres horloges flottantes, c'est tout l'intérêt du préordre). Formellement, soit $\mathcal{A}_\tau$ avec $\tau = ((I_y)_{y\in Y}, \prec, Z)$ et soit M l'ensemble des éléments maximaux de $\prec$. Pour tout état q de $\mathcal{A}_\tau$, nous construisons une ε-transition vers la copie de q de l'automate $\mathcal{A}_{\tau'}$ où $\tau' = ((I_y)_{y\in Y'}, \prec', Z')$ avec

$$\begin{cases} Y' = Y \setminus M \\ \prec' = \prec \cap (Y' \times Y') \\ Z' = Z \setminus \{w_{x,y}, w'_{x,y} \mid x \in M\} \end{cases}$$

Cette ε-transition est étiquetée par la contrainte

$$\bigwedge_{x\in M,\ w_{x,y}\in Z} (w_{x,y} \geq 1) \wedge \bigwedge_{x\in M,\ w'_{x,y}\in Z} (w'_{x,y} < 1) \wedge \bigwedge_{y\in Y} (z_y < 1)$$

et par la mise à jour

$$\bigwedge_{y\in M} y := \mathsf{sup}(I_y)$$

où $\mathsf{sup}(I_y)$ représente la borne supérieure de I_y, c'est-à-dire $d+1$ si $I_y =]d; d+1[$.
L'existence d'une horloge $w_{x,y}$ (respectivement $w'_{x,y}$) montre qu'une mise à jour de la forme $x :< y + c$ (respectivement $x :> y + c$) a été effectuée précédemment. La contrainte $w_{x,y} \geq 1$ (respectivement $w'_{x,y} < 1$) assure que nous avons bien simulé une telle mise à jour.

Modification des transitions.
Nous considérons une copie $\mathcal{A}_\tau$ avec $\tau = ((I_y)_{y\in Y}, \prec, Z)$ et une transition $(q_\tau, g, a, up, q'_\tau)$ de cette copie. Cette transition va être remplacée par une transition $(q_\tau, g, a, \widehat{up}, q'_{\widehat{\tau}})$ où $q'_{\widehat{\tau}}$ est l'état correspondant à q'_τ dans une autre copie $\mathcal{A}_{\widehat{\tau}}$ avec $\widehat{\tau} = (\widehat{I_y})_{y\in\widehat{Y}}, \widehat{\prec}, \widehat{Z})$ qui va être spécifié un peu plus tard.
Les éléments $\widehat{Y}$, $(\widehat{I_y})_{y\in\widehat{Y}}$, $\widehat{\prec}$ et $\widehat{up}$ vont être construits inductivement en considérant les uns après les autres les mises à jour up_x qui interviennent dans up (l'ordre dans lequel ils vont être traités n'importe pas).
La nouvelle mise à jour $\widehat{up}$ sera uniquement constituée de mises à jour de la forme $x := c$ ou $x := y + c$. Initialement, nous posons $\widehat{Y} = Y$, $\widehat{I_y} = I_y$ pour tout $y \in Y$, $\widehat{\prec} = \prec$, $\widehat{up} = \varnothing$ et $\widehat{Z} = Z$.

Avant de lister les différentes mises à jour, expliquons le rôle de l'ensemble Z qui n'a été que brièvement révélé jusqu'à présent. Supposons que l'horloge x soit remise à jour par $x :< y + c$ où y est une horloge fixée. Alors l'horloge x devient flottante. Nous utilisons l'horloge z_x pour stocker la partie fractionnaire de x, nous la mettons à zéro à ce moment. Il nous faut aussi garder en mémoire la valeur courante de la partie fractionnaire de y, stockée pour l'instant dans z_y. Comme z_x doit rester inférieure à z_y, z_y doit atteindre 1 avant z_x. Bien sûr, si l'horloge y n'est pas remise à jour, il va être possible de tester ceci grâce à z_y, mais si y est remise à jour avant que z_y n'atteigne 1, l'ancienne valeur de z_y va être perdue. Nous rajoutons alors l'horloge $w_{x,y}$ à l'ensemble Z et nous posons $w_{x,y} := z_y$. L'horloge $w_{x,y}$ va garder en mémoire l'ancienne valeur de z_y quoi qu'il advienne de y. La propriété qui nous restera à vérifier est donc $w_{x,y} \geq 1$. Le rôle des horloges $w'_{x,y}$ est semblable, sauf qu'elles sont utilisées pour les mises à jour de la forme $x :> y + c$ où y est une horloge fixée. La condition « z_x atteint 1 avant z_y » est testée grâce à $w'_{x,y} < 1$. L'exemple 3.46 illustre l'utilisation de ces horloges.

Comme annoncé, nous listons les différents cas possibles pour les mises à jour :

- si up_x est égal à $x := c$ alors nous devons juste considérer x comme devenant fixée :

$\widehat{Y} \leftarrow \widehat{Y} \setminus \{x\}$, $\widehat{Z} \leftarrow \widehat{Z} \setminus \{w_{x,y}, w'_{x,y} \mid y \in X\}$, $\widehat{up} \leftarrow \widehat{up} \wedge x := c \wedge z_x := 0$

- si up_x est égal à $x :\in I'_x$:
  1. si $I'_x =]c_x; +\infty[$, alors

     $\widehat{Y} \leftarrow \widehat{Y} \setminus \{x\}$, $\widehat{Z} \leftarrow \widehat{Z} \setminus \{w_{x,y}, w'_{x,y} \mid y \in X\}$, $\widehat{up} \leftarrow \widehat{up} \wedge x := c_x + 1 \wedge z_x := 0$
  2. si $I'_x =]c; c+1[$, alors

     $\widehat{Y} \leftarrow \widehat{Y} \cup \{x\}$, $\widehat{Z} \leftarrow \widehat{Z} \setminus \{w_{x,y}, w'_{x,y} \mid y \in X\}$, $\widehat{\precsim}$ devient un préordre total quelconque compatible avec $\widehat{\precsim}$ sur $\widehat{Y} \setminus \{x\}$, $\widehat{up} \leftarrow \widehat{up} \wedge z_x := 0$

– si up_x est égal à $x := y + c \wedge x :\in I'_x$, alors :
  1. si $y \notin Y$,
     – $\widehat{Y} \leftarrow \widehat{Y} \setminus \{x\}$,
     – $\widehat{Z} \leftarrow \widehat{Z} \setminus \{w_{x,y}, w'_{x,y} \mid y \in X\}$
     – $\widehat{up} \leftarrow \widehat{up} \wedge x := y + c \wedge z_x := z_y$
  2. si $y \in Y$,
     – si I'_x est borné,
       · $\widehat{Y} \leftarrow \widehat{Y} \cup \{x\}$,
       · $\widehat{Z} \leftarrow \widehat{Z} \setminus \{w_{x,y}, w'_{x,y} \mid y \in X\}$,
       · $x \widehat{\precsim} y$ et $y \widehat{\precsim} x$,
       · $\widehat{I_x} = I'_x$,
       · $\widehat{up} \leftarrow \widehat{up} \wedge z_x := x_y$
     – si I'_x n'est pas borné, *i.e.* $I'_x =]c_x; +\infty[$,
       · $\widehat{Y} \leftarrow \widehat{Y} \setminus \{x\}$,
       · $\widehat{Z} \leftarrow \widehat{Z} \setminus \{w_{x,y}, w'_{x,y} \mid y \in X\}$,
       · $\widehat{up} \leftarrow \widehat{up} \wedge x := c_x + 1 \wedge z_x := z_y$

- si up_x est $\left(\bigwedge_{y \in H} x :< y + c\right) \wedge x :\in I'_x$, nous posons $H_1 = H \cap Y$ et $H_2 = H \setminus Y$ et
  – si I'_x est borné,
    · $\widehat{Y} = \widehat{Y} \cup \{x\}$,
    · $\widehat{Z} = (\widehat{Z} \setminus \{w_{x,y}, w'_{x,y} \mid y \in X\}) \cup \{w_{x,y} \mid y \in H_2\}$,
    · $x \widehat{\precsim} y$ et $y \widehat{\not\precsim} x$ si $y \in H_1$,
    · $\widehat{up} \leftarrow \widehat{up} \wedge z_x := 0 \wedge \bigwedge_{y \in H_2} w_{x,y} := z_y$.
  – si I'_x est $]c_x; +\infty[$,
    · $\widehat{Y} = \widehat{Y} \setminus \{x\}$,
    · $\widehat{Z} = (\widehat{Z} \setminus \{w_{x,y}, w'_{x,y} \mid y \in X\})$,
    · $\widehat{up} \leftarrow \widehat{up} \wedge x := c_x + 1 \wedge z_x := 0$.
- si up_x est $\left(\bigwedge_{y \in H} x :> y + c\right) \wedge x :\in I'_x$, nous posons $H_1 = H \cap Y$ et $H_2 = H \setminus Y$ et
  – si I'_x est borné,
    · $\widehat{Y} = \widehat{Y} \cup \{x\}$,
    · $\widehat{Z} = (\widehat{Z} \setminus \{w_{x,y}, w'_{x,y} \mid y \in X\}) \cup \{w'_{x,y} \mid y \in H_2\}$,
    · $y \widehat{\precsim} x$ et $x \widehat{\not\precsim} y$ si $y \in H_1$,

- $\widehat{up} \leftarrow \widehat{up} \wedge z_x := 0 \wedge \bigwedge_{y \in H_2} w'_{x,y} := z_y$.

– si I'_x est $]c_x; +\infty[$,

- $\widehat{Y} = \widehat{Y} \setminus \{x\}$,
- $\widehat{Z} = (\widehat{Z} \setminus \{w_{x,y}, w'_{x,y} \mid y \in X\})$,
- $\widehat{up} \leftarrow \widehat{up} \wedge x := c_x + 1 \wedge z_x := 0$.

Il nous reste à montrer que ce gros automate que nous venons de construire simule faiblement l'automate initial et que, en outre, il reconnaît le même langage. En plus, nous utiliserons le théorème 3.37 pour conclure. Il faudra donc vérifier que ses hypothèses sont bien vérifiées dans notre construction.

Nous allons maintenant proposer une relation $\mathcal{R}$ qui est une simulation. Grossièrement, un état de l'automate original va être en relation avec toutes les copies de cet état dans les différentes copies de cet automate. L'ensemble des états du système de transitions de $\mathcal{A}$ est $Q \times \mathbb{T}^X$ alors que celui de $\mathcal{B}$ est

$$\{q_\tau \mid q \in Q \text{ et } \tau \in T\} \times \mathbb{T}^{X \cup \{z_x \mid x \in X\} \cup Z}$$

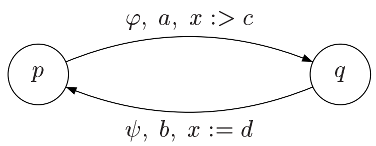
Nous définissons la relation $\succcurlyeq$ par

$$\succcurlyeq \; = \left\{ ((q_\tau, v'), (q, v)) \;\middle|\; \begin{array}{l} \forall y \in Y,\ v(y) \in I_y \text{ et } 0 \leq v'(z_y) \leq 1, \\ \forall y \in X \setminus Y,\ \text{soit } v(y) = v'(y) \text{ soit } (v(y) > c_y \text{ et } v'(y) > c_y), \\ y_1 \prec y_2 \implies \mathsf{frac}(v(y_1)) \leq \mathsf{frac}(v(y_2)), \\ w_{x,y} \in Z \implies \mathsf{frac}(v(x)) < v'(w_{x,y}) \\ \quad \text{et } w'_{x,y} \in Z \implies \mathsf{frac}(v(x)) > v'(w'_{x,y}). \end{array} \right\}$$

Il est assez facile mais fastidieux de montrer que $\succcurlyeq$ est une simulation et que l'automate construit reconnaît le même langage que l'automate initial.

L'automate construit n'a que des mises à jour déterministes et des contraintes non diagonales, nous pouvons utiliser le théorème 3.37. Ceci termine donc la preuve du théorème 3.44. □

***Exemple* 3.47** Considérons l'automate temporisé très simple suivant :



La transformation que nous venons de présenter dans la preuve construit l'automate dessiné sur la figure 3.7 qui suit (pour simplifier, nous ne dessinons pas les horloges $w_{x,y}$ et $w'_{x,y}$ car aucune n'est nécessaire ici).
La construction que nous venons de présenter souffre d'une explosion combinatoire assez importante, nous ne dessinons donc là qu'une petite partie de l'automate résultant, celle-ci étant suffisante pour la compréhension de la construction.

Construction pour des contraintes générales.

Comme dans ce qui précède, nous définissons une forme normale pour les automates temporisés avec contraintes générales. Nous reprenons les définitions de $\mathcal{I}_x$, $\mathcal{I}'_x$, $\mathcal{J}_{x,y}$ précédemment vues. Nous dirons qu'une contrainte générale

$$\bigwedge_{x \in X} x \in I_x \wedge \bigwedge_{x,y \in X} x - y \in J_{x,y}$$

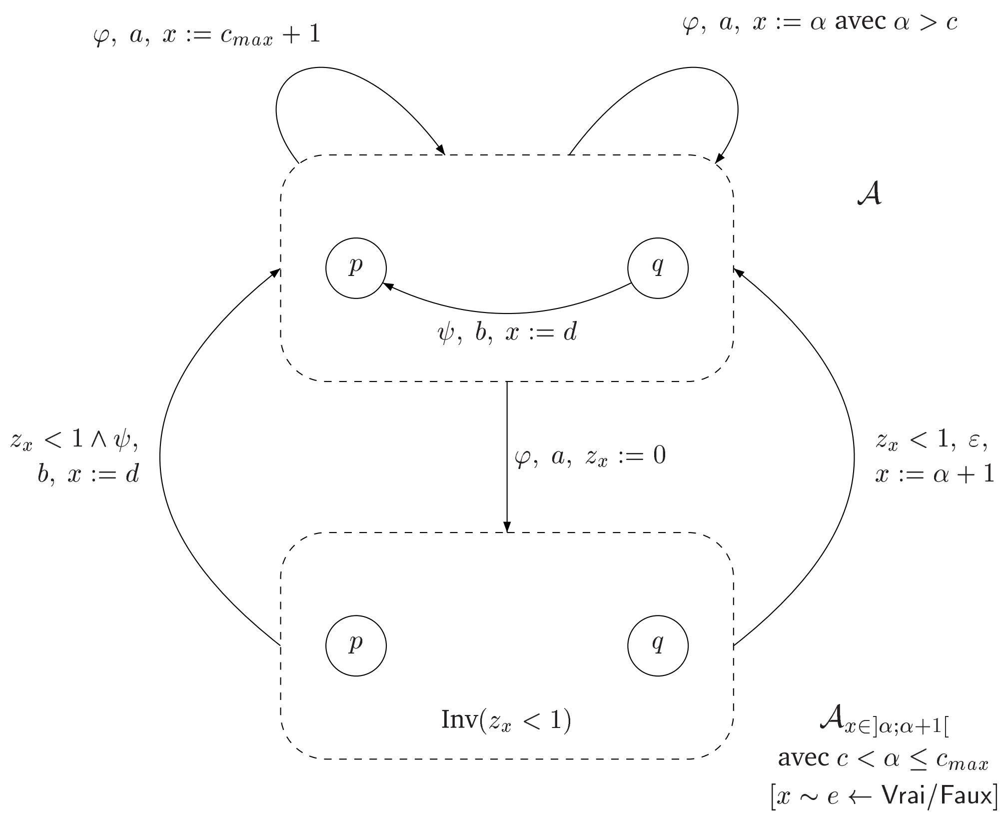


Figure 3.7: Élimination des mises à jour non déterministes

est *totale* si pour tout x, $I_x \in \mathcal{I}_x$ et pour toutes les horloges $x,y \in X$, $J_{x,y} \in \mathcal{J}_{x,y}$. Nous dirons aussi qu'une mise à jour pour l'horloge x, up_x, est *strictement élémentaire* si elle est de l'une des formes suivantes :

- $x := c$ avec $0 \leq c \leq c_x$,
- $x :\in I''_x$ avec $I''_x \in \mathcal{I}''_x$ ($\mathcal{I}''_x$ désigne l'ensemble $\{]c;c+1[\mid 0 \leq c < \max_x\}$),
- $(x := y \wedge x :\in I'_x)$ avec $I'_x \in \mathcal{I}'_x$.

Une mise à jour strictement élémentaire up_x est *compatible* avec une contrainte totale

$$\bigwedge_{x\in X} x \in I_x \wedge \bigwedge_{x,y\in X} x - y \in J_{x,y}$$

si $I_y \subseteq I'_x$ dès que up_x est $x := y \wedge x :\in I'_x$.

Définition 3.48 *Soient* $((\max_x)_{x\in X}, (\max_{x,y})_{x,y\in X})$ *des constantes et* $\mathcal{A}$ *un automate temporisé de Aut*$(\mathcal{C}(X),\mathcal{U}(X))$. *Il est dit en forme normale pour les constantes* $((\max_x)_{x\in X}, (\max_{x,y})_{x,y\in X})$ *si pour toute transition* $\xrightarrow{g,a,up} q'$ *de* $\mathcal{A}$*, nous avons :*

- *g est une contrainte totale,*
- $up = \bigwedge_{x\in X} up_x$ *avec pour tout horloge* x, up_x *est une mise à jour élémentaire compatible avec* g.

En appliquant les règles classiques du calcul propositionnel et en découpant les transitions, nous obtenons la forme normale suivante pour les automates temporisés avec contraintes générales (rappelons que nous nous restreignons aux mises à jour définies par ($\diamondsuit_{gen}$) de la page 66).

Proposition 3.49 *Soient $\mathcal{C}$ un ensemble de contraintes générales et $\mathcal{U}$ un ensemble de mises à jour engendré par la grammaire ($\diamondsuit_{gen}$). Nous supposons que le système ($\mathcal{S}_{gen}$) admet comme solution* $((\max_x)_{x\in X}, (\max_{x,y})_{x,y\in X})$. *Alors tout automate de Aut($\mathcal{C}$,$\mathcal{U}$) est fortement bisimilaire à un automate de Aut($\mathcal{C}(X)$,$\mathcal{U}(X)$) sous forme normale pour les constantes* $((\max_x)_{x\in X}, (\max_{x,y})_{x,y\in X})$.

Lorsque nous nous intéressons aux sous-classes décidables des automates temporisés utilisant des contraintes générales, nous devons fortement restreindre les mises à jour que nous utilisons. Comme l'établit le théorème suivant, ces classes sont encore faiblement bisimilaires à des automates temporisés classiques qui utilisent des ε-transitions.

Théorème 3.50 *Soient $\mathcal{C}$ un ensemble de contraintes générales et $\mathcal{U}$ un ensemble de mises à jour engendré par la grammaire ($\diamondsuit_{gen}$). Soit $\mathcal{A}$ un automate de Aut($\mathcal{C}$,$\mathcal{U}$). Alors il existe un automate $\mathcal{B}$ dans Aut$_\varepsilon$($\mathcal{C}(X)$,$\mathcal{U}_0(X)$) tel que $\mathcal{B} \succcurlyeq_w \mathcal{A}$ et $\mathcal{A} \equiv_\ell \mathcal{B}$.*

La preuve de ce théorème est très semblable à la preuve du théorème 3.44, et même plus simple car nous n'avons aucune mise à jour non déterministe qui implique deux horloges (c'est-à-dire des mises à jour de la forme $x :\sim y + c$ avec $\sim \in \{<, \leq, \geq, >\}$). Nous ne la détaillons pas ici.

3.4.5 Résumé des résultats d'expressivité

Dans cette partie, nous avons montré les résultats d'expressivité résumés dans le tableau 3.8 (TA représente la classe *Aut*($\mathcal{C}(X)$,$\mathcal{U}_0(X)$) alors que TA$_\varepsilon$ représente la classe *Aut*$_\varepsilon$($\mathcal{C}(X)$,$\mathcal{U}_0(X)$). Le symbole $>_\ell$ signifie « strictement plus expressifs » (d'un point de vue des langages)).

<table>
<tr><th></th><th>Contraintes non diagonales</th><th>Contraintes générales</th></tr>
<tr><td>$x := c$, $x := y$</td><td rowspan="3">$\equiv_s$ TA</td><td>$\equiv_s$ TA</td></tr>
<tr><td>$x := x + 1$</td><td rowspan="2">Turing</td></tr>
<tr><td>$x := y + c$</td></tr>
<tr><td>$x :< c$, $x :\leq c$</td><td rowspan="4">TA$_\varepsilon$</td><td>$>_\ell$ TA, TA$_\varepsilon$</td></tr>
<tr><td>$x :> c$, $x :\geq c$</td><td rowspan="3">Turing</td></tr>
<tr><td>$x :\sim y + c$</td></tr>
<tr><td>$(y+)c <: x :< (y+)d$</td></tr>
</table>

avec $\sim \in \{<, \leq, \geq, >\}$ et $c, d \in \mathbb{Q}^+$

Tableau 3.8: Résultats d'expressivité

Le modèle des automates temporisés avec mises à jour n'est donc pas beaucoup plus expressif que le modèle des automates temporisés classiques. La transformation qui permet de construire un automate temporisé classique à partir d'un automate temporisé avec mises à jour souffre cependant d'une explosion combinatoire énorme : les mises à jour apparaissent comme une manière de représenter de manière plus **synthétique** certains systèmes. Il est évidemment possible qu'il existe des constructions beaucoup plus simples que celles que nous avons proposées ici. Néanmoins, les exemples préliminaires 3.45 et 3.46 ainsi que l'exemple 3.47 laissent à penser que des constructions beaucoup plus simples n'existent pas. Nous n'avons pas tenté de le montrer formellement car l'intérêt de ces constructions est en fait assez limité : elles nous ont permis de mieux comprendre le pouvoir d'expression des automates temporisés avec mises à jour, mais elles ne visent aucunement à être utilisées en pratique. En effet, dans le chapitre 8 (commençant à la page 157), nous proposerons un algorithme pour tester le vide des automates temporisés appartenant à une classe décidable sans les transformer au préalable en automates temporisés classiques. Cet algorithme sera aussi simple que celui existant pour les automates temporisés classiques : les mises à jour apparaissent donc comme des macros très utiles pour la modélisation des systèmes temporisés.

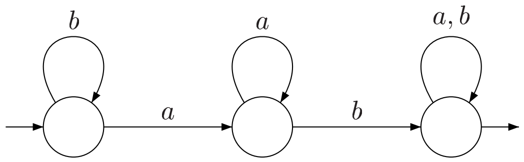
Chapitre 4

A la recherche de Kleene

Ce chapitre correspond à l'article [BP02] et reprend également rapidement les résultats présentés dans [BP99].

Dans le cadre de la théorie des langages formels, le théorème de Kleene [Kle56], ainsi que son extension aux mots infinis due à Büchi [Büc62], permettent de mettre en relation les langages acceptés par des automates finis avec ceux qui s'expriment grâce aux opérations d'union, de concaténation et d'itérations à partir des langages de base constitués des mots de longueur finie et de l'ensemble vide. Ce dernier formalisme, appelé *expressions rationnelles*, est très pratique pour décrire des langages, ce qui le rend adapté à l'écriture de spécifications. Le théorème de Kleene permet alors de transformer ces spécifications en automates finis, puis éventuellement, utilisant d'autres équivalences de formalismes, en des formules logiques.

Considérons par exemple l'automate suivant, qui reconnaît les mots sur l'alphabet $A = \{a, b\}$ qui ont au moins un a, puis, un peu plus tard, un b :



Ce langage peut aussi être décrit par l'expression rationnelle $A^*aA^*bA^*$.

Ces dernières années, quelques travaux sur le sujet ont été réalisés pour les langages temporisés. Parmi ceux-ci, nous pouvons citer ceux réalisés par Eugene Asarin, Paul Caspi et Oded Maler [ACM97, Asa98, ACM01]. Ils ont donné une sémantique un peu différente aux automates temporisés, que nous nommerons les *mots de durée* (pour les distinguer des mots temporisés) et ils ont proposé un théorème de Kleene pour cette sémantique en utilisant des opérations de renommage et l'intersection. Philippe Hermann a complété ces travaux en montrant que les opérations de renommage et l'intersection étaient fondamentales pour leurs expressions rationnelles [Her99]. Cătălin Dima a aussi travaillé sur ce sujet en se focalisant sur le modèle d'automates présenté dans [AFH94] plutôt qu'aux automates temporisés.

Nous nous sommes aussi intéressés à ce sujet et avons proposé un théorème de Kleene/Büchi dans notre article [BP99]. Celui-ci n'utilise pas les opérations de renommage et l'intersection mais considère une autre sémantique pour les automates temporisés, les *générateurs contraints*. Il nécessite l'utilisation de plusieurs opérations de concaténation (chacune étant indexée par un ensemble d'horloges) ainsi que leurs itérées.

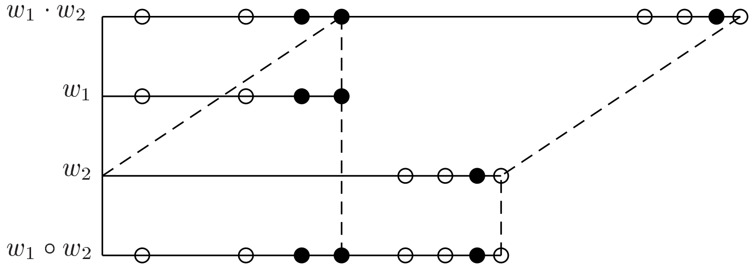
Dans la partie 4.6, nous parlerons rapidement de l'ensemble de ces travaux.

Dans ce chapitre, nous présentons le théorème à la « Kleene/Büchi » que nous avons proposé dans [BP02]. Ce théorème, ainsi que sa preuve, est vraiment très proche du cas des langages formels. Il nous semble que cela renforce son intérêt. Il utilise une nouvelle sémantique des automates temporisés en termes de *langages d'horloges*. Pour cette nouvelle sémantique, nous pouvons facilement définir une opération de concaténation (nous en proposons une seule, qui n'est plus indicée par un ensemble d'horloges) et d'itérations finies ou infinies.
Il nous est alors possible de définir la notion d'*expressions rationnelles*. Ces expressions n'utilisent que les opérations d'union, de concaténation et d'itérations finies ou infinies. Par contre, pour pouvoir exprimer toute la puissance des langages temporisés, nous avons besoin d'un nombre infini de langages de base, même si chaque langage rationnel ne s'exprime qu'à partir d'un nombre fini de ces langages. Nous montrons alors qu'un langage d'horloges est exprimable par une telle expression rationnelle si et seulement s'il est reconnu par un automate temporisé avec mises à jour. Nous verrons que, pour montrer la première implication de cette équivalence, nous allons juste utiliser une extension des constructions faites dans le cadre des langages formels. Pour montrer la deuxième implication de cette équivalence, nous allons procéder en résolvant des systèmes d'équations sur les langages d'horloges, suivant la même idée que pour les langages formels [MY60, Brz62]. Cette idée de résoudre un système d'équations pour calculer le langage accepté par un automate temporisé a déjà été utilisée pour les mots de durée et les générateurs contraints par respectivement Eugene Asarin [Asa98] et nous-mêmes [BP99].

Nous commençons par présenter la sémantique que nous utiliserons dans notre théorème de Kleene, les langages d'horloges. Nous définissons ensuite les langages rationnels puis les langages reconnaissables. L'équivalence entre ces deux notions sera ensuite prouvée. Nous proposerons ensuite quelques applications de ce résultat.

4.1 A propos des langages d'horloges

La sémantique communément utilisée pour décrire le comportement des automates temporisés (avec mises à jour ou non) est celle des mots temporisés. Nous l'avons présentée dans la partie 2.1 et utilisée dans tout ce qui précède. Cependant, la sémantique des mots temporisés n'est pas adaptée à ce que nous cherchons à faire. En effet, cherchant à définir des expressions rationnelles temporisées, la première chose à faire est de definir une concaténation sur les mots temporisés. Il y a, de manière naturelle, deux concaténations qui peuvent être définies :



La première concaténation ($w_1 \cdot w_2$) permet de mettre en séquence deux mots temporisés en décalant le deuxième mot de la durée totale du premier. La deuxième concaténation ($w_1 \circ w_2$), partielle, permet de mettre en séquence deux mots temporisés à partir du moment où le deuxième mot commence après que le premier mot se termine.
Dans le cadre des automates temporisés, la première concaténation correspond à la mise bout à bout de deux automates en remettant toutes les horloges à zéro, alors que la deuxième correspond

à la mise bout à bout de deux automates sans remettre aucune horloge à zéro. Ces deux cas extrêmes ne permettent pas de traiter le cas général où certaines horloges seulement sont remises à zéro. Ce cas général peut être réalisé en rajoutant l'intersection et surtout une opération de renommage, comme cela a été fait dans [ACM97] puis [ACM01]. Afin de s'affranchir de ces deux opérations supplémentaires, nous allons proposer une autre sémantique, les *mots d'horloges*.

4.1.1 Les mots d'horloges

Soit n un entier, un n-*mot d'horloges* est un élément

$$w = (t_0, \tau_0)(a_1, t_1, \tau_1)(a_2, t_2, \tau_2)\ldots \in (\mathbb{T}\times\mathbb{T}^n)\times(\Sigma\times\mathbb{T}\times\mathbb{T}^n)^\infty$$

tel que $(t_i)_{i\geq 0}$ est une suite de dates. Dans un tel mot d'horloges, la suite $(a_i)_{i\geq 0}$ représente la suite des actions qui sont réalisées et la suite de dates $(t_i)_{i\geq 0}$ représente les dates auxquelles ces actions sont réalisées. La suite $(\tau_i)_{i\geq 0}$ représente la valeur des horloges juste après que les actions sont réalisées.

Le n-mot d'horloges w est dit *fini de longueur* p (sa longueur est alors notée $|w|$) si $w \in (\mathbb{T}\times\mathbb{T}^n)\times(\Sigma\times\mathbb{T}\times\mathbb{T}^n)^p$, il est dit *infini* si $w \in (\mathbb{T}\times\mathbb{T}^n)\times(\Sigma\times\mathbb{T}\times\mathbb{T}^n)^\omega$. Un *mot d'horloges* est un n-mot d'horloges pour un entier n donné.
L'ensemble des n-langages d'horloges, *i.e.* les ensembles de n-mots d'horloges, est noté $\mathcal{CL}_n(\Sigma,\mathbb{T})$. L'ensemble des n-mots d'horloges vides est $\Gamma_n = \mathbb{T}\times\mathbb{T}^n$. L'ensemble de tous les langages d'horloges est défini par

$$\mathcal{CL}(\Sigma,\mathbb{T}) = \bigcup_{n\geq 0}\mathcal{CL}_n(\Sigma,\mathbb{T})$$

Deux mots d'horloges peuvent être concaténés s'ils utilisent le même nombre d'horloges et si la valeur des horloges à la fin du premier mot est la même que celle au début du deuxième mot. Plus formellement, si $w = (t_0,\tau_0)(a_1,t_1,\tau_1)\ldots(a_p,t_p,\tau_p)$ est un n-mot d'horloges fini de longueur p et si $w' = (t'_0,\tau'_0)(a'_1,t'_1,\tau'_1)(a'_2,t'_2,\tau'_2)\ldots$ est un n-mot d'horloges fini ou infini, w est dit *compatible* avec w' si $(t_p,\tau_p) = (t'_0,\tau'_0)$. Leur *concaténation*, notée $w\bullet w'$, est alors le n-mot d'horloges $(t''_0,\tau''_0)(a''_i,t''_i,\tau''_i)_{i\geq 1}$ défini par
- $(t''_0,\tau''_0) = (t_0,\tau_0)$,
- si $1\leq i\leq p$, $a''_i = a_i$ et $(t''_i,\tau''_i) = (t_i,\tau_i)$,
- si $i > p$, $a''_i = a'_{i-p}$ et $(t''_i,\tau''_i) = (t'_{i-p},\tau'_{i-p})$.

Il est alors facile de montrer, en utilisant la définition ci-dessus, que la concaténation est associative.

Proposition 4.1 *Soient w, w' et w'' des n-mots d'horloges tels que w soit compatible avec w' et w' soit compatible avec w''. Alors, les propriétés suivantes sont vérifiées :*
- *$(w\bullet w')$ est compatible avec w'',*
- *w est compatible avec $(w'\bullet w'')$,*
- *$(w\bullet w')\bullet w'' = w\bullet(w'\bullet w'')$.*

4.1.2 Opérations sur les langages d'horloges

La concaténation définie sur les mots d'horloges s'étend de manière naturelle et classique aux langages d'horloges. Si L et L' sont deux n-langages d'horloges, nous définissons la concaténation de L et L' par :

$$\begin{array}{rcl} L\bullet L' & = & \{w\bullet w' \mid w\in L,\ w'\in L' \text{ et } w \text{ est compatible avec } w'\} \\ & & \cup\ \{w \mid w \text{ est infini et } w\in L\} \end{array}$$

Concaténer deux langages L et L' revient donc à considérer tous les mots infinis de L et toutes les concaténations de mots finis de L avec des mots (finis ou infinis) de L'.

En utilisant la proposition 4.1, cette concaténation de langages d'horloges est bien sûr aussi associative. De plus, nous obtenons la propriété suivante :

$$\forall L \in \mathcal{CL}_n(\Sigma, \mathbb{T}), \;\; L \bullet \Gamma_n = \Gamma_n \bullet L = L$$

Nous pouvons maintenant définir naturellement les opérations d'itérations (finie et infinie) de langages d'horloges comme suit :

$$\begin{aligned} L^* \quad = \quad \Gamma_n \;\cup\; \{w_1 \bullet w_2 \bullet \ldots \bullet w_h \mid & \forall 1 \leq i \leq h, \; w_i \in L \text{ et} \\ & \forall 1 \leq i \leq h-1, \; w_i \text{ est compatible avec } w_{i+1}\} \end{aligned}$$

Bien sûr, dans cette définition, le dernier mot w_h peut être soit fini, soit infini.

$$\begin{aligned} L^\omega \quad = \quad & \{w_1 \bullet w_2 \bullet \ldots \bullet w_h \mid \forall 1 \leq i \leq h, \; w_i \in L \text{ et} \\ & \qquad \forall 1 \leq i \leq h-1, \; w_i \text{ est compatible avec } w_{i+1} \text{ et} \\ & \qquad w_h \text{ est un mot d'horloges infini } \} \\ & \cup \{w_1 \bullet w_2 \bullet \ldots \mid \forall 1 \leq i, \; w_i \in L \setminus \Gamma_n \text{ et} \\ & \qquad \forall 1 \leq i, \; w_i \text{ est compatible avec } w_{i+1}\} \end{aligned}$$

La propriété suivante résume l'ensemble des propriétés sur les langages d'horloges qui nous seront utiles par la suite. Sa preuve est sans difficulté et repose sur les définitions de la concaténation et des itérations des langages d'horloges. Si L et L' sont des langages d'horloges, la notation $L + L'$ représente l'union des langages L et L'.

Proposition 4.2 *Soient L, L' et L'' trois n-langages d'horloges. Alors, les propriétés suivantes sont vérifiées :*

1. $L \bullet (L' + L'') = (L \bullet L') + (L \bullet L'')$,
2. $(L + L') \bullet L'' = (L \bullet L'') + (L' \bullet L'')$,
3. $(L \bullet L') \bullet L'' = L \bullet (L' \bullet L'')$,
4. $L^* = \bigcup_{i \geq 0} L^i$ *où* $L^0 = \Gamma_n$ *et* $\forall i \geq 1, \; L^{i+1} = L^i \bullet L$,
5. $L^* \bullet L' = \sum_{i \geq 0} \left(L^i \bullet L'\right)$.

4.1.3 Une équation fondamentale

Le résultat suivant, qui porte sur des équations de langages d'horloges va jouer un rôle très important dans la suite de ce chapitre. Il étend à la fois les résultats de [MY60, Brz62] dans le cadre des langages formels et ceux de [Asa98] dans le cadre temporisé.

Lemme 4.3 *Soient L_1 et L_2 deux n-langages d'horloges de mots finis tels que $L_1 \cap \Gamma_n = \varnothing$. L'équation de n-langages d'horloges de mots **finis***

$$L = L_1 \bullet L + L_2 \tag{4.1}$$

a pour unique solution le langage $L_1^* \bullet L_2$.

PREUVE : Nous commençons par montrer que $K = L_1^* \bullet L_2$ est bien une solution de l'équation. Appliquant la proposition 4.2, nous pouvons écrire :

$$\begin{array}{rcl} L_1 \bullet K + L_2 & = & L_1 \bullet (L_1^* \bullet L_2) + L_2 \\ & = & (L_1 \bullet L_1^*) \bullet L_2 + L_2 \\ & = & (L_1 \bullet L_1^* + \Gamma_n) \bullet L_2 \\ & = & L_1^* \bullet L_2 \\ & = & K \end{array}$$

Supposons que L est une solution de l'équation. Il est immédiat que $L_2 \subseteq L$ et donc que $L_1 \bullet L_2 \subseteq L$. Par induction, nous pouvons montrer que pour tout $i \geq 0$, $L_1^i \bullet L_2 \subseteq L$. Ainsi, $\sum_{i \geq 0} (L_1^i \bullet L_2) \subseteq L$. Utilisant la proposition 4.2, nous déduisons que $L_1^* \bullet L_2 \subseteq L$ et donc que toute solution à l'équation contient en particulier le langage K.
Supposons maintenant que L est une solution de l'équation et que L contient strictement K. Soit w un mot d'horloges dans l'ensemble $L \setminus K$ dont la longueur est minimale parmi les mots de $L \setminus K$. Il y a alors deux cas possibles :

1. soit w est dans L_2 : cela est impossible vu que $L_2 \subseteq K$.
2. soit w est dans $L_1 \bullet L$. Comme w est un mot d'horloges fini, il existe $w_1 \in L_1$ et $w' \in L$ tels que $w = w_1 \bullet w'$. Si $w' \in K$, alors $w \in L_1 \bullet K \subseteq K$, ce qui contredit l'hypothèse. Sinon, nous obtenons que $w' \in L \setminus K$ et, comme $L_1 \cap \Gamma_n = \varnothing$, $|w'| < |w|$ ce qui contredit l'hypothèse de minimalité faite sur w.

Ainsi, $L = K$ et l'équation (4.1) a une unique solution, $L_1^* \bullet L_2$. □

4.2 Langages rationnels

Dans le cadre des langages formels, les expressions rationnelles sont construites en utilisant les opérations d'union, de concaténation et d'itération à partir d'un nombre fini d'objets de base dont la sémantique est la suivante :

– $\varnothing$, l'ensemble vide,
– $\{\varepsilon\}$, le langage restreint au mot vide,
– $\{a\}$, le langage réduit à un mot de longueur un.

Considérons maintenant le n-langage d'horloges

$$L_1 = \{(t_0, \tau_0)(a_1, t_1, \tau_1) \mid t_0, t_1 \in \mathbb{T}, \tau_0, \tau_1 \in \mathbb{T}^n\}$$

et supposons que nous souhaitons construire une « expression rationnelle d'horloges » définissant L_1. Si, comme dans le cas des langages formels, nous voulons que les objets de base correspondent à des langages réduits à un unique mot de longueur un, nous devons écrire L_1 comme une union non dénombrable (dès que $\mathbb{T}$ n'est pas dénombrable) de langages définis par des objets de base :

$$L_1 = \bigcup_{(t_0, \tau_0), (t_1, \tau_1) \in \mathbb{T} \times \mathbb{T}^n} \{(t_0, \tau_0)(a_1, t_1, \tau_1)\}$$

Ceci ne correspond pas à ce que nous souhaitons faire car la principale idée qui se cache derrière les expressions rationnelles est d'utiliser un nombre **fini** d'opérations pour définir, à partir des objets de base, des objets plus complexes.

En suivant ce fil directeur, nous proposons une définition d'expressions rationnelles d'horloges qui va permettre, comme dans le cas des langages formels, d'exprimer tous les langages rationnels à partir des objets de base en utilisant un nombre **fini** d'opérations. Remarquons tout de même que, pour pouvoir exprimer des classes de langages d'horloges intéressantes, il faudra utiliser une infinité d'objets de base. Cependant, chaque langage pourra être exprimé grâce à un nombre fini de tels objets.

4.2.1 Expressions rationnelles d'horloges

L'ensemble des *expressions rationnelles sur n horloges* sur un ensemble d'actions Σ et sur un domaine de temps $\mathbb{T}$ est défini par la grammaire suivante :

$$\begin{array}{rcl} E & ::= & \varnothing \\ & | & \varepsilon_n \\ & | & \langle g, a, up\rangle \;\text{ avec } a \in \Sigma,\ g \in \mathcal{C}(X),\ up \in \mathcal{U}(X) \\ & | & E + E \\ & | & E \bullet E \\ & | & E^* \\ & | & E^\omega \end{array}$$

si X désigne un ensemble de n horloges.

La sémantique de ces expressions rationnelles d'horloges associe à chaque expression rationnelle E un langage d'horloges $[\![E]\!]$ défini inductivement comme suit :

Les expressions triviales : $[\![\varnothing]\!] = \varnothing$
$[\![\varepsilon_n]\!] = \Gamma_n = \mathbb{T} \times \mathbb{T}^n$

Les expressions de base : $[\![\langle g, a, up\rangle]\!] = \{(t,\tau)(a, t+t_0, \tau') \in (\mathbb{T}\times\mathbb{T}^n) \times (\Sigma \times \mathbb{T} \times \mathbb{T}^n) \mid \tau + t_0 \models g \text{ et } \tau' \in up(\tau + t_0)\}$

Union : $[\![E + E']\!] = [\![E]\!] \cup [\![E']\!]$

Concaténation : $[\![E \bullet E']\!] = [\![E]\!] \bullet [\![E']\!]$

Itération finie : $[\![E^*]\!] = [\![E]\!]^*$

Itération infinie : $[\![E^\omega]\!] = [\![E]\!]^\omega$

4.2.2 Langages d'horloges rationnels

Un n-langage d'horloges $L \in \mathcal{CL}_n(\Sigma, \mathbb{T})$ est *n-rationnel* s'il existe une expression rationnelle sur n horloges E telle que $L = [\![E]\!]$. Un langage d'horloges $L \in \mathcal{CL}(\Sigma, \mathbb{T})$ est *rationnel* s'il existe un entier n pour lequel L est n-rationnel.

***Exemple* 4.4** Nous pouvons par exemple exprimer le langage

$$L = \{(a, t_i)_{i=1\ldots n} \mid n \in \mathbb{N} \text{ et } \exists i < j.\ t_j = t_i + 1\}$$

par l'expression 1-rationnelle suivante :

$$\langle \mathsf{Vrai}, a, \varnothing\rangle^* \bullet \langle \mathsf{Vrai}, a, x := 0\rangle \bullet \langle \mathsf{Vrai}, a, \varnothing\rangle^* \bullet \langle x = 1, a, \varnothing\rangle \bullet \langle \mathsf{Vrai}, a, \varnothing\rangle^*$$

4.3 Langages reconnaissables

Nous définissons une notion de langages d'horloges reconnaissables en utilisant les automates temporisés avec mises à jour définis et étudiés dans le chapitre 3.

Nous avons vu qu'un automate temporisé avec mises à jour est un uplet $\mathcal{A} = (Q, X, \Sigma, I, F, R, T)$. La sémantique du chapitre 3 associe un ensemble de mots temporisés à un tel automate. Nous enrichissons cette sémantique *via* les langages d'horloges. Cette idée, relativement simple, va se révéler assez puissante pour pouvoir exprimer tous les langages rationnels.

Si $P = q_0 \xrightarrow{g_1, a_1, up_1} q_1 \xrightarrow{g_2, a_2, up_2} q_2 \cdots$ est un chemin dans $\mathcal{A}$, si $(t_0, v_0) \in \mathbb{T} \times \mathbb{T}^n$ est une valuation initiale (n désigne le cardinal de X et nous confondons $\mathbb{T}^n$ et $\mathbb{T}^X$), une exécution initialisée par (t_0, v_0) sur P est de la forme

$$R = \langle q_0, t_0, v_0 \rangle \xrightarrow{g_1, a_1, up_1} \langle q_1, t_1, v_1 \rangle \xrightarrow{g_2, a_2, up_2} \ldots$$

où pour tout $i > 0$, $v_{i-1} + (t_i - t_{i-1}) \models g_i$ et $v_i \in up_i(v_{i-1} + (t_i - t_{i-1}))$. L'étiquette de l'exécution R est le mot d'horloges

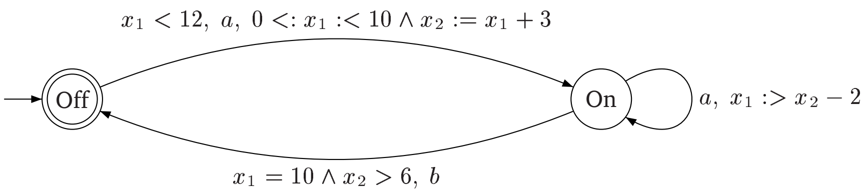
$$\ell(R) = (t_0, v_0)(a_1, t_1, v_1)(a_2, t_2, v_2) \cdots \in (\mathbb{T} \times \mathbb{T}^n) \times (\Sigma \times (\mathbb{T} \times \mathbb{T}^n))^\infty$$

Si le chemin P est constitué d'un unique état, le mot d'horloges associé à l'exécution de valuation initiale $(t_0, v_0) \in \mathbb{T} \times \mathbb{T}^n$ est tout simplement le mot « vide » (t_0, v_0).

Le langage d'horloges reconnu par l'automate temporisé avec mises à jour $\mathcal{A}$ est alors l'ensemble des étiquettes des exécutions formées sur tous les chemins acceptants de $\mathcal{A}$. Dans ce chapitre, contrairement au précédent, nous n'aurons pas besoin de distinguer la nature des contraintes d'horloges utilisées, ni la nature des mises à jour utilisées, mais nous aurons besoin de distinguer les automates temporisés par leur nombre d'horloges. L'ensemble des automates temporisés avec mises à jour sur l'alphabet Σ avec comme ensemble de dates $\mathbb{T}$ qui utilisent n horloges est noté $\mathit{Aut}_n(\Sigma, \mathbb{T})$.

Un n-langage d'horloges $L \in \mathcal{CL}_n(\Sigma, \mathbb{T})$ est *reconnaissable* s'il existe un automate temporisé avec mises à jour $\mathcal{A} \in \mathit{Aut}_n(\Sigma, \mathbb{T})$ tel que $L = L(\mathcal{A})$. Un langage d'horloges $L \in \mathcal{CL}(\Sigma, \mathbb{T})$ est *reconnaissable* s'il existe un entier n pour lequel L est un n-langage d'horloges reconnaissable.

***Exemple* 4.5** Considérons l'automate temporisé avec mises à jour suivant :



Une exécution possible de cet automate est la suivante :

$$\langle \mathsf{Off}, 0, (0,0) \rangle \xrightarrow{x_1 < 12,\ a,\ up_1} \langle \mathsf{On}, 3.1, (8, 6.1) \rangle \xrightarrow{a,\ up_2} \langle \mathsf{On}, 5.7, (9.1, 8.7) \rangle$$
$$\xrightarrow{x_1 = 10 \wedge x_2 > 6,\ b} \langle \mathsf{Off}, 6.6, (10, 9.6) \rangle \ldots$$

L'étiquette de cette exécution est le mot d'horloges

$$(0, (0,0))(a, 3.1, (8, 6.1))(a, 5.7, (9.1, 8.7))(b, 6.6, (10, 9.6)) \ldots$$

4.3.1 Propriétés de clôture

Nous allons étudier la clôture des langages reconnaissables pour les opérations d'union, de concaténation et d'itérations. Nous allons principalement montrer que les constructions habituelles pour les automates finis peuvent être appliquées aux automates temporisés avec mises à jour.

Le cas de l'union est immédiat et identique au cas des automates finis et des automates temporisés d'Alur et Dill.

Proposition 4.6 *Soient $\mathcal{A}$ et $\mathcal{B}$ deux automates temporisés avec mises à jour dans $Aut_n(\Sigma, \mathbb{T})$. Il existe un automate temporisé avec mises à jour $\mathcal{A} + \mathcal{B} \in Aut_n(\Sigma, \mathbb{T})$ tel que*

$$L(\mathcal{A} + \mathcal{B}) = L(\mathcal{A}) + L(\mathcal{B})$$

Le cas de la concaténation est le point délicat où nous avons vraiment besoin de toute la puissance des langages d'horloges. Nous détaillons donc la construction.

Proposition 4.7 *Soient $\mathcal{A}$ et $\mathcal{B}$ deux automates temporisés avec mises à jour dans $Aut_n(\Sigma, \mathbb{T})$. Il existe un automate temporisé avec mises à jour $\mathcal{A} \bullet \mathcal{B} \in Aut_n(\Sigma, \mathbb{T})$ tel que*

$$L(\mathcal{A} \bullet \mathcal{B}) = L(\mathcal{A}) \bullet L(\mathcal{B})$$

PREUVE : Nous supposons que $\mathcal{A} = (Q_1, X, \Sigma, I_1, F_1, R_1, T_1)$ et $\mathcal{B} = (Q_2, X, \Sigma, I_2, F_2, R_2, T_2)$ ont des ensembles d'états disjoints (et X est un ensemble de n horloges). L'automate $\mathcal{A} \bullet \mathcal{B}$ est défini par

$$(Q_1 \cup Q_2, X, \Sigma, I, F_2, R_1 \cup R_2, T_1 \cup T_2 \cup T)$$

où

$$\begin{cases} I & = & \begin{cases} I_1 & \text{si} \quad I_1 \cap F_1 = \varnothing \\ I_1 \cup I_2 & \text{si} \quad I_1 \cap F_1 \neq \varnothing \end{cases} \\ T & = & \{(q_1, g_1, a_1, up_1, i_2) \mid i_2 \in I_2 \text{ et } \exists f_1 \in F_1 \text{ t.q. } (q_1, g_1, a_1, up_1, f_1) \in T_1\} \end{cases}$$

L'automate $\mathcal{A} \bullet \mathcal{B}$ est représenté sur la figure 4.1.

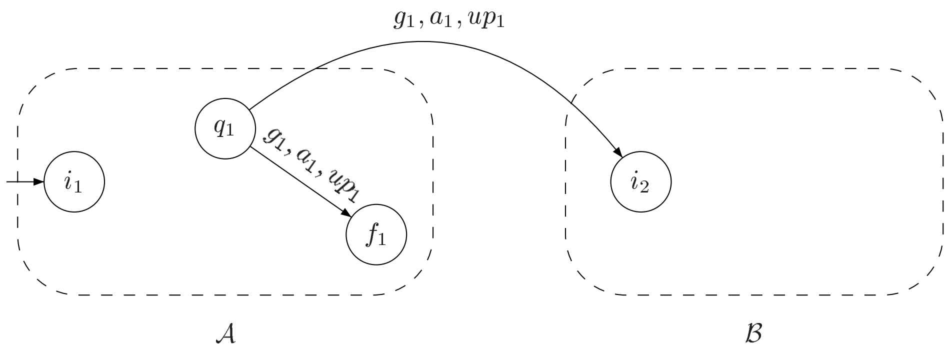


Figure 4.1: Construction de l'automate $\mathcal{A} \bullet \mathcal{B}$

Nous prouvons les deux inclusions entre $L(\mathcal{A} \bullet \mathcal{B})$ et $L(\mathcal{A}) \bullet L(\mathcal{B})$ séparément.

$\boxed{\text{Première inclusion : } L(\mathcal{A}) \bullet L(\mathcal{B}) \subseteq L(\mathcal{A} \bullet \mathcal{B})}$

Si w est un mot d'horloges infini de $L(\mathcal{A})$, w est aussi dans $L(\mathcal{A}) \bullet L(\mathcal{B})$ par définition de la concaténation. Par construction de $\mathcal{A} \bullet \mathcal{B}$, il est aussi immédiat que $w \in L(\mathcal{A} \bullet \mathcal{B})$.

Prenons maintenant un mot d'horloges fini w dans $L(\mathcal{A})$ et w' un mot d'horloges dans $L(\mathcal{B})$ tel que w soit compatible avec w'. Alors w peut s'écrire $w = (t_0, \tau_0)(a_1, t_1, \tau_1)\ldots(a_p, t_p, \tau_p)$ et il existe un chemin P acceptant w dans $\mathcal{A}$:

$$P = q_0 \xrightarrow{g_1, a_1, up_1} q_1 \ldots \xrightarrow{g_p, a_p, up_p} q_p$$

Il existe alors une exécution R sur le chemin P avec comme valuation initiale (t_0, τ_0) telle que $w = \ell(R)$.
De la même façon, w' peut s'écrire comme $w' = (t'_0, \tau'_0)(a'_1, t'_1, \tau'_1)(a'_2, t'_2, \tau'_2)\ldots$ et il existe un chemin acceptant P' dans $\mathcal{B}$:

$$P' = q'_0 \xrightarrow{g'_1, a'_1, up'_1} q'_1 \xrightarrow{g'_2, a'_2, up'_2} q'_2 \ldots$$

Il existe une exécution R' sur P' avec comme valuation initiale (t'_0, τ'_0) telle que $w' = \ell(R')$.

Nous distinguons maintenant deux cas :

<u>1er cas :</u> $w \notin \Gamma_n$

Par construction de l'automate $\mathcal{A}\bullet\mathcal{B}$, le chemin

$$P'' = q_0 \xrightarrow{g_1, a_1, up_1} q_1 \ldots q_{p-1} \xrightarrow{g_p, a_p, up_p} q'_0 \xrightarrow{g'_1, a'_1, up'_1} q'_1 \ldots$$

est un chemin acceptant de $\mathcal{A}\bullet\mathcal{B}$. De plus, comme w est compatible avec w', la propriété $(t_p, \tau_p) = (t'_0, \tau'_0)$ est vérifiée. Ainsi,

$$R'' = \langle q_0, t_0, \tau_0\rangle \xrightarrow{g_1, a_1, up_1} \ldots \langle q_{p-1}, t_{p-1}, \tau_{p-1}\rangle \xrightarrow{g_p, a_p, up_p} \langle q'_0, t'_0, \tau'_0\rangle \xrightarrow{g'_1, a'_1, up'_1} \ldots$$

est une exécution dans $\mathcal{A}\bullet\mathcal{B}$ sur le chemin P'' en prenant comme valuation initiale (t_0, τ_0). Nous en déduisons que $\ell(R'') = w\bullet w' \in L(\mathcal{A}\bullet\mathcal{B})$.

<u>2ème cas :</u> $w \in \Gamma_n$

Dans ce cas, nous avons que $I_1 \cap F_1 \neq \varnothing$ donc, par définition des états initiaux de $\mathcal{A}\bullet\mathcal{B}$, P' est un chemin acceptant de $\mathcal{A}\bullet\mathcal{B}$. Ainsi $\ell(R) = w' = w\bullet w' \in L(\mathcal{A}\bullet\mathcal{B})$.

Deuxième inclusion : $L(\mathcal{A}\bullet\mathcal{B}) \subseteq L(\mathcal{A})\bullet L(\mathcal{B})$

Prenons un mot w'' dans $L(\mathcal{A}\bullet\mathcal{B})$,

$$w" = (t''_0, \tau''_0)(a''_1, t''_1, \tau''_1)(a''_2, t''_2, \tau''_2)\ldots$$

et il existe un chemin acceptant P'' dans $\mathcal{A}\bullet\mathcal{B}$

$$P'' = q''_0 \xrightarrow{g''_1, a''_1, up''_1} q''_1 \xrightarrow{g''_2, a''_2, up''_2} q''_2 \ldots$$

Il existe une exécution R'' sur P'' avec comme valuation initiale (t''_0, τ''_0) telle que $w'' = \ell(R'')$.

De par la définition de $\mathcal{A}\bullet\mathcal{B}$, nous devons distinguer trois cas.

<u>1er cas :</u> P'' ne contient que des états de Q_1.

Alors P'' est un chemin acceptant de $\mathcal{A}$ et $w'' \in L(\mathcal{A})$. Comme $F \cap F_1 = \varnothing$, w'' est infini, donc $w'' \in L(\mathcal{A})\bullet L(\mathcal{B})$ de par la définition de la concaténation.

<u>2ème cas :</u> P'' ne contient que des états de Q_2.

P'' est alors un chemin acceptant de $\mathcal{B}$ et $I_1 \cap F_1 \neq \varnothing$. En particulier, $(t''_0, \tau''_0) \in \Gamma_n \subseteq L(\mathcal{A})$ et donc $w'' = (t''_0, \tau''_0)\bullet w'' \in L(\mathcal{A})\bullet L(\mathcal{B})$.

3ème cas : P'' contient au moins un état de Q_1 et au moins un état de Q_2.
Vue la définition de T, ceci implique qu'il existe un entier p tel que

$$P'' = q''_0 \xrightarrow{g''_1, a''_1, up''_1} q''_1 \dots \xrightarrow{g''_p, a''_p, up''_p} q''_p \xrightarrow{g''_{p+1}, a''_{p+1}, up''_{p+1}} q''_{p+1} \dots$$

où

$$P' = q''_p \xrightarrow{g''_{p+1}, a''_{p+1}, up''_{p+1}} q''_{p+1} \dots$$

est un chemin acceptant (fini ou infini) de $\mathcal{A}$ et il existe un état $f_1 \in F_1$ tel que

$$P = q''_0 \xrightarrow{g''_1, a''_1, up''_1} q''_1 \dots \xrightarrow{g''_p, a''_p, up''_p} f_1$$

soit un chemin acceptant fini de $\mathcal{B}$.
Comme R'' est une exécution sur P'' dans $\mathcal{A} \bullet \mathcal{B}$ en prenant comme valuation initiale (t''_0, τ''_0), il devient immédiat que

$$R = \langle q''_0, t''_0, \tau''_0 \rangle \xrightarrow{g''_1, a''_1, up''_1} \dots \xrightarrow{g''_p, a''_p, up''_p} \langle f_1, t''_p, \tau''_p \rangle$$

est une exécution sur P dans $\mathcal{A}$ en prenant comme valuation initiale (t''_0, τ''_0). De la même manière,

$$R' = \langle q''_p, t''_p, \tau''_p \rangle \xrightarrow{g''_{p+1}, a''_{p+1}, up''_{p+1}} \langle q''_{p+1}, t''_{p+1}, \tau''_{p+1} \rangle \dots$$

est une exécution sur P' dans $\mathcal{B}$ en prenant comme valuation initiale (t''_p, τ''_p).
Dans tous les cas, $w = \ell(R) \in L(\mathcal{A})$ et $w' = \ell(R') \in L(\mathcal{B})$. Ainsi, $w = w' \bullet w'' \in L(\mathcal{A}) \bullet L(\mathcal{B})$ et la preuve est terminée. $\square$

Le cas des itérations finies et infinies peut maintenant être traité plus rapidement car la preuve est similaire à celle de la concaténation.

Proposition 4.8 *Soient $\mathcal{A}$ et $\mathcal{B}$ deux automates temporisés avec mises à jour dans Aut$_n(\Sigma, \mathbb{T})$. Il existe deux automates temporisés avec mises à jour $\mathcal{A}^*$ et $\mathcal{A}^\omega$ dans Aut$_n(\Sigma, \mathbb{T})$ tels que*

$$\begin{cases} L(\mathcal{A}^*) = L(\mathcal{A})^* \\ L(\mathcal{A}^\omega) = L(\mathcal{A})^\omega \end{cases}$$

PREUVE : Supposons que $\mathcal{A} = (Q, X, \Sigma, I, F, R, T)$.

Itération finie : l'automate $\mathcal{A}^*$ est défini par

$$(Q \cup \{q_0\}, X, \sigma, \{q_0\}, \{q_0\}, R, T \cup T' \cup T'')$$

$$\text{où} \begin{cases} q_0 \text{ est un nouvel état qui n'est pas dans } Q, \\ T' = \{(q_0, g, a, up, q) \mid q \in Q \text{ et } \exists i \in I \text{ t.q. } (i, g, a, up, q) \in T\}, \\ T'' = \{(q, g, a, up, q_0) \mid q \in Q \text{ et } \exists f \in F \text{ t.q. } (q, g, a, up, f) \in T\}. \end{cases}$$

Itération infinie : l'automate $\mathcal{A}^\omega$ est défini par

$$(Q \cup \{q_0\}, X, \Sigma, \{q_0\}, \varnothing, R \cup \{q_0\}, T \cup T' \cup T'')$$

$$\text{où} \begin{cases} q_0 \text{ est un nouvel état qui n'est pas dans } Q, \\ T' = \{(q_0, g, a, up, q) \mid q \in Q \text{ et } \exists i \in I \text{ t.q. } (i, g, a, up, q) \in T\}, \\ T'' = \{(q, g, a, up, q_0) \mid q \in Q \text{ et } \exists f \in F \text{ t.q. } (q, g, a, up, f) \in T\}. \end{cases}$$

La preuve que $L(\mathcal{A}^*) = L(\mathcal{A})^*$ et $L(\mathcal{A}^\omega) = L(\mathcal{A})^\omega$ est technique mais semblable à la preuve de la proposition 4.7. Elles ne sont donc pas proposées ici. $\square$

4.4 Un théorème à la Kleene/Büchi

Dans la cadre de la théorie des langages formels, le théorème de Kleene [Kle56] (et son extension aux langages de mots infinis proposée par Büchi [Büc62]) est considéré comme fondamental en informatique théorique. Ce théorème exprime le fait que spécifier un langage *via* une expression rationnelle ou *via* un automate fini revient au même : les deux formalismes sont équivalents.

Nous allons généraliser le théorème de Kleene/Büchi aux langages d'horloges, ce qui confirmera la pertinence de nos définitions de langages d'horloges rationnels et de langages d'horloges reconnaissables. Plus précisément, nous montrons le résultat suivant :

Proposition 4.9 *Soit n un entier. L'ensemble des n-langages d'horloges rationnels coïncide avec celui des n-langages d'horloges reconnaissables.*

Comme conséquence immédiate, nous obtenons le théorème de Kleene/Büchi pour les langages d'horloges :

Théorème 4.10 *L'ensemble des langages d'horloges rationnels coïncide avec celui des langages d'horloges reconnaissables.*

Nous montrons les deux inclusions de la proposition 4.9 (et donc du théorème 4.10) en utilisant les résultats des paragraphes précédents.

4.4.1 Les langages d'horloges rationnels sont reconnaissables

Nous avons déjà vu dans un paragraphe précédent (voir les propositions 4.6, 4.7, 4.8) que les n-langages d'horloges reconnaissables sont clos par les opérations d'union, concaténation et itérations (finies comme infinies). Il est maintenant très facile de montrer que les langages

- $[\![\varnothing]\!]$,
- $[\![\varepsilon_n]\!]$,
- $[\![\langle g, a, up\rangle]\!]$.

sont reconnaissables. Ils sont reconnus respectivement par les automates représentés sur la figure 4.2(a), respectivement 4.2(b), respectivement 4.2(c). Ainsi, par définition des langages d'horloges rationnels, nous avons montré que tous les langages d'horloges rationnels sont reconnaissables.

(a) Automate $\mathcal{A}_\varnothing$ (b) Automate $\mathcal{A}_{\varepsilon_n}$ (c) Automate de base $\mathcal{A}_{\langle g,a,up\rangle}$

Figure 4.2: Automates de base

4.4.2 Les langages d'horloges reconnaissables sont rationnels

Nous allons montrer ce résultat en étendant une des preuves du théorème de Kleene pour les langages formels qui utilise la résolution d'équations de langages (voir par exemple [MY60, Brz62]). Le point important de cette preuve a déjà été décrit dans le lemme 4.3.

Soit L un langage d'horloges reconnaissable. Il existe un automate temporisé avec mises à jour $\mathcal{A} = (Q, X, \Sigma, I, F, R, T)$ tel que $L = L(\mathcal{A})$.
Si q et q' sont des états de Q, nous définissons l'automate $\mathcal{A}_{q,q'} = (Q, X, \Sigma, \{q\}, \{q'\}, \varnothing, T)$ et nous considérons le langage qui lui est associé, $L_{q,q'} = L(\mathcal{A}_{q,q'})$. Par définition d'une exécution dans les automates temporisés avec mises à jour, l'équation suivante est trivialement vérifiée :

$$L(\mathcal{A}) = \sum_{(i,f)\in I\times F} L_{i,f} + \sum_{(i,r)\in I\times R} L_{i,r} \bullet L^{\omega}_{r,r} \tag{4.2}$$

Pour obtenir une expression rationnelle correspondant au langage d'horloges accepté par $\mathcal{A}$, il suffit donc de trouver des expressions rationnelles pour les langages d'horloges $L_{q,q'}$ (avec $q,\ q' \in Q$).
Supposons maintenant que $f \in Q$ est fixé. La famille de langages d'horloges $(L_{q,f})_{q\in Q}$ vérifie le système d'équations suivant :

$$\begin{cases} L_{q,f} = \sum\limits_{(q,g,a,up,q')\in T} L(\mathcal{A}_{\langle g,a,up\rangle}) \bullet L_{q',f} + \Gamma_n & \text{si } q = f \\ L_{q,f} = \sum\limits_{(q,g,a,up,q')\in T} L(\mathcal{A}_{\langle g,a,up\rangle}) \bullet L_{q',f} & \text{sinon} \end{cases}$$

Nous prenons un ordre arbitraire sur les états de Q, $q_1 < q_2 < \ldots < q_n$. L'équation ayant pour membre gauche $L_{q_n,f}$ peut être résolue en prenant $L_{q_n,f}$ comme inconnue et en utilisant le lemme 4.3 car l'hypothèse de ce lemme est bien vérifiée ($[\![\varepsilon_n]\!] \cap L(\mathcal{A}_{\langle g,a,up\rangle}) = \varnothing$ pour tout triplet (g, a, up)). Nous obtenons alors :

$$L_{q_n,f} = \left(\sum_{(q_n,g,a,up,q_n)\in T} [\![\langle g,a,up\rangle]\!] \right)^{*} \bullet \left(\sum_{\substack{(q_n,g,a,up,q')\in T,\\ q'\neq q_n}} [\![\langle g,a,up\rangle]\!] \bullet L_{q',f} \ \ (+\Gamma_n) \right)$$

Nous remplaçons alors $L_{q_n,f}$ par cette formule dans les $n-1$ autres équations. Nous pouvons donc résoudre ce système en utilisant n fois le lemme 4.3 et en éliminant une variable à chaque fois (il est facile de vérifier que les hypothèses du lemme 4.3 sont vérifiées à chaque étape de la résolution).
La dernière étape permet de montrer que le langage d'horloges $L_{q_1,f}$ peut s'exprimer à l'aide des langages d'horloges élémentaires $[\![\varepsilon_n]\!]$ et $[\![\langle g, a, up\rangle]\!]$ en utilisant les opérateurs de composition.

Nous en déduisons donc que $L_{q_1,f}$ est un langage d'horloges rationnel et, par induction, que les langages $L_{q_2,f}, \ldots, L_{q_n,f}$ le sont aussi.

Nous avons donc fini de montrer la proposition 4.9 et le théorème 4.10. □

***Exemple* 4.11** Considérons l'automate $\mathcal{A}$ de l'exemple 4.5, page 93. Nous calculons une expression rationnelle E telle que $[\![E]\!] = L(\mathcal{A})$. Le langage accepté par $\mathcal{A}$ s'exprime de la manière suivante :

$$L(\mathcal{A}) = L_{\text{Off,Off}} \bullet L^{\omega}_{\text{Off,Off}} = L^{\omega}_{\text{Off,Off}}$$

Posant les équations associées à $\mathcal{A}$ comme décrit ci-avant, nous obtenons :

$$\begin{cases} L_{\text{Off,Off}} = [\![\langle x_1 < 12, a, up_1\rangle]\!] \bullet L_{\text{On,Off}} + \Gamma_2 \\ L_{\text{On,Off}} = [\![\langle a, up_2\rangle]\!] \bullet L_{\text{On,Off}} + [\![\langle x_1 = 10 \wedge x_2 > 6, b\rangle]\!] \bullet L_{\text{Off,Off}} \end{cases}$$

Ce système se résout aisément et nous obtenons alors :

$$\begin{array}{rcl} L(\mathcal{A}) & = & ((\llbracket\langle x_1 < 12, a, up_1\rangle\rrbracket \bullet \llbracket\langle a, up_2\rangle\rrbracket^* \bullet \llbracket\langle x_1 = 10 \wedge x_2 > 6, b\rangle\rrbracket)^*)^\omega \\ & = & \llbracket(\langle x_1 < 12, a, up_1\rangle \bullet \langle a, up_2\rangle^* \bullet \langle x_1 = 10 \wedge x_2 > 6, b\rangle)^\omega\rrbracket \end{array}$$

Une expression rationnelle qui décrit $L(\mathcal{A})$ peut donc s'écrire :

$$(\langle x_1 < 12, a, up_1\rangle \bullet \langle a, up_2\rangle^* \bullet \langle x_1 = 10 \wedge x_2 > 6, b\rangle)^\omega$$

Il faut bien noter que l'algorithme de calcul d'expressions rationnelles que nous avons proposé ici n'est pas plus compliqué que celui utilisé pour les automates finis.

4.5 Applications

4.5.1 Restrictions sur les contraintes et sur les mises à jour

Jusqu'à présent, nous n'avons fait aucune hypothèse sur les contraintes d'horloges et les mises à jour utilisées. Cependant, de nombreux travaux sur les automates temporisés ont considéré des ensembles plus restreints de contraintes (comme les contraintes non diagonales) et de mises à jour (par exemple, la classe la plus étudiée est celle des automates temporisés d'Alur et Dill où les seules mises à jour utilisées sont les mises à zéro). Dans le chapitre précédent (chapitre 3), nous avons étudié en détails les sous-classes des automates temporisés avec mises à jour qui sont décidables. Il est donc intéressant d'étudier l'impact de ces restrictions sur le théorème principal de ce chapitre, le théorème de Kleene. En regardant la preuve de ce théorème, nous nous rendons compte que restreindre les contraintes d'horloges et les mises à jour ne change pas la preuve du théorème. Plus précisément, si $\mathcal{C}$ est un ensemble de contraintes d'horloges et $\mathcal{U}$ un ensemble de mises à jour, nous disons qu'un langage d'horloges est $(\mathcal{C},\mathcal{U})$-*rationnel* s'il est rationnel et s'il existe une expression rationnelle qui le décrit telle que tous les langages de base utilisés dans l'expression rationnelle sont de la forme $\langle g, a, up\rangle$ avec $g \in \mathcal{C}$ et $up \in \mathcal{U}$.
Nous obtenons alors la version suivante du théorème de Kleene :

Proposition 4.12 *Soient $\mathcal{C}$ un ensemble de contraintes d'horloges et $\mathcal{U}$ un ensemble de mises à jour. Un langage d'horloges est $(\mathcal{C},\mathcal{U})$-rationnel si et seulement si il est accepté par un automate de Aut$(\mathcal{C},\mathcal{U})$.*

Nous avons présenté dans le chapitre 3, des résultats de décidabilité pour des sous-classes d'automates temporisés avec mises à jour. Nous avons dégagé des ensembles de contraintes d'horloges $\mathcal{C}$ et des ensembles de mises à jour $\mathcal{U}$ pour lesquels nous avons montré que la classe *Aut*$(\mathcal{C},\mathcal{U})$ est décidable. Utilisant le résultat de la proposition 4.12, nous avons donc aussi montré qu'il est décidable de tester le vide des langages $(\mathcal{C},\mathcal{U})$-rationnels pour les ensembles $\mathcal{C}$ et $\mathcal{U}$ dégagés dans le chapitre précédent.

4.5.2 Langages temporisés

Formellement, les langages temporisés définis par Alur et Dill correspondent aux langages sur 0 horloge. Cependant, les notions de langages temporisés reconnaissables et rationnels ne peuvent pas être définis comme étant les langages sur 0 horloge reconnaissables et rationnels. En effet, un langage sur 0 horloge n'apporte aucune information sur les dates, c'est-à-dire qu'un mot temporisé $(a_1,t_1)(a_2,t_2)\ldots$ serait dans un tel langage si et seulement si pour toute suite de dates $(t'_i)_{i\geq 1}$, le mot temporisé $(a_1,t'_1)(a_2,t'_2)\ldots$ est aussi dans le langage.

Il faut donc définir les langages temporisés rationnels et reconnaissables d'une façon un peu plus compliquée. L'idée est qu'un langage temporisé peut aussi être vu comme la projection d'un n-langage d'horloges sur ses deux premières composantes. Nous définissons donc la fonction partielle

$$\begin{array}{rccl} \Pi : & \bigcup_{n\geq 0}\left[(\mathbb{T}\times\mathbb{T}^n)\times(\Sigma\times\mathbb{T}\times\mathbb{T}^n)^\infty\right] & \longrightarrow & (\Sigma\times\mathbb{T})^\infty \\ & (t_0,\tau_0)(a_i,t_i,\tau_i)_{i\geq 1} & \longmapsto & \left\{\begin{array}{ll} (a_i,t_i)_{i\geq 1} & \text{si } (t_0,\tau_0)=(0,0^n) \\ \perp & \text{sinon} \end{array}\right. \end{array}$$

où $\perp$ représente la valeur indéfinie.
Nous pouvons donc définir un langage temporisé (respectivement langage temporisé rationnel, langage temporisé reconnaissable) comme étant la projection $\Pi(L)$ d'un certain langage d'horloges L (respectivement langage d'horloges rationnel L, langage d'horloges reconnaissable L). Si $\mathcal{A}$ est un automate temporisé avec mises à jour, le langage temporisé qu'il accepte est la projection du langage d'horloges qu'il accepte. Une conséquence immédiate de notre théorème 4.10 est la proposition suivante :

Proposition 4.13 *La classe des langages temporisés rationnels correspond à la classe des langages temporisés reconnaissables.*

Il faut remarquer que, contrairement au résultat de [ACM97], nous n'avons ici pas besoin de l'intersection pour définir les langages temporisés rationnels même si l'on utilise tout de même des projections.

4.6 Comparaison avec d'autres travaux

4.6.1 Comparaison avec les résultats de [BP99]

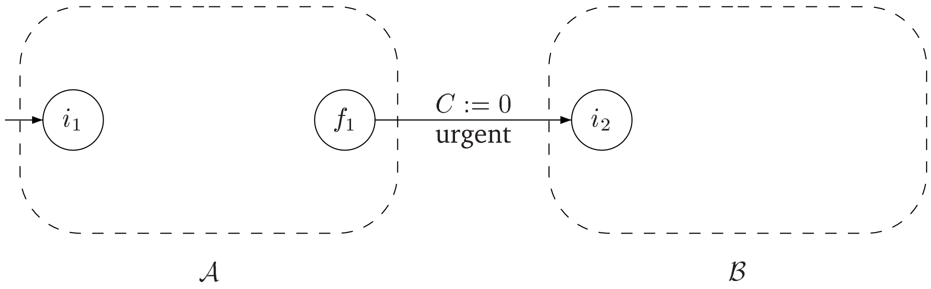
Le travail fait dans [BP99] consistait plus à composer et à décomposer des automates temporisés qu'à obtenir un véritable théorème de Kleene pour les langages temporisés. Nous pouvons réécrire les résultats précédents en ayant en tête le côté décomposition d'automates.
Soient $\mathcal{A}$ et $\mathcal{B}$ deux automates temporisés avec mises à jour. Nous disons qu'ils sont *équivalents* s'ils reconnaissent les mêmes langages d'horloges. Nous considérons alors l'ensemble $\mathcal{M}Aut$ des automates temporisés avec mises à jour obtenus modulairement à partir des automates de base $\mathcal{A}_{\varepsilon_n}$ et $\mathcal{A}_{\langle g,a,up\rangle}$ (décrits sur les figures 4.2(b) et 4.2(c) de la page 97) en utilisant les opérations d'union, de concaténation et d'itérations finies et infinies définies sur les automates temporisés avec mises à jour page 94. Le théorème de Kleene s'écrit alors de la manière suivante :

Proposition 4.14 *Tout automate temporisé avec mises à jour est équivalent à un automate de $\mathcal{M}Aut$. Et réciproquement.*

En particulier, pour tout automate temporisé avec mises à jour $\mathcal{A}$, il existe un automate temporisé défini de manière modulaire qui reconnaît le même langage temporisé.

Nous finissons ce paragraphe en décrivant, sans détailler, les résultats obtenus dans [BP99]. Dans cet article, nous ne considérons pas les langages d'horloges mais nous considérons les automates temporisés comme une fonction qui à toute valuation initiale et à tout mot temporisé associe un ensemble de valuations finales (cette fonction est appelée un *générateur contraint*). Un mot temporisé est alors accepté par un automate temporisé à partir d'une valuation initiale donnée si l'ensemble des valuations finales associé au mot temporisé et à la valuation initiale par le générateur contraint (associé à l'automate temporisé) est non vide.

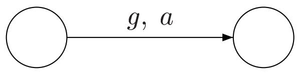
Ayant décrit ces générateurs, nous avons alors défini des opérations de composition sur les automates temporisés indicées par des sous-ensembles d'horloges (une telle opération est notée $\underset{C}{\bullet}$ si C est un sous-ensemble d'horloges). Si $\mathcal{A}$ et $\mathcal{B}$ sont des automates temporisés, la composition $\mathcal{A}\underset{C}{\bullet}\mathcal{B}$ est décrite sur la figure suivante où la notation « urgent » signifie que si cette transition est prise, alors il n'y aura pas eu d'attente dans l'état f_1.



Un mot temporisé u pourra être lu par la composition $\mathcal{A}\underset{C}{\bullet}\mathcal{B}$ s'il peut être décomposé en $v.w$ avec :

– v peut être lu dans $\mathcal{A}$ à partir d'une valuation v_0 en arrivant dans un état final de $\mathcal{A}$ avec comme valuation v_1,
– w peut être lu dans $\mathcal{B}$ à partir de la valuation $v_1[C \leftarrow 0]$ en arrivant à un état final de $\mathcal{B}$.

Les horloges de C sont donc remises à zéro au niveau du passage de $\mathcal{A}$ à $\mathcal{B}$, alors que les horloges qui ne sont pas dans C gardent leur ancienne valeur.
Utilisant toutes ces opérations (il y a donc plusieurs opérations de concaténation) ainsi que leurs itérées finies et infinies, nous avons montré, par une technique d'équations analogue à celle présentée ici, que tout automate temporisé peut se décomposer *via* toutes ces opérations en automates élémentaires de la forme



4.6.2 Comparaison avec les résultats de [ACM97, Asa98, ACM01]

Eugene Asarin, Paul Caspi et Oded Maler ont effectué différents travaux sur le sujet. En 1997, ils ont commencé par proposer un théorème de Kleene dans leur article [ACM97]. Cet article a été complété, peu de temps après, par une nouvelle technique de preuve à base d'équations de langages temporisés [Asa98]. L'ensemble de ces résultats, que nous décrivons brièvement ci-dessous sont présentés dans [ACM01]. La sémantique utilisée pour les automates temporisées est différente de toutes celles précédemment vues. Elle est constituée par ce que nous appellerons par la suite les *langages de durée*, qui sont des éléments du *shuffle* des monoïdes $\mathbb{R}^+$ et Σ^*. En fait ce sont des suites de réels entremêlés d'actions. Par exemple,

$$0.7 \cdot a \cdot b \cdot 3 \cdot 5.4 \cdot ab \cdot c \cdot 0 \cdot a \cdot \varepsilon \cdot 5.4 \cdot a \cdot 0.2$$

est un mot de durée. Ce mot peut se réduire en la « forme normale » suivante :

$$0.7 \cdot ab \cdot 8.4 \cdot abca \cdot 5.4 \cdot a \cdot 0.2$$

Un réel 0.6 entre deux actions a et b signifie qu'il y a un délai de 0.6 unités de temps entre le moment où l'action a a lieu et le moment où l'action b a lieu. Deux actions ab accolées signifient que les deux actions ont lieu à la même date, mais a est effectuée *avant* b.

Cette sémantique s'adapte assez simplement aux automates temporisés. Par exemple, considérons une exécution dans un automate temporisé quelconque

$$(q_0, v_0) \xrightarrow[t_1]{g_1, a_1, C_1} (q_1, v_1) \xrightarrow[t_2]{g_2, a_2, C_2} (q_2, v_2) \xrightarrow[t_3]{g_3, a_3, C_3} (q_3, v_3) \ldots$$

Le *mot de durée* accepté par cette exécution est alors

$$t_1 \cdot a_1 \cdot (t_2 - t_1) \cdot a_2 \cdot (t_3 - t_2) \cdot a_3 ...$$

Les expressions rationnelles sont définies par les éléments de base suivants :

- ε qui représente le mot vide,
- $\underline{a}$ qui représente l'ensemble $\{r \cdot a \mid r \in \mathbb{R}^+\}$

et par les opérateurs suivants :

- l'opérateur de restriction $\langle\varphi\rangle_I$ où I est un intervalle, sa sémantique est en fait la restriction des mots représentés par φ à ceux qui ont une *longueur* dans l'intervalle I,
- l'union,
- deux types de concaténation ainsi que leurs itérations finies respectives,
- l'intersection, le renommage.

Les deux types de concaténation correspondent exactement à celles dont nous avons parlé au début de la partie 4.1.

Ils ont ensuite montré que les expressions rationnelles telles que nous venons de les définir correspondent exactement aux langages de durée acceptés par les automates temporisés. La preuve (expression $\rightarrow$ automate) se fait grossièrement de la même manière que pour notre formalisme. La preuve (automate $\rightarrow$ expression rationnelle) se fait en décomposant les automates temporisés en automates temporisés avec une seule horloge. Une expression rationnelle est alors trouvée pour chaque automate temporisé avec une seule horloge (là aussi, la méthode utilise une résolution de systèmes d'équations, dites *quasi-linéaires*), puis, à l'aide des opérateurs d'intersection et de renommage, ils obtiennent une expression rationnelle pour l'automate initial.

Dans la version préliminaire de ce travail [ACM97], ils ont montré que le renommage était une opération nécessaire pour leur formalisme, et Philippe Herrmann [Her99] a montré que l'intersection était aussi une opération nécessaire.

L'avantage de leur méthode réside dans le fait que leurs langages de durée font abstraction de la notion d'horloges qui se cache un peu derrière notre formalisme. Par contre, l'inconvénient majeur réside dans l'opération de renommage qui éloigne beaucoup de la simplicité des expressions rationnelles dans le cadre des langages formels.

4.6.3 Travaux de Cătălin Dima

Cătălin Dima a aussi travaillé sur des théorèmes de Kleene pour des systèmes temps-réel. La sémantique qu'il utilise couramment est la sémantique des signaux, celle développée dans [ACM97]. Il caractérise grâce à un théorème de Kleene, les langages reconnus par les « event-clock automata » introduits par Alur, Fix et Henzinger dans [AFH94]. Il propose aussi une sous-classe très restreinte de ces automates « event-clock » pour lequel il dégage un théorème de Kleene et qui est close par passage au complémentaire. Ses travaux peuvent être trouvés dans [Dim01a, Dim01b].

4.7 Conclusion

Utilisant une nouvelle sémantique pour les systèmes temporisés beaucoup plus appropriée que les mots temporisés pour décrire les exécutions des automates temporisés avec mises à jour, nous avons proposé un théorème à la Kleene/Büchi assez semblable au théorème dans le cas des langages formels, sans utiliser les opérations d'intersection et de renommage. Il faut bien remarquer que les passages « langages d'horloges rationnels $\longrightarrow$ langages d'horloges reconnaissables » et « langages d'horloges reconnaissables $\longrightarrow$ langages d'horloges rationnels » sont très proches de ceux dans le cas de la théorie des langages formels.

Enfin, il faut noter que cette sémantique qui considère chaque étiquette de transition comme une lettre de l'alphabet n'est pas forcément spécifique aux automates temporisés classiques. Il semble relativement simple d'adapter ce formalisme à d'autres types d'automates.

Chapitre 5

Les langages de données : une approche algébrique

Ce travail a été publié dans [BPT01a]. La version longue est présentée dans [BPT01b].

Dans le cadre des langages formels, la caractérisation des langages reconnaissables la plus simple et la plus élégante est la caractérisation *purement algébrique* basée sur les monoïdes finis. Étant donné un monoïde fini, un langage est dit reconnu par ce monoïde s'il est l'image inverse par un morphisme d'un sous-ensemble du monoïde. Par exemple, le langage L défini par $A^*aA^*bA^*$ sur l'alphabet $A = \{a, b\}$ est accepté par le monoïde

$$M = \{0, 1, x, y, yx\} \text{ avec les relations } x^2 = x, \;\; y^2 = y \text{ et } xy = 0$$

En effet, considérons le morphisme $\varphi : A^* \longrightarrow M$ défini par $\varphi(a) = x$ et $\varphi(b) = y$. Nous avons alors $L = \varphi^{-1}(\{0\})$.

Le résultat d'équivalence entre ces langages et ceux acceptés par des automates finis a permis de montrer de nombreux résultats qui font le lien entre la théorie des langages et l'algèbre [Pin86, Pin96]. Par exemple, Schützenberger a montré que la classe des *langages sans étoile* (*i.e.* la classe de langages pouvant être définis par une expression rationnelle n'utilisant pas l'opérateur d'itération, appelé aussi l'*étoile*) coïncide avec la classe des langages dont le monoïde syntaxique est *apériodique* [Sch65]. Bien que de nature théorique, ces résultats se trouvent également être des **outils** très intéressants. Par exemple, utilisant à la fois le théorème de Kamp [Kam68] et le résultat que nous venons de citer, nous obtenons un algorithme qui permet de décider si un langage peut être défini par une formule de LTL [Pnu77]. L'algorithme consiste à calculer le monoïde syntaxique du langage considéré et de tester s'il est apériodique ou pas. Si c'est le cas, le langage peut être défini par une formule de LTL, qui peut être calculée car les preuves des théorèmes donnent des résultats effectifs[Pin96, PP01].

Dans le cadre des langages temporisés, nous n'avons trouvé dans la littérature aucune proposition de formalisme purement algébrique. Pour pallier ce manque et proposer un tel formalisme, nous avons été amenés à nous placer dans un cadre plus général que le cadre temporisé, le cadre des *langages de données*. Nous allons tout à la fois considérer un alphabet fini d'actions Σ et un ensemble quelconque de données $\mathcal{D}$ (cet ensemble peut être fini ou infini, il peut être un domaine de temps ou n'importe quoi d'autre). Un mot de données est alors une suite de paires (a, d) où $a \in \Sigma$ et $d \in \mathcal{D}$. Nous verrons que, dans ce cadre, la notion de reconnaissance par monoïde ne peut pas être obtenue aussi simplement que dans le cadre des langages formels. L'idée est alors de définir un mécanisme un peu plus complexe pour accepter les langages de données. Notons que, dans d'autres cadres, de tels mécanismes plus complexes ont déjà été définis et utilisés. Nous illustrerons ce point un peu plus loin, à la page 107.

Dans ce chapitre, nous décrivons notre mécanisme de reconnaissance par monoïdes. Nous étudions ensuite diverses propriétés de ce mode de reconnaissance ainsi que les rôles des différents composants du mécanisme. Nous verrons en particulier que le monoïde joue un rôle fondamental, c'est-à-dire que deux variétés de monoïdes n'engendrent pas les mêmes ensembles de langages.

Nous proposons ensuite une notion d'automates de données, un modèle d'automates qui permet d'accepter des langages de données. Puis nous montrons le théorème qui permet de relier entre eux les deux modes de reconnaissance proposés : le formalisme des automates de données est équivalent au formalisme de reconnaissance par monoïdes. Ceci nous permet alors de définir une notion assez robuste de *langages de données reconnaissable*. Nous remarquons aussi à ce niveau que notre preuve d'équivalence est très proche de celle dans le cadre des langages formels, ce qui, nous pensons, renforce l'intérêt de nos définitions.

Dans le cas où l'ensemble des données est un ensemble de temps, nous comparons notre modèle aux automates temporisés classiques et montrons que notre modèle est strictement plus expressif.

Nous proposons ensuite une condition suffisante de décidabilité pour les langages de données reconnaissables. Ce résultat de décidabilité est basé sur une construction d'un « automate des régions », comme celui présenté dans le cadre des automates temporisés classiques dans la partie 2.3.2.

Nous poursuivons notre étude par deux extensions possibles des automates de données. Une des extensions permet une réflexion sur notre notion de reconnaissance par monoïdes. Nous étudions ensuite une version non déterministe des notions abordées jusqu'à présent. Nous terminons ce chapitre en plaçant notre travail dans le cadre de la théorie des langages formels sur un alphabet infini. Nous y récapitulons des travaux existants dans ce domaine.

Définitions de base. Nous considérons, outre l'alphabet fini d'actions Σ, un ensemble de données $\mathcal{D}$, *a priori* fini ou infini. Par exemple, $\mathcal{D}$ peut être un domaine de temps, comme dans le cadre des langages temporisés, mais il peut être n'importe quel autre domaine, par exemple un alphabet infini sans relation d'ordre, un monoïde... Nous supposons que $\mathcal{D}$ a au moins un élément, l'élément vide, que nous noterons $\perp$ et qui, dans le cas où $\mathcal{D}$ est ordonné et minoré, est l'élément initial. Par exemple, si $\mathcal{D}$ est un ensemble de temps, alors $\perp$ sera 0. Un *mot de données* sur Σ et $\mathcal{D}$ est un élément de $(\Sigma \times \mathcal{D})^*$. Un *langage de données* est un ensemble de mots de données.

5.1 Monoïdes et langages de données

5.1.1 Le formalisme de reconnaissance par monoïde

Le cadre général

Le principe de la reconnaissance par monoïde fini consiste à associer à chaque mot d'un monoïde libre Γ^* (où Γ est un alphabet fini ou infini) un élément d'un monoïde fini M et de définir un langage reconnu par M par l'ensemble des mots auxquels sont associés les éléments d'un sous-ensemble particulier P de M. Bien sûr, cette notion de reconnaissance par monoïde n'est intéressante que s'il est possible d'obtenir des propriétés du langage grâce aux propriétés du monoïde.

Dans le cas des langages formels (Γ est alors un alphabet fini), la notion la plus connue de reconnaissabilité par monoïde consiste à utiliser simplement un morphisme $\varphi : \Gamma^* \longrightarrow M$ qui associe à chaque élément de Γ^* un élément du monoïde M. Dans ce cadre, pour décider si un mot $w \in \Gamma^*$ est ou non dans le langage L, il est suffisant de faire les calculs décrits dans le mécanisme 5.1.

Mécanisme 5.1 Le mécanisme pour les langages formels

```
% Initialisation
   m := 1
% Calculs
   Tant que w n'a pas été lu entièrement, faire
      Lire la lettre suivante, a, de w
      Calculer φ(a)
      Calculer m := m.φ(a)
   Fin du tant que
% Fin du calcul
   Si m ∈ P alors répondre «~oui~» sinon répondre «~non~»
```

Il est alors possible de montrer qu'un langage est reconnu par un monoïde fini de cette manière si et seulement s'il est reconnu par un automate fini. En plus de sa simplicité, cette équivalence permet de montrer de très élégants résultats qui font le lien entre les langages formels et l'algèbre, comme nous l'avons déjà dit au début de ce chapitre.

Malheureusement, un tel mécanisme ne serait pas très intéressant pour les langages de données. En effet, si φ est un morphisme de $(\Sigma \times \mathcal{D})^*$ dans un monoïde fini M, l'image par φ de $\Sigma \times \mathcal{D}$ est finie donc le langage, pourtant simple, défini par $L = \{(a,d)(a,d') \mid d \neq d'\}$ ne sera pas reconnaissable par monoïde dès que $\mathcal{D}$ est infini : supposons que M soit un monoïde fini et $\varphi : (\{a\} \times \mathcal{D})^* \longrightarrow M$ un morphisme tels que $L = \varphi^{-1}(P)$ avec $P \subseteq M$. Le monoïde M étant fini alors que $\mathcal{D}$ est infini, il existe deux éléments distincts d_1 et d_2 de $\mathcal{D}$ tels que $\varphi(a,d_1) = \varphi(a,d_2)$. Notons m cet élément. Comme $d_1 \neq d_2$, le mot $(a,d_1)(a,d_2)$ est dans L, donc, en particulier, $m^2 = \varphi((a,d_1)(a,d_2))$ est dans P. Ainsi, $\varphi((a,d_1)(a,d_1)) = m^2 \in P$ entraîne que le mot $(a,d_1)(a,d_1)$ est dans L, ce qui est bien entendu faux. Un mécanisme aussi simple que dans le cas des langages formels ne semble donc pas adapté ici.

Il va donc falloir utiliser un mécanisme de reconnaissance un peu plus compliqué qu'un simple morphisme. Notons que cela n'est pas si exceptionnel que ça, car de tels mécanismes ont déjà été définis et étudiés par exemple dans le cadre des *langages de feuilles* ou même dans le cadre des *programmes sur les monoïdes*. Nous présentons deux exemples.

***Exemple* 5.1** Dans [Bar89], il a été montré que la classe NC^1 des langages reconnus par des circuits booléens de profondeur logarithmique peut être définie de manière algébrique grâce aux *programmes sur les monoïdes* de longueur polynômiale. Un programme sur un monoïde M pour les mots de longueur n est une suite finie

$$p_n = (i_1, \varphi_1) \dots (i_s, \varphi_s)$$

où pour tout j, $1 \leq i_j \leq n$ et φ_j est un morphisme de Σ^* dans M. Le programme est dit de longueur polynômiale si s est polynômial en n. Le calcul du programme p_n sur le mot de longueur n, $u = a_1 \dots a_n$ renvoie la valeur $\varphi_1(a_{i_1}) \dots \varphi_s(a_{i_s})$. Cette valeur appartient au monoïde M. Le langage reconnu par le programme est bien entendu l'ensemble des mots dont le calcul de p_n envoie sur une sous-partie particulière de M. Un autre résultat sur ces programmes caractérise les langages reconnus par des circuits booléens de profondeur constante AC^0 comme étant l'ensemble des langages acceptés par des programmes de longueur polynômiale en utilisant des monoïdes apériodiques [BT88].

***Exemple* 5.2** Présentons maintenant les *langages de feuilles*. Si $\mathcal{M}$ est une machine de Turing non déterministe qui s'arrête en temps polynômial, pour chaque mot d'entrée w et pour chaque exécution sur w dans la

machine de Turing, le résultat du calcul est soit 0, soit 1. Munissant l'ensemble des chemins possibles dans la machine de Turing d'un ordre, pour chaque mot d'entrée w, nous obtenons un mot de $\{0,1\}^*$ qui représente les résultats de toutes les exécutions de la machine de Turing sur ce mot w en mettant les 0 et les 1 dans l'ordre des chemins. Ce mot est appelé le *mot de feuilles* associé à w dans cette machine de Turing. Un *langage de feuilles* est alors l'ensemble des mots de feuilles associés à tous les mots d'un langage grâce à une machine de Turing.
Le modèle de calcul étant défini, nous considérons un exemple simple. Une machine de Turing non déterministe s'arrêtant en temps polynômial accepte un langage L de NP si : un mot est dans L si et seulement si le mot de feuilles qui lui est associé est dans le langage $0^*1(0+1)^*$. En effet, il faut et il suffit qu'il existe un chemin acceptant pour ce mot dans la machine de Turing. On dit alors que NP se réduit à $0^*1(0+1)^*$. Dans [HLS$^+$93], la classe PSPACE a été caractérisée de la sorte : un langage est dans PSPACE si et seulement si il se réduit à un langage régulier *via* les langages de feuilles.

Le cas des langages de données

Le mécanisme que nous proposons ici consiste à utiliser un nombre (fini) de registres qui serviront de mémoire auxiliaire. Il sera possible de stocker dans ces registres certaines des données lues. Par exemple, si nous revenons au langage $\{(a,d)(a,d') \mid d \neq d'\}$, l'idée sera, intuitivement, de stocker la première donnée lue dans un premier registre, puis de stocker la deuxième donnée dans un deuxième registre. La valeur calculée dans le monoïde dépendra alors du test « le premier registre est-il égal au second registre ? », ce qui permettra de reconnaître exactement le langage précité.

De manière grossière, si nous considérons un monoïde fini M, une sous-partie P de M et un nombre fini de registres, un mot de données w est dans le langage si et seulement si les calculs décrits dans le mécanisme 5.2 permettent d'obtenir la réponse « `oui` ».

Mécanisme 5.2 Un mécanisme qui utilise des registres

```
% Initialisation
   m := 1
   Tous les registres sont initialisés à ⊥
% Calculs
   Tant que w n'a pas été lu entièrement, faire
      Lire la lettre suivante, (a,d), de w
      Mettre à jour les registres avec la donnée d
      Calculer la nouvelle valeur m dans le monoïde
   Fin du tant que
% Fin du calcul
   Si m ∈ P alors répondre «˜oui˜» sinon répondre «˜non˜»
```

Maintenant que nous avons décrit le squelette du mécanisme que nous voulons utiliser, il nous reste à préciser plusieurs points :

- comment les registres sont-ils mis à jour ?
- comment est calculée la nouvelle valeur dans le monoïde ?

Le mécanisme que nous construisons doit être suffisamment simple pour ne pas masquer le rôle du monoïde. Voilà ce que nous proposons alors comme réponses aux questions que nous venons de poser :

- les registres sont remis à jour en rangeant la donnée courante dans certains des registres (sans faire de calculs au préalable)
- la nouvelle valeur dans le monoïde va dépendre d'une information **finie et bornée** sur les registres

Nous proposons donc la définition formelle suivante pour un mécanisme à k registres. Une *mise à jour* de k registres est un sous ensemble de $\{1, \ldots, k\}$. Si up est une telle mise à jour et si $\theta \in \mathcal{D}^k$ est un vecteur représentant la valeur des registres, si d est une donnée, nous noterons $up(\theta, d)$ la valeur des registres après la lecture de la donnée et la mise à jour up.

Un *mécanisme à k registres* sur un monoïde fini M est un triplet $\rho = [(up_{m,a})_{m \in M, a \in \Sigma}, \sim, \varphi]$ où

- pour chaque couple $(m, a) \in M \times \Sigma$, $up_{m,a}$ est une mise à jour de k registres, c'est-à-dire un sous-ensemble de $\{1, \ldots, k\}$,
- $\sim$ est une relation d'équivalence d'index fini sur $\mathcal{D}^k$,
- φ est un morphisme de $(\Sigma \times \mathcal{D}^k/\!\sim)^*$ dans M.

Remarquons que si $k = 0$, un mécanisme à k registres est en fait un morphisme de Σ^* dans M, ce qui nous ramène au cadre des langages formels.

Si ρ est un mécanisme à k registres sur un monoïde fini M et si $w = (a_1, d_1)(a_2, d_2) \ldots (a_n, d_n)$ est mot de données de $(\Sigma \times \mathcal{D})^*$, le calcul de ρ sur w est simplement l'élément de M calculé par le mécanisme 5.3. Dans ce calcul, θ est un tableau de taille k correspondant aux k registres et $\overline{\theta}$ représente la classe d'équivalence de θ pour la relation d'équivalence $\sim$.

Mécanisme 5.3 Calculs dans un mécanisme ρ à k registres

```
% Initialisation
   m := 1
   Pour tout 1 ≤ j ≤ k, θ[j] := ⊥
% Calculs
   Pour i allant de 1 à n, faire
      θ := up_{m,a_i}(θ, d_i)
      m := m.φ(a_i, θ̄)
   Fin du pour
% Fin du calcul
   Renvoyer m
```

Le résultat du calcul décrit dans le mécanisme 5.3 est un élément du monoïde M et est noté $\rho(w)$. Dans ce qui suit, si $w = (a_1, d_1) \ldots (a_n, d_n)$ est un mot de données de $(\Sigma \times \mathcal{D})^*$, la valeur de θ à l'étape i de la boucle « `Pour` » est notée θ_i et la valeur de m à cette même étape est notée m_i.

De cette définition d'un mécanisme à k registres, nous pouvons maintenant définir la notion de langage de données reconnu par un monoïde M. Soient L un langage de données défini sur Σ et $\mathcal{D}$) et M un monoïde fini. Le monoïde M *reconnaît* L s'il existe un sous-ensemble P de M et un mécanisme à k registres $\rho = [(up_{m,a})_{m \in M, a \in \Sigma}, \sim, \varphi]$ tel que $L = \rho^{-1}(P)$.
Un langage de données est alors dit *reconnaissable par monoïde* s'il existe un monoïde fini qui le reconnaît.

***Exemple* 5.3** Revenons au langage $L = \{(a, d)(a, d') \mid d \neq \perp, d \neq d'\}$ défini sur $\{a\} \times \mathcal{D}$. Nous allons voir qu'il est reconnu par le monoïde fini $M = \{1, y, y^2, 0\}$ où $y^3 = 0$ et $0.x = x.0 = 0$ pour tout x dans M. Pour montrer cela, nous utilisons deux registres. Nous définissons le mécanisme à deux registres $\rho = [(up_{m,a})_{m \in M, a \in \Sigma}, \sim, \varphi]$ de la façon suivante :
– Les mises à jour sont $up_{1,a} = \{1\}$ et si $z \in M \setminus \{1\}, up_{z,a} = \{2\}$.
– La relation $\sim$ a deux classes d'équivalence, $g_{\neq} = \{(d, d') \mid d \neq d'\}$ et $g_{=} = \mathcal{D}^2 \setminus g_{\neq}$.
– Le morphisme $\varphi : (\{a\} \times \{g_{\neq}, g_{=}\})^* \longrightarrow M$ est défini par $\varphi(a, g_{\neq}) = y$ et $\varphi(a, g_{=}) = 0$.
Avec cette définition, le langage L est reconnu par ρ en prenant $P = \{y^2\}$.

Comme exemple de calcul, considérons le mot de données $(a,d)(a,d')$ avec $d \neq \perp$ et $d \neq d'$.

Dans le monoïde M	1_M	$\xrightarrow[d]{a}$	y	$\xrightarrow[d']{a}$	y^2
Valeurs des deux registres	$\begin{pmatrix} \perp \\ \perp \end{pmatrix}$		$\begin{pmatrix} d \\ \perp \end{pmatrix}$		$\begin{pmatrix} d \\ d' \end{pmatrix}$
Classes d'équivalence	$g_=$		$g_{\neq}$		$g_{\neq}$

Il faut bien remarquer que les registres ne font que stocker des données, mais ils ne font aucun calcul. Si par exemple, nous avions pris comme ensemble de données l'ensemble des rationnels, avec un seul registre, nous aurions pu calculer la différence $d' - d$ au lieu de stocker dans le deuxième registre la valeur de d'. Il aurait alors suffi de tester que $d' - d \neq 0$. Ce genre de calculs n'est pas autorisé dans notre modèle.

***Exemple* 5.4** Le langage de données $\{(a,d_1)\dots(a,d_n)(a,d) \mid d \notin \{d_1,\dots,d_n\}\}$ sur l'alphabet $\{a\}$ et sur l'ensemble de données $\mathcal{D}$ (avec $\mathcal{D}$ infini) n'est reconnu par aucun monoïde fini. Intuitivement, il faudrait un nombre non borné de registres pour stocker chaque donnée d_i, ce que nous ne pouvons pas faire.

Remarque : Si $\mathcal{D}$ n'a qu'un seul élément, $\perp$, alors Σ et $\Sigma \times \mathcal{D}$ sont en bijection et un langage formel sur l'alphabet Σ est reconnu par un monoïde (dans le sens des langages formels) si et seulement si il est reconnu par un monoïde en tant que langage de données. Nous pouvons aussi montrer que si $\mathcal{D}$ est fini, alors tout langage de données reconnaissable par monoïde l'est aussi en tant que langage formel sur l'alphabet $\Sigma \times \mathcal{D}$ (qui est alors un alphabet fini).

5.1.2 Propriétés des langages reconnus par monoïdes

Si M est un monoïde fini et k un entier, l'ensemble des langages de données sur $(\Sigma \times \mathcal{D})$ reconnus par M en utilisant k registres est noté $\mathcal{L}_{M,k}(\Sigma,\mathcal{D})$, ou simplement $\mathcal{L}_{M,k}$ si l'alphabet Σ et l'ensemble des données $\mathcal{D}$ sont évidents. Nous définissons aussi les ensembles

$$\mathcal{L}_M = \bigcup_k \mathcal{L}_{M,k} \quad \text{et} \quad \mathcal{L}_k = \bigcup_M \mathcal{L}_{M,k}$$

Propriétés de clôture

Proposition 5.5 – *L'ensemble $\mathcal{L}_{M,k}$ est clos par passage au complémentaire.*
– *Si $L_1 \in \mathcal{L}_{M_1,k_1}$ et $L_2 \in \mathcal{L}_{M_2,k_2}$, alors $L_1 \cup L_2$ et $L_1 \cap L_2$ sont dans $\mathcal{L}_{M_1 \times M_2, k_1+k_2}$.*

PREUVE : **L'ensemble $\mathcal{L}_{M,k}$ est clos par passage au complémentaire.**
Soit $L \in \mathcal{L}_{M,k}$ un langage de données. Supposons que ρ soit un mécanisme à k registres et que P soit un sous-ensemble de M tel que $L = \rho^{-1}(P)$. Soit $(a_1,d_1)\dots(a_p,d_p)$ un mot de données. Nous avons alors l'équivalence suivante :

$$(a_1,d_1)\dots(a_p,d_p) \in L \quad \Longleftrightarrow \quad \rho((a_1,d_1)\dots(a_p,d_p)) \in P.$$

Donc

$$(a_1,d_1)\dots(a_p,d_p) \notin L \quad \Longleftrightarrow \quad \rho((a_1,d_1)\dots(a_p,d_p)) \in M \setminus P.$$

Si $L_1 \in \mathcal{L}_{M_1,k_1}$ et $L_2 \in \mathcal{L}_{M_2,k_2}$, alors $L_1 \cup L_2$ et $L_1 \cap L_2$ sont dans $\mathcal{L}_{M_1 \times M_2, k_1+k_2}$.
Soient $L_1 \in \mathcal{L}_{M_1,k_1}$ et $L_2 \in \mathcal{L}_{M_2,k_2}$. Supposons que pour $i = 1,2$, $\rho_i = [(up^{(i)}_{m,a})_{m\in M, a\in\Sigma}, \sim_i, \varphi_i]$ est un mécanisme à k_i registres et $P_i \subseteq M_i$ soit un sous-ensemble de M_i tel que $L_i = \rho_i^{-1}(P_i)$.
Nous définissons alors l'entier $k = k_1 + k_2$ et le monoïde $M = M_1 \times M_2$ avec pour loi de monoïde

$$(m_1,m_2)\cdot(m_1',m_2') = (m_1m_1', m_2, m_2').$$

Nous définissons alors l'équivalence $\sim$ sur $\mathcal{D}^k$ par

$$\theta_1\theta_2 \sim \theta_1'\theta_2' \iff \theta_1 \sim_1 \theta_1' \text{ et } \theta_2 \sim_2 \theta_2'$$

et le morphisme $\varphi(a, \overline{\theta_1.\theta_2}) = (\varphi_1(a, \overline{\theta_1}), \varphi_2(a, \overline{\theta_2}))$. Nous définissons enfin pour tout $m \in M$ et pour tout $a \in \Sigma$, la mise à jour de k registres $up_{m,a}$ par

$$up_{m,a} = up^{(1)}_{m,a} \cup (k_1 + up^{(2)}_{m,a})$$

Le langage $L_1 \cup L_2$ est alors reconnu par M en utilisant le mécanisme $\rho = [(up_{m,a})_{m\in M, a\in\Sigma}, \sim, \varphi]$ pour $P = (P_1 \times M_2) \cup (M_1 \times P_2)$ tandis que $L_1 \cap L_2$ est reconnu par M en utilisant ρ pour $P = P_1 \times P_2$. $\square$

Rôles du monoïde et des registres

D'un point de vue algébrique, l'intérêt de notre définition est renforcé par le résultat suivant, qui montre que la structure du monoïde est fondamentale et joue un rôle similaire à celui joué dans le cadre des langages formels. Notons que ce point est fondamental. En effet, l'intérêt de disposer de différents formalismes équivalents pour définir des ensembles de langages réside principalement dans le fait que nous puissions obtenir des propriétés des langages à partir de propriétés du formalisme employé. Dans le cas présent, le but est de recueillir des informations sur les langages à partir des propriétés des monoïdes finis. Le résultat que nous allons présenter montre que la structure du monoïde joue un rôle primordial dans le mécanisme de reconnaissance. En outre, ce résultat montre en particulier que, dans certains cas, augmenter le nombre de registres ne suffit pas pour reconnaître certains langages si le monoïde que l'on utilise n'est pas assez puissant.

Si M et M' sont des monoïdes, nous disons que M *divise* M' si M est le quotient d'un sous-monoïde de M' [Pin86].

Proposition 5.6 *Si M et M' sont des monoïdes finis tels que M ne divise pas M' et M' ne divise pas M, alors $\mathcal{L}_M \neq \mathcal{L}_{M'}$.*

PREUVE : Soit L un langage formel sur l'alphabet Σ. Nous définissons le langage de données suivant

$$L_{\mathcal{D}} = \{(a_1, d_1)\dots(a_n, d_n) \mid a_1 \dots a_n \in L \text{ et } \forall i,\ d_i \in \mathcal{D}\}$$

Soit M un monoïde fini. Nous montrons que

$$L \text{ est reconnu par } M \iff L_{\mathcal{D}} \text{ est reconnu par } M$$

L'hypothèse faite sur M et M' implique que les ensembles de langages formels reconnus respectivement par M, et M' sont incomparables [Pin86]. Nous en déduisons alors que les ensembles de langages de données respectivement reconnus par M et M' sont aussi incomparables.

Nous montrons les deux implications séparément.

Supposons que L soit un langage reconnu par M. Il existe un morphisme $\varphi : \Sigma^* \longrightarrow M$ et un sous-ensemble P de M tels que $L = \varphi^{-1}(P)$. Il est alors facile de montrer que M reconnaît $L_{\mathcal{D}}$ (avec un seul registre).

Supposons maintenant que $L_{\mathcal{D}}$ est un langage de données reconnu par le monoïde fini M en utilisant un mécanisme à k registres $\rho = [(up_{m,a})_{m\in M, a\in\Sigma}, \sim, \varphi]$ et $P \subseteq M$.

En particulier, si $a_1 \dots a_n$ est dans Σ^*, l'image du mot de données $(a_1, \perp) \dots (a_n, \perp)$ dans $(\Sigma \times \mathcal{D}^k/_\sim)^*$ en considérant le calcul fait dans le mécanisme 5.3 est $(a_1, \overline{\perp^k}) \dots (a_n, \overline{\perp^k})$. Nous définissons le morphisme $\psi : \Sigma^* \longrightarrow M$ par $\psi(a) = \varphi((a, \overline{\perp^k}))$. Alors,

$$\begin{array}{rcl} a_1 \dots a_n \in L & \iff & (a_1, \perp) \dots (a_n, \perp) \in L_{\mathcal{D}} \\ & \iff & \rho((a_1, \perp) \dots (a_n, \perp)) \in P \\ & \iff & \varphi((a_1, \overline{\perp^k}) \dots (a_n, \overline{\perp^k})) \in P \\ & \iff & \psi(a_1 \dots a_n) \in P \end{array}$$

Ainsi, M reconnaît le langage L et nous en déduisons le résultat voulu. □

Nous allons étudier les rôles relatifs du monoïde d'une part et des registres d'autre part. Nous allons par exemple montrer qu'augmenter le nombre de registres augmente strictement la classe de langages reconnus. Ce résultat est à rapprocher de celui sur les automates temporisés [HKWT95] qui dit qu'augmenter le nombre d'horloges accroît strictement l'expressivité du modèle. Par contre, si le monoïde est fixé, nous allons montrer que la hiérarchie sur le nombre de registres est stationnaire.

Proposition 5.7 *Les deux propriétés suivantes sont vérifiées :*

1. *La suite* $(\mathcal{L}_k(\Sigma, \mathcal{D}))_{k \geq 0}$ *est strictement croissante, si* $\mathcal{D}$ *est un ensemble infini de données.*
2. *Si* M *est un monoïde fini, la suite* $(\mathcal{L}_{M,k}(\Sigma, \mathcal{D}))_{k \geq 0}$ *est stationnaire. Plus précisément,*

$$\mathcal{L}_{M, 2^{|\Sigma \times M|} - 1} = \mathcal{L}_{M, 2^{|\Sigma \times M|}}$$

PREUVE : *1.* Nous montrons que le langage de données

$$L_k = \{(a, d_1) \dots (a, d_n) \mid i \equiv j \mod (k-1) \implies d_i = d_j\}$$

sur l'alphabet $\{a\}$ et sur l'ensemble de données $\mathcal{D}$ ($\mathcal{D}$ est supposé infini) est reconnu par un monoïde fini en utilisant k registres, mais n'est reconnu par aucun monoïde fini en utilisant strictement moins de k registres.

Pour prouver que L_k est reconnu par un monoïde fini en utilisant k registres, nous définissons tout d'abord le monoïde M comme contenant 1, 0 et comme étant engendré par

$$\{x_P \mid P \text{ partition de } \{0, \dots, k-1\}\}$$

où pour toute partition $P_1, \dots, P_{2k-1}$,

$$\begin{cases} x_{P_1} \dots x_{P_{k-1}} x_{P_k} \dots x_{P_{2k-1}} = x_{P_1} \dots x_{P_{k-1}} \\ \qquad\qquad\qquad\qquad \text{si } P_{k+l} \text{ est compatible}^1 \text{avec } \{l, l+1\} \\ x_{P_1} \dots x_{P_{k-1}} x_{P_k} \dots x_{P_k+l} = 0 \\ \qquad\qquad\qquad\qquad \text{si } P_{k+l} \text{ n'est pas compatible avec } \{l, l+1\} \end{cases}$$

Nous définissons aussi le mécanisme à k registres $\rho = [(up_{m,a})_{m \in M, a \in \Sigma}, \sim, \varphi]$ en définissant séparément les trois composantes.
La relation d'équivalence $\sim$ définie sur $\mathcal{D}^k$ est constituée des classes suivantes : pour toute partition P de $\{0, \dots, k-1\}$,

$$g_P = \{\theta \in \mathcal{D}^k \mid i \text{ et } j \text{ sont équivalents dans } P \iff \theta[i] = \theta[j]\}$$

[1] Une partition d'entiers est compatible avec un ensemble d'entiers E si E est incluse dans une des classes de la partition.

Le morphisme φ est défini par $\varphi(a, g_P) = x_P$.
Les mises à jour sont définies par $up_{x_{P_1} \dots x_{P_{k+l}}, a} = \{l\}$ et $up_{0,a} = \varnothing$.
Il est alors facile de montrer que L est reconnu par M en utilisant le mécanisme ρ.

Nous montrons maintenant, par l'absurde, que L n'est reconnu par aucun monoïde fini en utilisant strictement moins de k registres. Soit $(a_1, d_1) \dots (a_p, d_p)$ un mot de données, nous définissons la suite $(\theta_i)_{i=0\dots p}$ comme dans le calcul présenté dans le mécanisme 5.3.
Nous disons alors que le mot de données $(a_1, d_1) \dots (a_p, d_p)$ est lu sur le chemin $(\overline{\theta_1}, \dots, \overline{\theta_k})$. Pour chaque suite $c = (\overline{\theta_1}, \dots, \overline{\theta_k})$, nous définissons l'ensemble

$$E(c) = \{(a_1, d_1) \dots (a_k, d_k) \text{ lu sur le chemin } c\}$$

Pour chaque tel chemin, il existe un entier $n(c)$ tel que la donnée indicée par $n(c)$ n'est pas stockée dans l'un des registres à la fin du chemin, (*i.e.* si θ_k représente l'état des registres à la fin du chemin, θ_k ne contient pas la donnée indicée par $n(c)$) – un tel entier $n(c)$ existe car si $l < k$, l registres ne peuvent pas stocker k données. Nous définissons pour chaque $1 \leq i \leq k$ une partition P_i de $(\Sigma \times \mathcal{D})^k$ par : $P_i = \bigsqcup_j H_i^{(j)}$ tel que $w,\ w' \in (\Sigma \times \mathcal{D})^k$ sont dans le même $H_i^{(j)}$ si et seulement s'ils ne diffèrent que de la $i^{\text{ème}}$ lettre. Nous restreignons maintenant l'ensemble des données à un sous-ensemble fini D de $\mathcal{D}$ tel que $|D| > |\{c \text{ chemin de longueur } k\}|$. Dans ce cas, il existe un chemin c_0 tel que

$$|E(c_0)| \geq \frac{|D|^k}{|\{c \text{ chemin de longueur } k\}|}$$

Pour tout i, nous avons que $|\{j \mid H_i^{(j)} \text{ is a part of } P_i\}| = |D|^{k-1}$. Ainsi, comme

$$|D| > |\{c \text{ chemin de longueur } k\}|$$

il existe deux mots de données $(a, d_1) \dots (a, d_{n(c_0)}) \dots (a, d_k)$ et $(a, d_1) \dots (a, d'_{n(c_0)}) \dots (a, d_k)$ dans $E(c_0)$ tels que :

$$\begin{array}{cccccccccc}
(a, d_1) & & \dots & & (a, d_{n(c_0)}) & & \dots & & (a, d_k) & \\
& \uparrow & & \uparrow & & \uparrow & & \uparrow & & \uparrow \\
& \theta_1 & & \theta_{n(c_0)-1} & & \theta_{n(c_0)} & & \theta_{k-1} & & \theta_k \\
& \parallel & & \parallel & & \wr & & \wr & & \parallel \\
& \theta'_1 & & \theta'_{n(c_0)-1} & & \theta'_{n(c_0)} & & \theta'_{k-1} & & \theta'_k \\
& \downarrow & & \downarrow & & \downarrow & & \downarrow & & \\
(a, d_1) & & \dots & & (a, d'_{n(c_0)}) & & \dots & & (a, d_k) &
\end{array}$$

Comme $\theta_k = \theta'_k$, si le mot de données $(a, d_1) \dots (a, d_{n(c_0)}) \dots (a, d_k)w$ est dans L, alors le mot de données $(a, d_1) \dots (a, d'_{n(c_0)}) \dots (a, d_k)w$ est aussi dans L. Cependant, ce n'est pas le cas, vu la définition de L. Donc, L n'est reconnu par aucun monoïde en utilisant strictement moins de k registres.

2. Les mises à jour sont paramétrées par des couples de $M \times \Sigma$. Donc, si L est un langage de données reconnu par un monoïde fini M utilisant k registres, nous pouvons définir l'application

$$\begin{array}{rcccl}
\lambda & : & \{1 \dots k\} & \longrightarrow & \{up_{m,a} \mid (m, a) \in M \times \Sigma\} \\
& & i & \longmapsto & \{up_{m,a} \mid (m, a) \in M \times \Sigma \text{ et } i \in up_{m,a}\}
\end{array}$$

et l'équivalence $i \cong j \iff \lambda(i) = \lambda(j)$. Intuitivement, si $i \cong j$, cela signifie que les registres i et j jouent le même rôle, c'est-à-dire qu'ils sont mis à jour toujours en même temps. Il est donc suffisant de ne garder qu'un seul registre pour chaque classe d'équivalence de $\cong$. En outre, la classe de registres i tels que $\lambda(i) = \varnothing$ n'est pas utile.

Plus formellement, soit $(a_1, d_1) \dots (a_p, d_p)$ un mot de données. Une exécution de ce mot dans M est définie par la suite $(a_1, \theta_1) \dots (a_p, \theta_p)$ comme dans le mécanisme 5.3. Nous pouvons remarquer que pour tout $1 \leq h \leq p$, pour tout $i,\ j$,

$$i \cong j \implies \theta_h[i] = \theta_h[j]$$

où $\theta_h[i]$ représente le $i^{\text{ème}}$ composant du vecteur de données θ_h. Nous définissons la fonction

$$\mu : \{1 \dots k\} \longrightarrow \{1 \dots |\cong|\}$$

telle que

$$\mu(i) = \mu(j) \iff i \cong j$$

et si $\lambda(i) = \varnothing$, alors $\mu(i) = 1$. Nous pouvons remarquer que $|\cong| \leq 2^{|\Sigma \times M|}$. Nous définissons sur $\mathcal{D}^h$ (ou sur $\mathcal{D}^{h-1}$ s'il existe i tel que $\lambda(i) = \varnothing$) une relation d'équivalence $\approx$ de la manière suivante :

$$\beta \approx \beta' \iff \theta \sim \theta' \text{ si } \theta_i = \beta_{\mu(i)}$$

La relation d'équivalence $\approx$ est d'indice fini et nous pouvons définir le morphisme

$$\psi : (\Sigma \times \mathcal{D}^h/_{\approx}) \longrightarrow M$$

par $\psi(a, \overline{\beta}) = \varphi(a, \overline{\theta})$ (suivant la définition de $\approx$). Le langage de données L est reconnu par M en utilisant un mécanisme à h registres $[(\mu(up_{m,a}))_{m \in M, a \in \Sigma}, \approx, \psi]$. □

Remarque : Cette proposition montre en particulier que si l'alphabet et le monoïde sont fixés, nous pouvons borner le nombre de registres que nous utilisons. Par contre, ceci n'est plus vrai si seul le nombre de registres est fixé.

***Exemple* 5.8** Considérons par exemple le monoïde fini $\{1, 0, x\}$ où $x^2 = x$. Pour tout entier k, nous considérons l'alphabet $\Sigma_k = \{a_0, a_1, \dots, a_{k-1}\}$. Pour tout mot de données $u \in (\Sigma_k \times \mathcal{D})^*$ et pour tout $i = 1 \dots k-1$, nous notons $\mu_i(u)$ la donnée d (si elle existe) telle que $u = u'\ (a_i, d)\ u''$ où u'' ne contient aucun a_i ; si cette donnée n'existe pas, nous posons $\mu_i(u) = \perp$. Pour tout entier k, nous définissons le langage de données

$$L_k = \{u\ (a_0, d'_1) \dots (a_0, d'_n) \mid u \in ((\Sigma_k \setminus \{a_0\}) \times \mathcal{D})^* \text{ et pour tout } j,\ d'_j \in \cup_{i=1}^{k-1} \{\mu_i(u)\}\ \}$$

Le langage L_k vérifie la propriété suivante :

$$L_k \in \mathcal{L}_{M_0,k}(\Sigma_k, \mathcal{D}) \setminus \bigcup_{k' < k} \mathcal{L}_{k'}$$

En effet, définissons le mécanisme à k registres $\rho = [(up_{m,a})_{m \in M, a \in \Sigma}, \sim, \varphi]$ où $\sim$ est définie par les deux classes d'équivalence

$$g = \{\theta \in \mathcal{D}^k \mid \exists 1 \leq i \leq k-1 \text{ tel que } \theta_0 = \theta_i\} \text{ et } g' = \mathcal{D}^k \setminus g$$

Les mises à jour des registres sont définies par $up_{z,a_i} = \{i\}$ si $0 \leq i \leq k-1$ et $z \in M$. Le morphisme $\varphi : (\mathcal{D} \times \{g, g'\})^* \longrightarrow M$ est défini par $\varphi(a_0, g) = x$, $\varphi(a_0, g') = 0$ et $\varphi(a_i, -) = x$ si $1 \leq i \leq k-1$. En utilisant cette construction, il est facile de montrer que M_0 reconnaît L_k.

Il n'est pas très difficile de montrer que L_k n'est reconnu par aucun monoïde utilisant strictement moins de k registres. Une preuve similaire à celle de la proposition 5.7 (*2.*) permet de le faire.

5.1.3 Les automates de données

Nous allons maintenant proposer une notion d'automates qui vont reconnaître des langages de données. Notre but est de renforcer la notion de reconnaissance par monoïde que nous avons définie en proposant un autre formalisme qui s'avèrera équivalent au premier. Nous proposons alors la définition suivante :

Définition 5.9 *Un* automate de données *est un 8-uplet* $\mathcal{A} = (Q, k, \Sigma, \mathcal{D}, \sim, q_0, F, T)$ *où*

- Q *est un ensemble fini d'états,*
- k *est un entier (qui représente le nombre de registres de l'automate),*
- Σ *est l'alphabet fini d'actions,*
- $\mathcal{D}$ *est l'ensemble des données,*
- $\sim$ *est une relation d'équivalence d'indice fini définie sur* $\mathcal{D}^k$*,*
- $q_0 \in Q$*,* $F \subseteq Q$ *sont respectivement l'ensemble des états initiaux et finals,*
- $T \subseteq Q \times \mathcal{D}^k/_\sim \times \Sigma \times 2^k \times \mathcal{D}^k/_\sim \times Q$ *est l'ensemble des transitions.*

Nous dirons que l'automate de données $\mathcal{A}$ *est* déterministe *s'il vérifie en plus les deux conditions suivantes :*

- *Pour chaque* $(q, g, a) \in Q \times \mathcal{D}^k/_\sim \times \Sigma$*, il y a au plus une remise à jour* up *telle que pour toute transition* $(q, g, a, up', g', q') \in T$*,* $up' = up$*.*
- *Si* (q, g, a, up, g', q'_1) *et* (q, g, a, up, g', q'_2) *sont dans* T*, alors* $q'_1 = q'_2$*.*

Un *chemin* dans l'automate est une suite (finie) de transitions consécutives

$$q_0 \xrightarrow{g_1, a_1, up_1, g'_1} q_1 \xrightarrow{g_2, a_2, up_2, g'_2} q_2 \ldots \xrightarrow{g_p, a_p, up_p, g'_p} q_p$$

Soit $u = (a_1, d_1) \ldots (a_p, d_p)$ un mot de données. Une *exécution* sur le chemin précédent pour le mot u est de la forme :

$$(q_0, \theta_0) \xrightarrow[d_1]{g_1, a_1, up_1, g'_1} (q_1, \theta_1) \xrightarrow[d_2]{g_2, a_2, up_2, g'_2} (q_2, \theta_2) \ldots \xrightarrow[d_p]{g_p, a_p, up_p, g'_p} (q_p, \theta_p)$$

où la suite $(\theta_i)_{i=0\ldots p}$ est définie par :

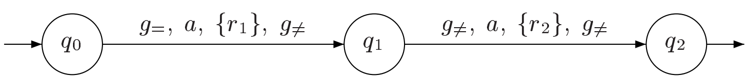
$$\begin{cases} \theta_0 = \perp^k \text{ ($\perp$ est un symbole initial)} \\ \theta_{i+1} = up_{i+1}(\theta_i, d_{i+1}) \text{ si } 0 \leq i \leq p-1 \end{cases}$$

et vérifie :

$$\begin{cases} \overline{\theta_{i-1}} = g_i \text{ si } 1 \leq i \leq p \\ \overline{\theta_i} = g'_i \text{ si } 1 \leq i \leq p \\ q_n \in F \end{cases}$$

[$\overline{\theta}$ représente la classe de θ modulo $\sim$]
L'exécution précédente est *acceptante* si q_0 est un état initial et si q_p est un état final. Le mot u est alors dit accepté par $\mathcal{A}$. L'ensemble des mots acceptés par $\mathcal{A}$ est noté $L(\mathcal{A})$.

***Exemple* 5.10** Le langage décrit par $L = \{(a, d)(a, d') \mid d \neq \perp,\ d \neq d'\}$ est accepté par l'automate de données dessiné sur la figure qui suit.



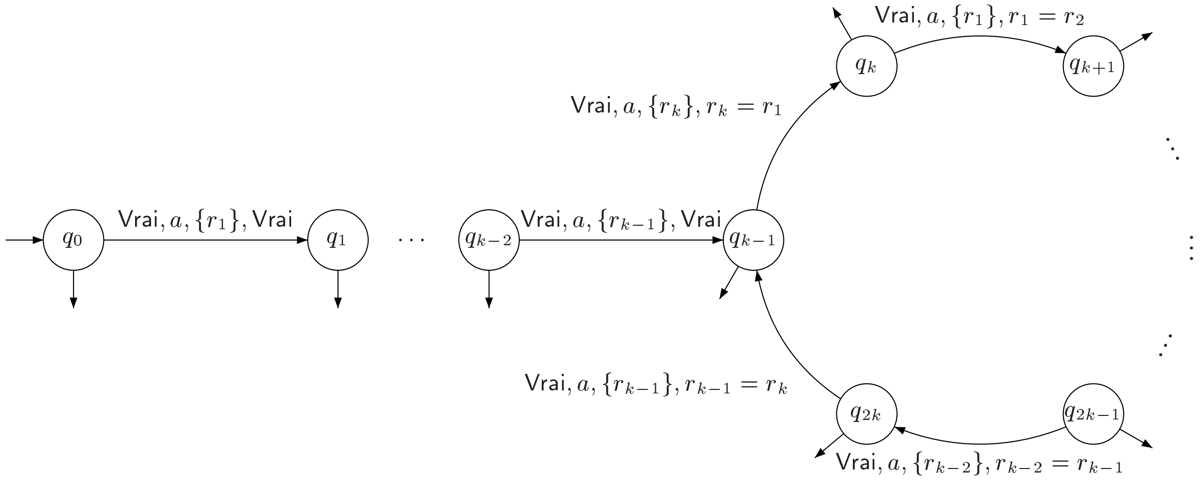
où $g_= = \{(d, d') \in \mathcal{D}^2 \mid d = d'\}$ et $g_{\neq} = \mathcal{D}^2 \setminus g_= = \{(d, d') \in \mathcal{D}^2 \mid d \neq d'\}$.

Dans la suite, pour simplifier l'écriture, les contraintes écrites sur les transitions pourront être des unions de gardes atomiques. Par exemple, si une transition peut être franchie sans condition, alors nous écrirons Vrai sur la transition au lieu d'écrire une transition pour chaque classe d'équivalence de la relation d'équivalence.

***Exemple* 5.11** Considérons le langage suivant

$$L_k = \{(a,d_1)\dots(a,d_n) \mid i \equiv j \mod (k-1) \implies d_i = d_j\}$$

Il représente « le » langage périodique que nous avons déjà vu dans la preuve de la proposition 5.7. Ce langage est accepté par l'automate de données suivant :



5.1.4 Équivalence entre les monoïdes et les automates de données

Nous allons montrer dans cette partie que la notion de reconnaissance par monoïdes que nous avons définie dans la partie 5.1.1 correspond au formalisme des automates de données déterministes que nous venons de définir.

Théorème 5.12 *Soit L un langage de données sur l'alphabet Σ avec comme ensemble de données $\mathcal{D}$. Alors L est reconnu par un automate de données déterministe si et seulement s'il est reconnu par un monoïde fini.*

Ce théorème est très similaire au théorème d'équivalence entre monoïdes et automates dans le cadre des langages formels (voir [Pin86] pour une présentation de ce théorème). Comme nous allons le voir en écrivant la preuve, les transformations « du monoïde à l'automate » et « de l'automate au monoïde » sont semblables à celles dans le cadre des langages formels.

PREUVE : **Implication « si ».** Tout d'abord, nous supposons que $L \subseteq (\Sigma \times \mathcal{D})^*$ est reconnu par le monoïde fini M en utilisant le mécanisme à k registres $\rho = [(up_{m,a})_{m\in M, a\in\Sigma}, \sim, \varphi]$ et avec l'ensemble acceptant $P \subseteq M$. Nous construisons alors l'automate de données sur Σ et $\mathcal{D}$,

$$\mathcal{A} = (Q, k, \Sigma, \mathcal{D}, \sim, q_0, F, T)$$

de la manière suivante :

- L'entier k et la relation d'équivalence $\sim$ proviennent du mécanisme ρ,
- $Q = M$ et $q_0 = 1_M$
- $T = \{(m, g, a, up_{m,a}, g', m') \mid m \in M,\ g, g' \in \mathcal{D}^k_\sim,\ a \in \Sigma,\ m' = m.\varphi(a, g')\}$.

Nous montrons que $\mathcal{A}$ est un automate de données déterministe et que $L(\mathcal{A}) = L$. Constatons tout d'abord que si m est un état de $\mathcal{A}$, si g est une classe d'équivalence et si a est une action, alors il y a une unique mise à jour up telle que $\mathcal{A}$ a une transition $(q, g, a, up, -, -)$, c'est-à-dire que $up = up_{m,a}$. De plus, si g' est une classe d'équivalence, l'état m' tel que $(m, g, a, up_{m,a}, g', m')$ est une transition de $\mathcal{A}$, est définie de manière unique. Ainsi, $\mathcal{A}$ vérifie l'hypothèse de déterminisme définie dans la partie 5.1.3, page 115.

Supposons que le mot de données $w = (a_1, d_1) \dots (a_p, d_p)$ est dans L. Les suites $(\theta_i)_{i=0\dots n}$ et $(m_i)_{i=0\dots n}$ définies par :

$$\begin{cases} \theta_0 = \perp^k \\ \theta_{i+1} = up_{m_i, a_{i+1}}(\theta_i, d_{i+1}) \end{cases} \qquad \text{et} \qquad \begin{cases} m_0 = 1_M \\ m_{i+1} = m_i \ \varphi(a_{i+1}, \overline{\theta_{i+1}}) \end{cases}$$

vérifient que $m_n \in P$. Considérons le chemin suivant dans $\mathcal{A}$

$$\begin{array}{ccccccccc} 1_M & \xrightarrow[d_1]{\overline{\theta_0}, a_1, up_{1_M, a_1}, \overline{\theta_1}} & m_1 & \xrightarrow[d_2]{\overline{\theta_1}, a_2, up_{m_1, a_2}, \overline{\theta_2}} & m_2 & \dots & \xrightarrow[d_n]{\overline{\theta_{n-1}}, a_n, up_{m_{n-1}, a_n}, \overline{\theta_n}} & m_n \\ \theta_0 & & \theta_1 & & \theta_2 & & & \theta_n \end{array}$$

C'est un chemin acceptant pour w dans $\mathcal{A}$ car $\theta_{i+1} = up_{m_i, a_{i+1}}(\theta_i, d_{i+1})$ et $m_{i+1} = m_i \varphi(a_{i+1}, \overline{\theta_{i+1}})$.

Réciproquement, supposons que $w = (a_1, d_1) \dots (a_p, d_p)$ est dans $L(\mathcal{A})$ et considérons l'exécution

$$\begin{array}{ccccccccc} 1_M & \xrightarrow[d_1]{g_1, a_1, up_1, g'_1} & m_1 & \xrightarrow[d_2]{g_2, a_2, up_1, g'_2} & m_2 & \dots & \xrightarrow[d_n]{g_n, a_n, up_n, g'_n} & m_n \\ \theta_0 & & \theta_1 & & \theta_2 & & & \theta_n \end{array}$$

où $\theta_0 = \perp^k$ et $\theta_{i+1} = up_{m_i, a_{i+1}}(\theta_i, d_{i+1})$ vérifient $\overline{\theta_i} = g_{i+1}$ et $\overline{\theta_{i+1}} = g'_{i+1}$.a
Ainsi, $\theta_0 = \perp^k$, $\theta_{i+1} = up_{m_i, a_{i+1}}(\theta_i, d_{i+1})$, $m_0 = 1_M$, $m_{i+1} = m_i \varphi(a_{i+1}, \overline{\theta_{i+1}})$ vérifient $m_n \in F$ et $w \in L$.

Implication « seulement si ». Nous supposons maintenant que $L \subseteq (\Sigma \times \mathcal{D})^*$ est reconnu par l'automate de données

$$\mathcal{A} = (Q, k, \Sigma, \mathcal{D}, \sim, q_0, F, T)$$

Nous définissons le monoÍde M comme étant l'ensemble des applications de $Q \times \mathcal{D}^k\!/\!\sim$ dans lui-même. Nous montrons que L est reconnu par M. Le morphisme $\varphi : (\Sigma \times \mathcal{D}^k\!/\!\sim)^* \longrightarrow M$ est défini par

$$(a, g') \longmapsto \Big[(q, g) \mapsto (q', g')\Big]$$

où q' est l'unique état pour lequel il existe une transition (q, g, a, up, g', q') dans $\mathcal{A}$ (l'unicité de q' provient de l'hypothèse de déterminisme faite sur $\mathcal{A}$).

Soit $m \in M$. Nous posons $m((q_0, g_0)) = (q, g)$ (g_0 est la classe d'équivalence de $\perp^k$). A nouveau en raison du déterminisme, pour toute action a, il y a une unique mise à jour up telle qu'il existe une transition $(q, g, a, up, -, -)$ et nous définissons alors $up_{m,a} = up$.

Enfin, nous définissons $P = \{m \mid m((q_0, \overline{\perp^k})) \in P \times \mathcal{D}^k\!/\!\sim\}$. Nous notons L' le langage de données accepté par M en utilisant le mécanisme à k registres $[(up_{m,a})_{m \in M, a \in \Sigma}, \sim, \varphi]$ et l'ensemble P.

Supposons que $w = (a_1, d_1) \dots (a_p, d_p)$ est dans L en utilisant l'exécution suivante :

$$\begin{array}{ccccccccc} q_0 & \xrightarrow[d_1]{g_1, a_1, up_1, g'_1} & q_1 & \xrightarrow[d_2]{g_2, a_2, up_1, g'_2} & q_2 & \dots & \xrightarrow[d_n]{g_n, a_n, up_n, g'_n} & q_n \\ \theta_0 & & \theta_1 & & \theta_2 & & & \theta_n \end{array}$$

où $\theta_0 = \perp^k$ et $\theta_{i+1} = up_{i+1}(\theta_i, d_{i+1})$. Cela vérifie $\overline{\theta_i} = g_{i+1}$, $\overline{\theta_{i+1}} = g'_{i+1}$ et $q_n \in F$.

Soit m_0 l'identité de $Q \times \mathcal{D}^k\!/\!\sim$ et pour $i \geq 0$, soit m_{i+1} le résultat de la composition de m_i avec

$$\varphi(a_{i+1}, \overline{\theta_{i+1}}) = \varphi(a_{i+1}, g'_{i+1})$$

i.e. de m_i avec

$$[(q, \overline{\alpha}) \longmapsto (q', g'_{i+1})]$$

pour l'unique état q' tel qu'il existe une transition $(q, \overline{\alpha}, a_{i+1}, up, g'_{i+1})$.

Par induction, si l'on suppose que $m_i((q_0, \perp^k)) = (q_i, \overline{\theta_i})$, nous obtenons alors que $q' = q_{i+1}$ et $m_{i+1}((q_0, \overline{\perp^k})) = (q_{i+1}, \overline{\theta_i + 1})$ et donc que $w \in L'$.

Réciproquement, supposons que $w = (a_1, d_1) \ldots (a_p, d_p)$ est dans L'. Les suites définies par

$$\begin{cases} \theta_0 = \perp^k \\ \theta_{i+1} = up_{m_i, a_{i+1}}(\theta_i, d_{i+1}) \end{cases} \quad \text{et} \quad \begin{cases} m_0 = 1_M \\ m_{i+1} = m_i\ \varphi(a_{i+1}, \overline{\theta_{i+1}}) \end{cases}$$

vérifient la propriété que $m_n((q_0, \overline{\perp^k})) = (q, \overline{\theta})$ pour tout $q \in F$.

Nous définissons alors $(q_i, \overline{\theta_i})$ par $(q_i, \overline{\theta_i}) = m_i((q_0, \overline{\perp^k}))$. L'exécution

$$\begin{array}{ccccccccc} q_0 & \xrightarrow[d_1]{\overline{\theta_0}, a_1, up_{m_0, a_1}, \overline{\theta_1}} & q_1 & \xrightarrow[d_2]{\overline{\theta_1}, a_2, up_{m_1, a_2}, \overline{\theta_2}} & q_2 & \ldots & \xrightarrow[d_n]{\overline{\theta_{n-1}}, a_n, up_{m_{n-1}, a_n}, \overline{\theta_n}} & q_n \\ \theta_0 & & \theta_1 & & \theta_2 & & & \theta_n \end{array}$$

est acceptante dans $\mathcal{A}$. Ainsi, $w \in L$.

Nous venons donc de montrer l'équivalence entre les monoïdes et les automates. $\square$

Remarquons que les deux transformations présentées ci-dessus ne changent pas le nombre de registres utilisés, ni l'ensemble des mises à jour, ni même la relation d'équivalence.

Un langage de données est dit *reconnaissable* s'il est reconnu par un automate de données déterministe (il serait équivalent de dire qu'il est accepté par un monoïde fini).

5.2 Comparaison avec les automates temporisés classiques

La motivation principale de ce travail était d'obtenir une caractérisation algébrique des langages temporisés. Il apparaît clairement que si $\mathcal{D}$ est un domaine de temps (par exemple $\mathbb{N}$ ou bien $\mathbb{Q}^+$ ou bien $\mathbb{R}^+$), alors les langages temporisés sont des cas particuliers de langages de données (dans lesquels nous forçons le temps à croître).

Proposition 5.13 *Soit $\mathcal{A}$ un automate temporisé (respectivement déterministe) avec n horloges sur le domaine de temps $\mathcal{D}$. Il existe un automate de données (respectivement déterministe) avec $2n+2$ registres qui reconnaît le même langage.*

PREUVE : Soit $\mathcal{A}$ un automate temporisé déterministe qui a n horloges, $\{x_1, \ldots x_n\}$. Nous adjoignons à $\mathcal{A}$ une horloge universelle x_0, c'est-à-dire une horloge qui croît avec le temps, mais qui n'est jamais remise à zéro dans l'automate $\mathcal{A}$.

Nous supposons que $\equiv$ est l'équivalence des régions associée à $\mathcal{A}$ (voir la partie 2.3.3) et nous transformons $\mathcal{A}$ de telle façon que toutes les contraintes apparaissant sur les transitions soient des classes d'équivalence de $\equiv$. Nous allons construire un automate de données (déterministe) $\mathcal{B}$ avec $2n+2$ registres de la manière suivante.

L'ensemble des états de $\mathcal{B}$ est $Q \times \mathcal{F}$ où Q est l'ensemble des états de $\mathcal{A}$ et $\mathcal{F}$ est l'ensemble des fonctions $f : \{x_0, \ldots, x_n\} \longrightarrow \{0, \ldots, 2n+1\}$ tel que pour tout $0 \leq i \leq n$, $f(x_i) \in \{i, n+1+i\}$.

Intuitivement, la valeur de l'horloge x_i va être alternativement stockée dans les registres i et $n+1+i$.
Nous définissons la relation d'équivalence $\equiv_2$ sur $\mathcal{D}^{2n}$ comme étant l'équivalence des régions (avec les mêmes constantes maximales que $\equiv$).
La relation d'équivalence $\sim$ de $\mathcal{B}$ est alors définie par :

$$(\theta_i)_{0\leq i\leq 2n+1} \sim (\theta'_i)_{0\leq i\leq 2n+1} \quad \Longleftrightarrow \quad \begin{cases} \theta_0 < \theta_{n+1} \iff \theta'_0 < \theta'_{n+1} \\ \theta_0 > \theta_{n+1} \iff \theta'_0 > \theta'_{n+1} \\ (\theta_{f(x_0)} - \theta_i)_{1\leq i\leq 2n} \equiv_2 (\theta'_{f(x_0)} - \theta'_i)_{1\leq i\leq 2n} \\ \qquad\qquad \text{si } f(x_0) = \begin{cases} 0 \text{ si } \theta_0 > \theta_{n+1} \\ n+1 \text{ si } \theta_0 < \theta_{n+1} \end{cases} \end{cases}$$

En particulier, si $f \in \mathcal{F}$ est telle que $f(x_0) = \begin{cases} 0 \text{ si } \theta_0 > \theta_{n+1} \\ n+1 \text{ si } \theta_0 < \theta_{n+1} \end{cases}$ alors

$$(\theta_{f(x_0)} - \theta_{f(x_i)})_{1\leq i\leq n} \equiv (\theta'_{f(x_0)} - \theta'_{f(x_i)})_{1\leq i\leq n}$$

Considérons une transition de $\mathcal{A}$:

$$q \xrightarrow{\ g,\ a,\ C := 0\ } q'$$

Pour chaque fonction f de $\mathcal{F}$, nous construisons des transitions dans $\mathcal{A}$ de la manière suivante :

$$q, f \xrightarrow{\ \overline{\theta},\ a,\ \alpha,\ \overline{\theta'}\ } q', f'$$

où :
- $\overline{\theta}$ est une classe d'équivalence quelconque de $\sim$ compatible avec f, c'est-à-dire que si $v \in \overline{\theta}$, alors $v_0 > v_{n+1}$ si $f(x_0) = 0$ alors que $v_0 < v_{n+1}$ sinon,
- $\alpha = \{0, 1, \ldots, 2n+1\} \setminus \{f(x_0), \ldots, f(x_n)\}$,
- $f' \in \mathcal{F}$ est telle que

$$f'(x_0) \quad = \quad \begin{cases} 0 & \text{si } f(x_0) = n+1 \\ n+1 & \text{si } f(x_0) = 0 \end{cases}$$

$$f'(x_i) \quad = \quad \begin{cases} f(x_i) & \text{si } x_i \notin C \\ n+1+i & \text{si } x_i \in C \text{ et } f(x_i) = i \\ i & \text{si } x_i \in C \text{ et } f(x_i) = n+1+i \end{cases}$$

- $\overline{\theta'}$ est une classe d'équivalence de $\sim$ telle que

$$(\beta_i)_{0\leq i\leq 2n+1} \in \overline{\theta'} \implies (\beta_{f'(x_0)} - \beta_{f(x_i)})_{1\leq i\leq n} \in g \text{ et } \beta_{f'(x_0)} > \beta_{f(x_0)}$$

L'automate de données $\mathcal{B}$ que nous venons de construire est déterministe (si $\mathcal{A}$ est déterministe) et reconnaît le même langage (temporisé) que $\mathcal{A}$. $\square$

Le principe de la construction peut être illustrée par l'exemple suivant.

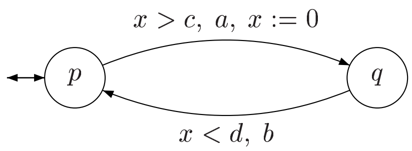


Figure 5.1: Automate temporisé $\mathcal{A}$

Exemple **5.14** Considérons l'automate temporisé à une horloge $\mathcal{A}$ de la figure 5.1.
Nous construisons un automate de données avec 4 registres qui va reconnaître le même langage. Pour faciliter la lecture, nous appellerons r_0, r'_0 les registres associés à l'horloge universelle et r_x, r'_x ceux associés à l'horloge x (au lieu de les appeler 0,1,2,3 comme c'est le cas dans la preuve). À chaque instant, l'un des deux registres r_0 ou r'_0 est actif pour l'horloge universelle, et l'un des deux registres r_x ou r'_x est actif pour l'horloge x. Le registre actif pour l'horloge universelle contient la date du dernier franchissement d'une transition alors que le registre actif pour x contient la date de la dernière remise à zéro de l'horloge x. L'automate de données $\mathcal{B}$ de la figure 5.2 correspond à la construction de la preuve, sauf que, pour simplifier, les contraintes qui apparaissent sur les transitions ne sont pas des classes d'équivalence modulo la relation $\sim$ de la preuve, mais des unions de telles classes.

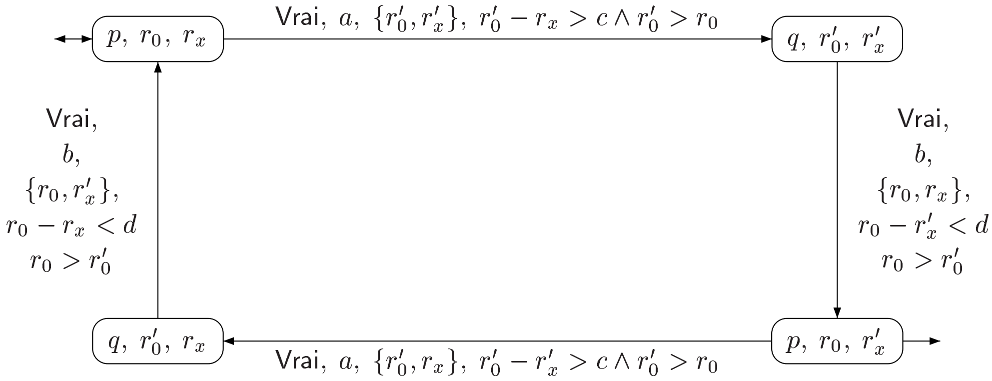


Figure 5.2: Automate de données $\mathcal{B}$

Nous décrivons comment l'automate $\mathcal{B}$ reproduit le comportement de l'automate $\mathcal{A}$. Au début, les registres actifs sont r_0 et r_x (on est dans l'état $(p,\ r_0,\ r_x)$, les registres r_0 et r_x sont initialisés à zéro). Ensuite, pour pouvoir faire l'action a, il faut au préalable vérifier que l'horloge x vaut plus que c. Pour cela, nous commençons par stocker la valeur courante du temps universel dans le registre r'_0. La valeur de l'horloge x est alors égale à la différence $r'_0 - r_x$, c'est pour cette raison que nous testons $r'_0 - r_x > c$. Pour vérifier que le temps a augmenté strictement, nous testons aussi $r'_0 > r_0$. Enfin, il faut aussi remettre à zéro l'horloge. Pour cela, nous mettons à jour le registre r'_x avec la valeur courante. Les registres actifs à la fin de cette transition sont r'_0 et r'_x. Nous recommençons la même explication pour la deuxième transition. La seule différence tient dans le fait que l'horloge x n'est, cette fois, pas remise à zéro. Les registres actifs à la fin de cette transition sont alors r_0 et r'_x (la valeur de x correspond bien à la différence $r_0 - r'_x$).

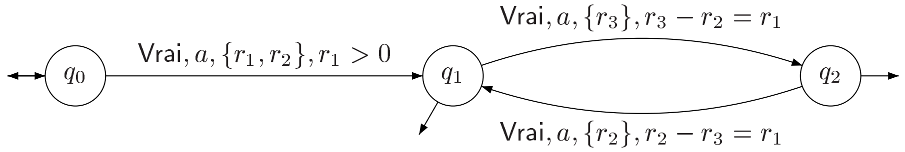
Nous donnons un exemple d'exécution dans les automates $\mathcal{A}$ et $\mathcal{B}$ (les registres actifs de $\mathcal{B}$ sont soulignés et en gras) :

horloge x	0		0		$t_2 - t_1$		0		$t_4 - t_3$	$\dots$	dans $\mathcal{A}$
	p	$\xrightarrow[t_1]{a}$	q	$\xrightarrow[t_2]{b}$	p	$\xrightarrow[t_3]{a}$	q	$\xrightarrow[t_4]{b}$	p	$\dots$	
registre r_0	$\underline{\mathbf{0}}$		0		$\underline{\mathbf{t_2}}$		t_2		$\underline{\mathbf{t_4}}$	$\dots$	dans $\mathcal{B}$
registre r'_0	0		$\underline{\mathbf{t_1}}$		t_1		$\underline{\mathbf{t_3}}$		t_3	$\dots$	
registre r_x	$\underline{\mathbf{0}}$		0		0		$\underline{\mathbf{t_3}}$		$\underline{\mathbf{t_3}}$	$\dots$	
registre r'_x	0		$\underline{\mathbf{t_1}}$		$\underline{\mathbf{t_1}}$		t_1		t_1	$\dots$	

En restreignant l'ensemble de données à un ensemble de temps, nous allons montrer que les automates de données permettent de reconnaître une classe de langages temporisés plus large que celle correspondant aux automates temporisés. Nous avons la propriété suivante :

Lemme 5.15 *Le langage temporisé* $\{(a,\tau)(a,2\tau)\dots(a,n\tau) \mid \tau \in \mathbb{Q}_+\}$ *est reconnu par un automate de données (déterministe)*

PREUVE : Construisons l'automate suivant :



Il reconnaît le langage $\{(a,\tau)(a,2\tau)\dots(a,n\tau) \mid \tau \in \mathbb{Q}_+\}$. □

Cependant, comme il a été montré dans [DZ98], ce langage n'est accepté par aucun automate temporisé classique, ni même par l'extension proposée dans ce même article.

Dans notre cadre, nous pouvons aussi définir des langages un peu plus exotiques comme par exemple le langage $\{(a,t_1)\dots(a,t_n) \mid \forall i,\ t_i$ est un nombre premier$\}$. Ce langage est reconnu par un monoïde ayant deux éléments avec un registre et une relation d'équivalence d'index 2 (la première classe contient les nombres premiers, l'autre classe les nombres non premiers).

5.3 Décidabilité des automates de données

Nous devons tout d'abord remarquer que la classe générale des langages de données reconnaissables est indécidable. Dans la partie 3.2.1, nous avons présenté en détails le modèle de la machine à deux compteurs et nous avons énoncé sa propriété d'être un modèle indécidable. Or, il est relativement facile de simuler une machine à deux compteurs [Min67] avec les automates de données, ce dernier modèle est donc bien indécidable. Nous allons cependant donner une condition suffisante pour pouvoir tester la vacuité d'un langage de données reconnaissable.

5.3.1 Une condition suffisante

Commençons par définir, pour toute mise à jour de k registres up, une relation sur l'ensemble $\mathcal{D}^k/_{\sim}$, notée $\xrightarrow{up}$, de la manière suivante :

$$g \xrightarrow{up} g' \;\; ssi \;\; \exists v \in g,\ \exists d \in \mathcal{D},\ up(v,d) \in g'$$

Pour dégager des sous-classes décidables de notre modèle, nous définissons la condition :

$$g \xrightarrow{up} g' \;\; ssi \;\; \forall v \in g,\ \exists d \in \mathcal{D},\ up(v,d) \in g' \qquad (\dagger)$$

Si $\rho = [(up_{m,a})_{m\in M, a\in\Sigma}, \sim, \varphi]$ est un mécanisme à k registres, nous dirons que ρ satisfait la condition (†) si cette même condition est vérifiée pour toute classe de la relation $\sim$ et pour toute mise à jour $up_{m,a}$. Notre but est de montrer que sous cette condition, le problème du vide est décidable. Le principe de la preuve est similaire à celui de la construction de l'automate des régions proposée dans la partie 3.3.1, page 51.

Théorème 5.16 *Soit L un langage de données reconnaissable sur l'alphabet Σ avec comme ensemble de données $\mathcal{D}$. Supposons que L soit reconnu par le monoïde fini M en utilisant un mécanisme à k registres satisfaisant la condition (†) Alors la vacuité de L est décidable avec une complexité* PSPACE.

PREUVE : Soit $L \subseteq (\Sigma \times \mathcal{D})^*$ un langage de données reconnaissable. Soit M un monoïde fini qui permet de reconnaître L en utilisant un mécanisme à k registres $\rho = [(up_{m,a})_{m \in M, a \in \Sigma}, \sim, \varphi]$ qui vérifie la condition (†). Comme dans la preuve du théorème 5.12, nous construisons un automate de données $\mathcal{A}$ dont les transitions sont :

$$m \xrightarrow{g,\ a,\ up_{m,a},\ g'} m\,\varphi(a, g')$$

Bien sûr, $L = L(\mathcal{A})$. À partir de $\mathcal{A}$, nous construisons un automate fini $\mathcal{B} = (Q, I, F, T)$ où $Q = M \times \mathcal{D}^k/\!\sim$, $I = (1_M, \overline{\perp^k})$, $F = P \times \mathcal{D}^k/\!\sim$ (P est l'ensemble acceptant du monoïde) et T est défini par

$$((m, g), a, (m', g')) \in T \quad \Longleftrightarrow \quad m \xrightarrow{g,\ a,\ up_{m,a},\ g'} m' \text{ et } g \xrightarrow{up_{m,a}} g'$$

Nous allons montrer que, comme la condition (†) est vérifiée, cet automate fini accepte le langage

$$\mathsf{Undata}(L) = \{a_1 \dots a_n \mid \exists d_1, \dots, d_n,\ (a_1, d_1) \dots (a_n, d_n) \in L\}$$

Supposons que $\mathcal{A}$ accepte le mot de données $w = (a_1, d_1) \dots (a_n, d_n)$ sur le chemin :

$$\begin{array}{ccccccccc} m_0 & \xrightarrow[d_1]{g_1, a_1, up_{m_0,a_1}, g_1'} & m_1 & \xrightarrow[d_2]{g_2, a_2, up_{m_1,a_2}, g_2'} & m_2 & \dots & \xrightarrow[d_n]{g_n, a_n, up_{m_{n-1},a_n}, g_n'} & m_n \\ \theta_0 & & \theta_1 & & \theta_2 & & & \theta_n \end{array}$$

En particulier, pour tout i, $\theta_{i+1} = up_{m_i, a_{i+1}}(\theta_i, d_{i+1})$, $\overline{\theta_i} = g_{i+1}$ et $\overline{\theta_{i+1}} = g'_{i+1}$. Ainsi, pour tout i, $g_i \xrightarrow{up_{m_i,a_{i+1}}} g_i'$ et il y a une transition $((m_i, g_i), a_{i+1}, (m_{i+1}, g_i'))$ dans $\mathcal{B}$. Le chemin suivant de $\mathcal{B}$ accepte donc le mot $\mathsf{Undata}(w) = a_1 \dots a_n$.

$$(m_0, g_1) \xrightarrow{a_1} (m_1, g_1') = (m_1, g_2) \xrightarrow{a_2} (m_2, g_2') = (m_2, g_3) \dots \xrightarrow{a_n} (m_n, g_n')$$

Réciproquement, si $a_1 \dots a_n$ est un mot accepté par $\mathcal{B}$ sur le chemin

$$(m_0, g_0) \xrightarrow{a_1} (m_1, g_1) \xrightarrow{a_2} (m_2, g_2) \dots \xrightarrow{a_n} (m_n, g_n)$$

alors pour tout i, $(m_i, g_i, a_{i+1}, up_{m_i,a_{i+1}}, g_{i+1}, m_{i+1})$ est une transition de $\mathcal{A}$ et $g_i \xrightarrow{up_{m_i,a_{i+1}}} g_{i+1}$. Nous définissons $\theta_0 = \perp^k$ et inductivement θ_{i+1}, en utilisant la condition (†) par : comme $\overline{\theta_i} = g_i$, il existe d_{i+1} tel que $up_{m_i,a_{i+1}}(\theta_i, d_{i+1}) \in g_{i+1}$; nous définissons alors θ_{i+1} par $up_{m_i,a_{i+1}}(\theta_i, d_{i+1})$. Ainsi, le chemin suivant accepte le mot $(a_1, d_1) \dots (a_n, d_n)$ dans $\mathcal{A}$:

$$\begin{array}{ccccccccc} m_0 & \xrightarrow[d_1]{g_0, a_1, up_{m_0,a_1}, g_1} & m_1 & \xrightarrow[d_2]{g_1, a_2, up_{m_1,a_2}, g_2} & m_2 & \dots & \xrightarrow[d_n]{g_{n-1}, a_n, up_{m_{n-1},a_n}, g_n} & m_n \\ \theta_0 & & \theta_1 & & \theta_2 & & & \theta_n \end{array}$$

La preuve est maintenant complète : $\mathcal{B}$ accepte $\mathsf{Undata}(L)$.

Ainsi, L est vide si et seulement si $\mathsf{Undata}(L)$ est vide. Nous pouvons donc décider de la vacuité de L en appliquant l'algorithme non déterministe à l'automate que nous venons de construire. Comme l'automate a $|M \times \mathcal{D}^k/\!\sim|$ états, l'algorithme peut être implémenté dans un espace $\log(|M \times \mathcal{D}^k/\!\sim|)$, ce qui est polynomial dans la taille de l'entrée (constituée du monoïde et du mécanisme à k registres). □

Nous allons présenter deux exemples d'automates pour lesquels cette condition (†) est vérifiée.

***Exemple* 5.17** Revenons à l'automate que nous avons construit dans la preuve du lemme 5.15. Dans cette automate, la relation d'équivalence a quatre classes d'équivalence :

- $g_1 = \{(d_1, d_2, d_3) \in \mathcal{D}^3 \mid d_1 > 0 \text{ et } d_1 = d_2 - d_3\}$
- $g_2 = \{(d_1, d_2, d_3) \in \mathcal{D}^3 \mid d_1 > 0 \text{ et } d_1 = d_3 - d_2\}$
- $g_3 = \{(d_1, d_2, d_3) \in \mathcal{D}^3 \mid d_1 > 0 \text{ et } d_1 \notin \{d_2 - d_3, d_3 - d_2\}\}$
- $g_4 = \{(d_1, d_2, d_3) \in \mathcal{D}^3 \mid d_1 \leq 0\}$

Nous n'allons pas décrire tous les cas possibles, mais considérons la transition de q_1 à q_2 et la garde g_1. De g_1, il est possible d'aller dans g_2 avec la mise à jour $\{r_3\}$, il suffit de prendre $(1, 3, 2) \in g_1$ et, *via* la mise à jour sus-citée, nous pouvons aller dans $(1, 3, 4)$. Il est alors facile de voir que pour tout triplet (d_1, d_2, d_3) dans g_1, il sera possible de le faire. Tous les autres cas sont aussi simples.

L'exemple suivant est un peu plus difficile, mais est aussi plus général.

***Exemple* 5.18** Considérons la construction de la partie précédente qui permet de transformer un automate temporisé en un automate de données. Nous allons vérifier que si $\mathcal{A}$ est un automate temporisé, l'automate qui est construit vérifie la condition (†). Pour cela, considérons une transition

$$(q, f) \xrightarrow{\overline{\theta}, a, \alpha, \overline{\theta'}} (q', f')$$

de l'automate que l'on construit telle qu'il existe $\beta \in \overline{\theta}$ avec $\alpha(\beta, d) \in \overline{\theta'}$ (pour $d \in \mathcal{D}$). Prenons maintenant $\gamma \in \overline{\theta}$. Nous avons que $\beta \sim \gamma$, donc

$$(\beta_{f(x_0)} - \beta_i)_{1 \leq i \leq 2n} \equiv_2 (\gamma_{f(x_0)} - \gamma_i)_{1 \leq i \leq 2n}$$

Il existe alors un successeur de γ tel que

$$(d - \beta_i)_{1 \leq i \leq 2n} \equiv_2 (d' - \gamma_i)_{1 \leq i \leq 2n}$$

Il est alors évident que $\alpha(\gamma, d') \in \overline{\theta'}$. Ceci achève notre preuve du fait que l'automate que nous avons construit dans la partie précédente vérifie la condition (†).

5.3.2 Calcul de cette condition

En général, la condition (†) est indécidable. Néanmoins, comme c'est le cas pour les automates de données provenant des automates temporisés (voir l'exemple 5.18), dans certains cas, il va être possible de décider de la condition (†).

Pour cela, nous définissons :

$$\begin{cases} \widehat{up}(g) = \{v' \mid \exists v \in g,\ \exists d \in \mathcal{D},\ v' = up(v, d)\} \\ \widehat{up}^{-1}(g') = \{v \mid \exists d \in \mathcal{D},\ up(v, d) \in g'\} \end{cases}$$

La condition (†) peut alors se réécrire de la manière suivante :

$$(\dagger) \iff \left[\widehat{up}(g) \cap g' \neq \varnothing \implies \widehat{up}^{-1}(g') \cap g = g\right]$$

Ainsi, si les mises à jour up que nous considérons sont telles que :

- pour toute contrainte g, $\widehat{up}(g)$ et $\widehat{up}^{-1}(g)$ peuvent être calculés,
- l'intersection peut être calculée,
- le vide et l'inclusion peuvent être testés,

alors la condition (†) peut être décidée.

Les mises à jour que nous considérons ne permettent de faire aucun calcul et sont très simples. La condition (†) peut être encore simplifiée.

Si X est un sous-ensemble de $\{1 \dots k\}$, nous définissons π_X comme la projection sur l'ensemble X. Si up est une mise à jour (*i.e.* $up \subseteq \{1 \dots k\}$), nous définissons le produit $\times_{up}$ par : si R est un sous-ensemble de $\mathcal{D}^{k-|up|}$ et si R' est un sous-ensemble de $\mathcal{D}^{|up|}$,

$$R \times_{up} R' = \{v \in \mathcal{D}^k \mid \pi_{\overline{up}}(v) \in R \text{ et } \pi_{up}(v) \in R'\}$$

où $\overline{up} = \{1 \dots k\} \setminus up$. La condition (†) est alors équivalente à :

$$\left((\pi_{\overline{up}}(g) \times_{up} \mathcal{D}^{|up|}) \cap g' \neq \varnothing \right) \implies \pi_{\overline{up}}(g) \subseteq \pi_{\overline{up}}(g')$$

Donc, si la relation d'équivalence $\sim$ est définie de telle manière que :

1. $\pi_{\overline{up}}(g)$ peut être calculé,
2. $\pi_{\overline{up}}(g) \times_{up} \mathcal{D}^{|up|}$ peut être calculé,
3. $(\pi_{\overline{up}}(g) \times_{up} \mathcal{D}^{|up|}) \cap g' \neq \varnothing$ peut être décidé,
4. $\pi_{\overline{up}}(g) \subseteq \pi_{\overline{up}}(g')$ peut être décidé

alors la condition (†) peut être décidée.
Nous pouvons remarquer que toutes les opérations de la liste précédente sont des opérations élémentaires sur les classes d'équivalence de la relation $\sim$.

5.4 Extensions du modèle

5.4.1 Effacement de données et permutation des registres

La première extension du modèle que nous allons étudier consiste à étendre les mises à jour utilisées avec des possibilités d'effacement et de permutation de registres. Une *mise à jour étendue* est alors une fonction up qui met dans chaque registre soit la valeur courante, soit la valeur d'un autre registre, soit $\perp$, la donnée vide.

Proposition 5.19 *Les automates de données avec mises à jour étendues ont le même pouvoir d'expression que les automates de données classiques.*

Preuve : Soit $\mathcal{A} = (Q, k, \Sigma, \mathcal{D}, \sim, q_0, F, T)$ un automate de données avec mises à jour étendues. Nous notons $\mathcal{F}$ l'ensemble des fonctions $f : \{1, \dots, k\} \longrightarrow \{0, 1, \dots, k\}$. Intuitivement, f est une fonction de représentation, dans le sens où la valeur théorique du registre i sera en fait stockée dans le registre j. Le registre 0 est un registre particulier qui n'est jamais mis à jour et qui contient donc toujours la valeur $\perp$. Construisons l'automate $\mathcal{B} = (Q', k, \Sigma, \mathcal{D}, \equiv, q_0', F', T')$ de la manière suivante :

- $Q' = Q \times \mathcal{F}$,
- $q_0' = (q_0, \mathsf{Id})$, où $\mathsf{Id}(i) = i$ pour tout registre i,
- $F' = F \times \mathcal{F}$,
- $\equiv$ est défini par :

$$\theta \equiv \theta' \iff \forall f \in \mathcal{F},\ (\theta_{f(i)})_{i=1\dots k} \sim (\theta'_{f(i)})_{i=1\dots k}$$

- si $(q, g, a, up, g', q') \in T$, alors pour tout f dans $\mathcal{F}$, $((q, f), \overline{g}, a, up', \overline{g'}, (q', f'))$ est dans T' si
  - $(\theta_{f(i)})_{1 \leq i \leq k} \in \overline{g} \implies (\theta_i)_{0 \leq i \leq k} \in g$
  - $(\theta_{f(i)})_{1 \leq i \leq k} \in \overline{g'} \implies (\theta_i)_{0 \leq i \leq k} \in g'$
  - $f'(i) = f(j)$ si up met la valeur du registre j dans le registre i. Notons I l'ensemble de ces registres (l'ensemble des i tels que $f'(i)$ soit déjà défini). Si I est de cardinal k, alors f' est totalement définie. Sinon, il reste au moins un registre à part 0, disons h, qui n'est pas dans $f'(I)$. Pour tout registre i tel que up affecte à i la valeur courante de la donnée, nous posons $f'(i) = h$. Pour tout registre i tel que up affecte à i la valeur $\perp$, nous posons $f'(i) = 0$.
  - up' met la valeur courante dans le registre h (s'il a été défini auparavant).

À partir de cette construction, il est facile de montrer que le langage de données accepté par $\mathcal{A}$ est le même que celui accepté par $\mathcal{B}$. $\square$

L'effacement de données et la permutation de registres ne sont donc que des macros qui ne rajoutent pas d'expressivité au modèle. Cependant, ces macros peuvent être très utiles. Par exemple, avec elles, la preuve de la proposition 5.24 ca être nettement simplifiée.

La preuve précédente montre aussi que nous pourrions restreindre nos mises à jour à la mise à jour d'un seul registre sur chaque transition.

5.4.2 En utilisant des mises à jour plus générales

Les mises à jour utilisées dans notre modèle sont très simples. Nous pouvons juste « écrire » une donnée lue dans un ou plusieurs registres. Nous ne pouvons faire aucun calcul préalable sur les données avant de les mettre en mémoire dans les registres. Une question naturelle est de savoir ce que tout cela devient si jamais nous autorisons certains calculs sur les registres. Dans cette partie, une *mise à jour étendue* est une fonction $up : \mathcal{D}^k \times \mathcal{D} \longrightarrow \mathcal{D}^k$.

Dans notre étude précédente, nous avons montré que le monoïde joue un rôle fondamental dans le mécanisme de reconnaissance (voir la proposition 5.6). Nous allons voir que si nous autorisons des mises à jour étendues, ceci n'est plus vrai : le monoïde ne joue plus aucun rôle.

Proposition 5.20 *Soit L un langage formel sur l'alphabet Σ. Supposons que L soit reconnu par le monoïde fini M. Alors le langage de données $L_M = \{(a_1, m_1) \ldots (a_n, m_n) \mid a_1 \ldots a_n \in L\}$ sur l'alphabet Σ avec comme ensemble de données M est reconnu par le monoïde $N = \{1, x, y\}$ où $zx = x$ et $zy = y$ pour tout z dans N.*

Preuve : Nous supposons que $L \subseteq \Sigma^*$ est reconnu par le monoïde fini M. Il existe un morphisme $\varphi : \Sigma^* \longrightarrow M$, un sous-ensemble $P \subseteq M$ tels que $L = \varphi^{-1}(P)$. Nous définissons maintenant $k = 1$ (il n'y a qu'un seul registre) et $\mathcal{D} = M$. Alors, pour tout $z \in N$, pour tout $a \in \Sigma$, nous définissons $up_{z,a} : M \times M \longrightarrow M$ par $up_{z,a}(m, d) = m\varphi(a)$. Nous définissons aussi le morphisme $\psi : (\Sigma \times M)^* \longrightarrow N$ par :

$$\psi(a, m) = \begin{cases} x & \text{si } m \in P \\ y & \text{si } m \in M \setminus P \end{cases}$$

Alors, en utilisant cette construction, nous pouvons montrer que le monoïde fini N reconnaît le langage de données L_M. $\square$

Remarque : Nous pouvons remarquer qu'autoriser des mises à jour étendues élargit la classe des langages de données qui sont reconnus par un monoïde fini. Par exemple, considérons le langage

$$\{(a, p_1) \ldots (a, p_n) \mid \forall i,\ p_i \text{ nombre premier et } i \neq j \implies p_i \neq p_j\}$$

sur l'alphabet Σ avec comme ensemble de données $\mathbb{N}$. Ce langage de données n'est pas reconnaissable (par les automates de données classiques), mais est reconnu par un monoïde fini en utilisant les mises à jour étendues, notamment la mise à jour up telle que $up((\theta_1, \theta_2), d) = (d, d.\theta_2)$.
Cependant, même avec les mises à jour étendues, les résultats d'équivalence entre monoïdes et automates et de décidabilité sont toujours vrais car ces résultats ne dépendent pas des mises à jour utilisées.

5.5 Le modèle non déterministe

Dans ce qui précède, nous avons étudié le modèle des automates de données déterministes. Cependant, nous avons aussi défini un modèle d'automates non déterministes (voir la définition 5.9 à la page 115). Quelles sont les propriétés de ces automates ? Nous définissons un mécanisme non déterministe à k registres par un triplet $[(U_{m,a})_{m\in M, a\in\Sigma}, \sim, \varphi]$ où la seule différence avec un mécanisme déterministe réside dans la définition des mises à jour. Ici, $U_{m,a}$ est un ensemble de mises à jour de k registres au lieu d'être une seule mise à jour. Nous disons donc qu'un monoïde fini M reconnaît de manière non déterministe un langage de données L s'il existe un mécanisme non déterministe à k registres qui le reconnaît. Certaines propriétés, vraies pour le modèle déterministe restent vraies pour la version non déterministe. En particulier,

Proposition 5.21 *Soit L un langage de données sur l'alphabet Σ avec comme ensemble de données $\mathcal{D}$. Alors,*

– *L est reconnu de manière non déterministe par un monoïde fini si et seulement si il est reconnu par un automate de données (non déterministe).*

Nous disons alors que le langage L est nd-reconnaissable.

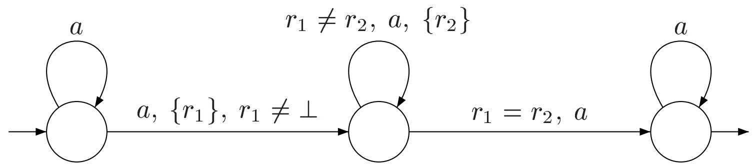
– *La condition (†) assure aussi la décidabilité du problème du vide, i.e. si L est reconnu par un automate de données qui vérifie la condition (†) alors il est possible de tester le vide du langage accepté.*

Nous ne proposons pas de preuve pour ces résultats, les preuves précédentes s'adaptant parfaitement au cas non déterministe. Par contre, nous avons la propriété suivante :

Proposition 5.22 *La classe des langages de données nd-reconnaissables est strictement plus expressive que la classe des langages de données reconnaissables.*

Preuve : Nous proposons un langage de données qui est nd-reconnaissable mais qui n'est pas reconnaissable. Ceci montrera la proposition ci-dessus.

Considérons le langage de données L reconnu par l'automate de données non déterministe suivant : Ce langage s'exprime par $L = \{(a, d_1)\ldots(a, d_n) \mid \exists 1 \leq i < j < n,\ d_i = d_j \neq \perp\}$. Nous



allons montrer que ce langage n'est reconnu par aucun automate de données (déterministe).

Supposons que L soit accepté par l'automate de données à k registres $\mathcal{A}$. Il y a un nombre fini de chemins de longueur $k+1$ dans $\mathcal{A}$. Pour chaque chemin c, nous définissons

$$E(c) = \{(a, d_1)\ldots(a, d_{k+1})\text{est lu sur le chemin } c \text{ et } i \neq j \implies d_i \neq d_j\}$$

Il existe un entier $n(c)$ tel que la donnée indexée par $n(c)$ (d'un mot de données quelconque lu sur le chemin c) n'est pas retenue en mémoire dans un registre à la fin du chemin c. En utilisant un argument combinatoire similaire à celui développé dans la preuve de la proposition 5.7, il existe deux mots de données dans un certain $E(c)$ qui ne diffèrent que de la donnée indexée par $n(c)$.

$$\begin{array}{cccccccccc}
q_0 & \xrightarrow{(a,d_1)} & q_1 & \dots & q_{n(c)-1} & \xrightarrow{(a,d_{n(c)})} & q_{n(c)} & \dots & q_k & \xrightarrow{(a,d_{k+1})} & q_{k+1} \\
\theta_0 & & \theta_1 & & \theta_{n(c)-1} & & \theta_{n(c)} & & \theta_k & & \theta_{k+1} \\
= & & = & & = & & \sim & & \sim & & = \\
\theta'_0 & & \theta'_1 & & \theta'_{n(c)-1} & & \theta'_{n(c)} & & \theta'_k & & \theta'_{k+1} \\
q_0 & \xrightarrow{(a,d_1)} & q_1 & \dots & q_{n(c)-1} & \xrightarrow{(a,d'_{n(c)})} & q_{n(c)} & \dots & q_k & \xrightarrow{(a,d_{k+1})} & q_{k+1}
\end{array}$$

Comme $\theta_k = \theta'_k$, pour tout mot de données w, $(a,d_1)\dots(a,d_{n(c)})\dots(a,d_{k+1})$ est dans L si et seulement si $(a,d_1)\dots(a,d'_{n(c)})\dots(a,d_{k+1})$ est dans L. Bien sûr, ceci n'est pas vrai. Ainsi, L n'est accepté par aucun automate de données (déterministe). □

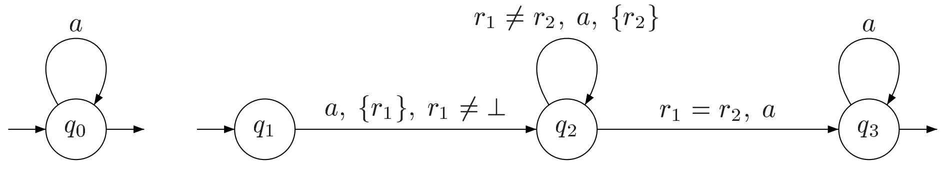
Une conséquence simple de la construction faite dans la preuve ci-dessus est le résultat suivant :

Corollaire 5.23 *La classe des langages de données reconnaissables n'est pas close par concaténation.*

PREUVE : Considérons le langage de données précédent L. Bien que non reconnaissable, ce langage résulte de la concaténation des deux langages reconnaissables suivants :

$$\{(a,d_1)\dots(a,d_p) \mid d_i \in \mathcal{D}\} \ \text{ et } \ \{(a,d_0)\dots(a,d_n) \mid \exists 1 \leq j < n,\ d_j = d_0\}.$$

Ces langages sont reconnus par les automates de données suivants : Ceci conclut le fait que la



classe des langages de données reconnaissables n'est pas close par concaténation. □

Proposition 5.24 *La classe des langages de données nd-reconnaissables est close par union finie, intersection finie, concaténation et itération finie. Elle n'est pas close par passage au complémentaire.*

PREUVE : En ce qui concerne l'union et l'intersection, les constructions classiques suffisent. Soient L_1 et L_2 des langages de données acceptés respectivement par des automates ayant k_1 et k_2 registres. Nous allons montrer que $L_1 \cdot L_2$ est accepté par un automate de données avec $k = \max(k_1,k_2)$ registres. Nous utilisons la proposition 5.19. Supposons que pour $i = 1,2$, $\mathcal{A}_i = (Q_i, k_i, \Sigma, \mathcal{D}, \sim_i, q_{0,i}, F_i, T_i)$. Nous construisons l'automate $\mathcal{A} = (Q, k, \Sigma, \mathcal{D}, \sim, q_0, F, T)$ où :
– $Q = Q_1 \cup Q_2$
– $\theta \sim \theta'$ si $\theta_{|1\dots l_i} \sim_i \theta'_{|1\dots l_i}$ pour $i = 1,2$
– $q_0 = q_{0,1}$
– $F = \begin{cases} F_2 & \text{si } I_2 \cap F_2 = \varnothing \\ F_1 \cup F_2 & \text{si } I_2 \cap F_2 \neq \varnothing \end{cases}$
– $q \xrightarrow{g,a,up,g'} q' \in T \iff \begin{cases} \text{soit } q \xrightarrow{G,a,up,G'} q' \in T_1 \text{ avec } G \implies g \text{ et } G' \implies g' \\ \text{soit } q \xrightarrow{G,a,up,G'} q' \in T_2 \text{ avec } G \implies g \text{ et } G' \implies g' \\ \text{soit } q \in F_1 \text{ et } \exists i \xrightarrow{\bot^{k_2},a,\overline{up},G'} q' \in T_2 \text{ avec } i \in I_2,\ G' \implies g' \\ \qquad \text{et } \overline{up} \text{ met la donnée courante dans les registres de } up \\ \qquad \text{et met } \bot \text{ dans les autres registres}^2 \end{cases}$

[2]Nous utilisons là le résultat de la proposition 5.19.

L'automate de données $\mathcal{A}$ reconnaît le langage de données $L(\mathcal{A}_1) \cdot L(\mathcal{A}_2)$: supposons que le mot de données w soit dans $L(\mathcal{A}_1) \cdot L(\mathcal{A}_2)$, $w = uv$ où $u \in L(\mathcal{A}_1)$ et $v \in L(\mathcal{A}_2)$ et considérons les exécutions sur u dans $\mathcal{A}_1$ et sur v dans $\mathcal{A}_2$ respectivement :

$$\begin{array}{ccccccccccc} q_0 & \longrightarrow & \dots & q_{n-1} & \xrightarrow[d_n]{g_n, a_n, up_n, g'_n} & q_n & \text{et} & q'_0 & \xrightarrow[d'_1]{g_1, a'_1, up_1, g'_1} & q'_1 & \dots \\ \theta_0 & & & \theta_{n-1} & & \theta_n & & \theta'_0 & & \theta'_1 & \end{array}$$

Dans $\mathcal{A}$, leur concaténation correspond au chemin suivant :

$$\begin{array}{ccccccccc} q_0 & \longrightarrow & \dots & q_{n-1} & \xrightarrow[d_n]{G_n, a_n, up_n, G'_n} & q_n & \xrightarrow[d'_1]{G'_n, a'_1, \overline{up_1}, G'_1} & q'_1 & \dots \\ \theta_0 & & & \theta_{n-1} & & \theta_n & & \theta'_1 & \end{array}$$

où pour tout i, $g_i \implies G_i$ et $g'_i \implies G'_i$ et $\overline{up_1}$ est défini comme dans T. C'est un chemin acceptant pour w dans $\mathcal{A}$. Ainsi, $L(\mathcal{A}_1) \cdot L(\mathcal{A}_2) \subseteq L(\mathcal{A})$.
Réciproquement, si $w \in L(\mathcal{A})$, supposons que w peut être lu dans $\mathcal{A}$ sur le chemin suivant :

$$\begin{array}{ccccccccc} q_0 & \longrightarrow & \dots & q_{n-1} & \xrightarrow[d_n]{G_n, a_n, up_n, G'_n} & q_n & \xrightarrow[d'_1]{G'_n, a'_1, \overline{up_1}, G'_1} & q'_1 & \dots \\ \theta_0 & & & \theta_{n-1} & & \theta_n & & \theta'_1 & \end{array}$$

Ce chemin peut être découpé en deux parties : une dans $\mathcal{A}_1$ et une autre dans $\mathcal{A}_2$:

$$\begin{array}{ccccccccccc} q_0 & \longrightarrow & \dots & q_{n-1} & \xrightarrow[d_n]{g_n, a_n, up_n, g'_n} & q_n & \text{et} & q'_0 & \xrightarrow[d'_1]{g_1, a'_1, up_1, g'_1} & q'_1 & \dots \\ \theta_0 & & & \theta_{n-1} & & \theta_n & & \theta'_0 & & \theta'_1 & \end{array}$$

Ces chemins sont des chemins acceptants pour respectivement u et v tels que $|u| = n$ et $w = uv$. Ainsi, $u \in L(\mathcal{A}_1)$ et $v \in L(\mathcal{A}_2)$ et donc $w \in L(\mathcal{A}_1) \cdot L(\mathcal{A}_2)$.
La preuve est maintenant complète : $\mathcal{A}$ reconnaît la concaténation $L(\mathcal{A}_1) \cdot L(\mathcal{A}_2)$.

Une construction similaire à celle que nous venons de faire permet de traiter l'itération, car, dans la construction précédente, nous n'augmentons pas le nombre de registres utilisés.

Finalement, le langage L que nous avons considéré dans la preuve de la proposition 5.21 est nd-reconnaissable. Nous pouvons montrer que le complémentaire de ce langage, c'est-à-dire le langage

$$\overline{L} = \{(a, d_1) \dots (a, d_n) \mid \forall 1 \leq i < j < n,\ d_i \neq d_j\}$$

n'est reconnu par aucun monoïde fini (bien sûr si $\mathcal{D}$ est infini). La preuve utilise des arguments similaires à ceux des preuves des propositions 5.7 et 5.21. □

5.6 Langages de données et langages sur un alphabet infini

À notre connaissance, aucun travail n'avait été fait pour mettre en relation l'algèbre et les langages temporisés. En cherchant un formalisme algébrique pour décrire les langages temporisés, nous avons été amenés à nous placer dans un cadre plus général, les langages de données. Il apparaît alors que ces langages de données sont des langages sur des alphabets infinis : si nous oublions la structure interne de $\Gamma = \Sigma \times \mathcal{D}$, Γ peut être vu comme un alphabet infini. Il est alors naturel de rapprocher notre étude de celles sur les langages sur des alphabets infinis. Nous allons alors considérer plusieurs travaux qui peuvent être mis en relation avec les nôtres.

5.6.1 Les travaux de Autebert, Beauquier et Boasson

Dans [ABB80], plusieurs notions de langages reconnaissables (et algébriques) sont étudiées. Après avoir mis en avant que la solution de l'image inverse par un morphisme d'une sous-partie d'un monoïde fini n'était pas une solution acceptable, ils ont par exemple étudié la notion de langage *H-rationnel*. Un langage L sur un alphabet infini $\mathcal{D}$ est dit *H-rationnel* si pour tout alphabet fini X, pour tout morphisme alphabétique $\varphi : \mathcal{D}^* \longrightarrow X^*$, $\varphi(L)$ est reconnaissable. Ils ont aussi proposé une définition similaire de langage *H-algébrique*. Ils ont ensuite étudié la clôture de ces classes par les opérations habituelles puis ont étudié l'adaptation des théorèmes classiques de la théorie des langages à leurs classes de langages.

Par contre, ils n'ont proposé aucun formalisme d'automate permettant d'exprimer les mêmes langages qu'eux. Leurs travaux et les nôtres sont difficilement comparables, même s'il est facile de voir que tout langage L qui est H-reconnaissable est aussi reconnaissable en tant que langage de données. En effet, considérons un langage L sur l'alphabet $\mathcal{D}$ qui est H-reconnaissable et considérons aussi un alphabet fini X. Soit φ un morphisme alphabétique $\varphi : \mathcal{D}^* \longrightarrow X^*$. Il existe un monoïde M, une sous-partie $P \subseteq M$ et un morphisme $\psi : X^* \longrightarrow M$ tels que $\varphi(L) = \psi^{-1}(P)$. Nous prenons un seul registre et nous définissons la relation d'équivalence sur $\mathcal{D}$ par $d \sim d' \iff \varphi(d) = \varphi(d')$. Alors il est facile de voir que le mécanisme à un registre $\rho = [(up_m)_{m \in M}, \sim, \psi \circ \varphi]$ avec $up_m = \{r_1\}$ pour tout m dans M, reconnaît le langage L en tant que langage de données.

5.6.2 Les travaux de Kaminski et Francez

Dans [KF94], des *automates avec registres* ont aussi été définis. Cette classe d'automates est close par intersection, union, concaténation et étoile, elle n'est pas close par passage au complémentaire. De plus, la restriction de leur modèle aux alphabets finis correspond aux automates finis. Nous constatons aussi que les mises à jour des registres autorisées dans le modèle correspondent aux nôtres, dans le sens où aucun calcul n'est autorisé et que la machine peut juste « stocker » la donnée courante dans un des registres.

Cependant, ce modèle est vraiment très restrictif, les contraintes sur les registres qui peuvent être testées consistent uniquement à tester si la valeur courante est la même que celle présente dans un registre particulier. Ceci entraîne la propriété que, étant données des valeurs initiales stockées dans les registres, pour tout automorphisme ι de l'ensemble de ces valeurs, si u est un mot accepté par l'automate, alors $\iota(u)$ est aussi accepté par l'automate. Ceci signifie que la « valeur » des lettres lues importe peu, mais que c'est plus le nombre de fois qu'elles interviennent qui est important.

Enfin, ils proposent uniquement un modèle d'automate pour leurs langages sur des alphabets, mais ils ne proposent pas d'autres formalismes algébrique ou logique.

5.6.3 Les travaux de Neven, Schwentick et Vianu

Dans l'article [NSV01], l'étude des automates avec registres définis dans [KF94] a été poursuivie, surtout en termes logiques. Les auteurs ont aussi étudié une nouvelle classe d'automates, appelés *automates à galets*, cette classe leur semble plus adaptée que celle des automates avec registres pour reconnaître les mots sur un alphabet infini. Dans les automates à galets, au lieu de stocker certaines lettres lues dans les registres, il est possible de marquer un nombre borné de lettres du mot qu'on est en train de lire. Les transitions sont contraintes par des conditions sur les lettres qui sont marquées de la sorte. Les logiques qu'ils ont étudiées sont des logiques du premier et du second ordre auxquelles ont été rajoutés des prédicats qui permettent de tester l'égalité de deux lettres du mot ou l'égalité d'une lettre du mot et d'une lettre fixée. Cependant, l'étude logique n'a pas permis de dégager un langage logique adapté aux automates avec registres, ni même aux automates à galets.

5.7 Conclusion

Dans ce chapitre, nous avons proposé une notion de reconnaissance par monoïde pour les langages de données, une notion plus générale que les langages temporisés. Nous avons aussi proposé une notion d'automates de données dont la version déterministe est équivalente au formalisme algébrique de reconnaissance par monoïde. Ce résultat, ainsi que sa preuve, sont très proches de ce qui existe dans le cadre des langages formels. Comme instance de nos résultats, nous obtenons une *caractérisation purement algébrique* des langages temporisés.

Cette théorie doit maintenant être développée. Par exemple, une notion de langage apériodique pourrait être définie de manière naturelle et aurait à être étudiée de manière précise.

En se restreignant au cadre temporisé, nous avons encore à étudier les relations exactes entre nos automates et les divers modèles temporisés existants, par exemple les automates de [HRS98] et ceux de [DZ98].

Si l'ensemble de données considéré est fini, les langages de données correspondants peuvent aussi être considérés comme des langages formels. Il serait intéressant de comparer la famille de langages de données reconnus par un monoïde fixé avec celle des langages formels reconnus par ce même monoïde.

Enfin, il serait aussi intéressant d'étudier de manière plus fine les relations entre le monoïde et les mises à jour des registres. Ici, nous avons étudié les deux extrémités du spectre : si les registres ne peuvent faire aucun calcul, alors le monoïde joue un rôle fondamental (proposition 5.6). Inversement, si les registres peuvent faire d'importants calculs, le monoïde ne joue plus aucun rôle (proposition 5.20). Tous les cas intermédiaires restent à étudier.

Pour compléter ce panorama déjà assez engageant, nous présenterons au chapitre suivant un formalisme logique pour ces langages de données.

Chapitre 6

Logique et automates de données

Ce chapitre provient de l'article [Bou02].

Les logiques sont des moyens naturels pour représenter les propriétés. Parmi ces logiques, nous pouvons citer les logiques temporelles comme LTL [Pnu77] ou CTL [CES86], mais aussi les logiques provenant de l'arithmétique comme les logiques du premier et du second ordre (voir par exemple [Pin94, PP01]).

Dans le cadre des langages formels, la *logique monadique du second ordre* $\mathsf{MSO}(<)$ (appelée aussi le *calcul séquentiel de Büchi*) permet de décrire exactement tous les langages acceptés par des automates finis [Büc62] (voir [Pin94, PP01, Wei99] pour une présentation de ce résultat). Par exemple, si l'alphabet que nous considérons est A, la formule

$$\exists x \exists y \Big((x < y) \wedge R_a x \wedge R_b y \Big)$$

décrit le langage $A^*aA^*bA^*$. Des sous-classes de $\mathsf{MSO}(<)$ permettent aussi de décrire des familles intéressantes de langages, comme la *logique du premier ordre* qui permet de décrire les langages sans étoile [MP71, Pin94, PP01] ou les langages définis par les formules de LTL [Kam68].

Dans le cadre temporisé, plusieurs travaux ont été effectués pour mettre en relation les automates temporisés et certains logiques. Par exemple, Thomas Wilke a proposé une logique monadique du second ordre avec distances qui permet d'exprimer tous les langages reconnus par des automates temporisés [Wil94]. Dans leur article [HRS98], Thomas Henzinger, Jean-François Raskin et Pierre-Yves Schobbens ont dégagé plusieurs logiques qui caractérisent des sous-classes des automates temporisés qui ont comme avantage d'être, suivant leur terminologie, « totalement décidables » (« *fully decidable* »), ce qui signifie que les logiques qu'ils proposent sont closes pour toutes les opérations booléennes et leur satisfaisabilité est décidable. Les classes de systèmes temporisés qu'ils caractérisent de la sorte sont des sous-classes des automates « à horloges d'évènements » (« *event-clock automata* ») définis dans [AFH94].

Nous proposons ici un langage logique pour caractériser exactement les langages de données qui sont acceptés par les automates de données (pas forcément déterministes). Le langage que nous proposons est plus proche de celui de Thomas Wilke que des logiques « totalement décidables » de Henzinger, Raskin et Schobbens. Après avoir défini notre langage, nous montrons son équivalence avec celle des automates de données. Nous énonçons ensuite une condition sous laquelle nous pouvons décider de la satisfaisabilité des formules de notre langage. L'ensemble des formules qui satisfont notre condition est strictement plus expressif que le langage monadique avec distances défini dans [Wil94]. Nous obtenons donc une logique dont un large fragment est décidable, mais qui n'est pas « fully decidable » (les formules ne sont pas closes par la négation).

6.1 Description de la logique

Dans tout ce travail, nous supposerons donnés un alphabet fini Σ et un ensemble de données $\mathcal{D}$.

Notre but est de proposer un langage logique équivalent au modèle des automates de données (non déterministes). Nous commençons par considérer le langage logique $\mathcal{L}_c$ formé de toutes les formules closes $\varphi ::= \exists X_1 \dots \exists X_k.\psi$ où les $(X_i)_{1 \leq i \leq k}$ sont des variables du second ordre et ψ est définie inductivement par la grammaire

$$\psi ::= x < y \mid x = y \mid \exists x.\psi \mid \forall x.\psi \mid R_a x \mid Xx \mid Q_g(x_1, \dots, x_\ell) \mid \neg\psi \mid \psi \wedge \psi \mid \psi \vee \psi$$

où g est une contrainte de $\mathcal{D}^\ell$, $a \in \Sigma$, X est une variable du second ordre et $x, y, x_1, \dots, x_\ell$ sont des variables du premier ordre. Ce langage logique est une logique du second ordre à laquelle des prédicats de données $Q_g(x_1, \dots, x_\ell)$ ont été rajoutés. De manière informelle, ces prédicats permettent de tester si les données représentées par les variables du premier ordre $x_1, \dots, x_\ell$ vérifient ou pas la contrainte g.

6.1.1 Interprétation de $\mathcal{L}_c$

Nous définissons formellement la sémantique de $\mathcal{L}_c$.

La logique $\mathcal{L}_c$ a pour signature $\mathcal{S} = \{<, (R_a)_{a\in\Sigma}, (Q_g)_{g \text{ contrainte de } \mathcal{D}^\ell}\}$. Chaque mot de données $u = (a_1, d_1) \dots (a_{|u|}, d_{|u|})$ est alors vu comme une structure

$$u^{\mathcal{S}} = (\mathsf{Dom}(u), <, (\iota_u(R_a))_{a\in\Sigma}, (\iota_u(Q_g))_{g \text{ contrainte de } \mathcal{D}^\ell})$$

où :

- $\mathsf{Dom}(u) = \{1, \dots, |u|\}$ est le domaine,
- le prédicat $<$ est interprété par l'ordre naturel sur $\mathsf{Dom}(u)$,
- pour $a \in \Sigma$, $\iota_u(R_a) = \{i \in \mathsf{Dom}(u) \mid a_i = a\}$,
- pour g contrainte de $\mathcal{D}^\ell$, $\iota_u(Q_g) = \{(i_1, \dots i_\ell) \in \mathsf{Dom}(u)^\ell \mid (d_{i_1}, \dots, d_{i_\ell}) \in g\}$.

Soit φ une formule de $\mathcal{L}_c$. Soit v l'ensemble de ses variables du premier ordre et V l'ensemble de ses variables du second ordre. Si D est un domaine, une valuation des variables du premier ordre est une application $\mu : v \longrightarrow D$ alors qu'une valuation des variables du second ordre est une application $\nu : V \longrightarrow 2^D$. Si μ est une valuation des variables du premier ordre, nous notons $\mu[x/i]$ la valuation qui associe à x l'entier i et à toute autre variable la même valeur que μ. La notation $\nu[X/\alpha]$ est similaire pour les variables du second ordre (sauf que α est un ensemble d'entiers).

Si u est un mot de données, si $u^{\mathcal{S}}$ est sa structure associée, si φ est une formule, si μ et ν sont des valuations pour φ (avec comme domaine $\mathsf{Dom}(u)$), nous disons que la formule φ s'interprète sur la structure $u^{\mathcal{S}}$ avec les valuations μ et ν, ce que nous notons $(u^{\mathcal{S}}, \mu.\nu) \models \varphi$, si les conditions du tableau 6.1 sont vérifiées.

Dans toute la suite, par abus de notation, nous noterons $(u, \mu.\nu) \models \varphi$ au lieu de $(u^{\mathcal{S}}, \mu.\nu) \models \varphi$. Nous remarquons aussi que la validité de $(u^{\mathcal{S}}, \mu.\nu) \models \varphi$ ne dépend que des valeurs assignées aux variables libres de φ par μ et ν. Donc, si φ est une formule close, nous nous autoriserons à noter $u \models \varphi$ au lieu de $(u, \mu.\nu) \models \varphi$ car la validité de la formule précédente ne dépend ni de μ ni de ν. Un langage de données L est l'ensemble des *modèles* d'une formule close φ si

$$L = \{u \in (\Sigma \times \mathcal{D})^* \mid u \models \varphi\}.$$

$(u^{\mathcal{S}};\mu.\nu) \models x < y$	si	$\mu(x) < \mu(y)$
$(u^{\mathcal{S}};\mu.\nu) \models x = y$	si	$\mu(x) = \mu(y)$
$(u^{\mathcal{S}};\mu.\nu) \models \exists x.\psi$	si	il existe $1 \leq i \leq n,\ (u^{\mathcal{S}};\mu[x/i].\nu) \models \psi$
$(u^{\mathcal{S}};\mu.\nu) \models \forall x.\psi$	si	pour tout $1 \leq i \leq n,\ (u^{\mathcal{S}};\mu[x/i].\nu) \models \psi$
$(u^{\mathcal{S}};\mu.\nu) \models R_a x$	si	$\mu(x) \in \iota_u(R_a)$
$(u^{\mathcal{S}};\mu.\nu) \models Xx$	si	$\mu(x) \in \nu(X)$
$(u^{\mathcal{S}};\mu.\nu) \models g(x_1,\ldots,x_l)$	si	$(\mu(x_1),\ldots,\mu(x_\ell)) \in \iota_u(g)$
$(u^{\mathcal{S}};\mu.\nu) \models \neg\psi$	si	$(u^{\mathcal{S}};\mu.\nu) \not\models \psi$
$(u^{\mathcal{S}};\mu.\nu) \models \psi_1 \wedge \psi_2$	si	$(u^{\mathcal{S}};\mu.\nu) \models \psi_1$ et $(u^{\mathcal{S}};\mu.\nu) \models \psi_2$
$(u^{\mathcal{S}};\mu.\nu) \models \psi_1 \vee \psi_2$	si	$(u^{\mathcal{S}};\mu.\nu) \models \psi_1$ ou $(u^{\mathcal{S}};\mu.\nu) \models \psi_2$
$(u^{\mathcal{S}};\mu.\nu) \models \exists X.\psi$	si	il existe $\alpha \subseteq \{1,\ldots,\lvert u\rvert\}$ tel que $(u^{\mathcal{S}};\mu.\nu[X/\alpha]) \models \psi$

Tableau 6.1: Relation de satisfaction pour $\mathcal{L}_c$

6.1.2 Quelques exemples

***Exemple* 6.1** Le langage de données

$$\{(a_1,d_1)\ldots(a_n,d_n) \mid n \in \mathbb{N} \text{ et } \exists 1 \leq i < j \leq n,\ (a_i,d_i) = (a_j,d_j)\}$$

peut être décrit par la formule qui appartient au langage $\mathcal{L}_c$

$$\exists x.\exists y. \left(x < y \wedge \bigvee_{a\in\Sigma} (R_a x \wedge R_a y) \wedge Q_g(x,y) \right)$$

où $g = \{(d,d') \mid d = d'\}$ est la diagonale de $\mathcal{D}^2$.

***Exemple* 6.2** Le complémentaire du langage précédent, c'est-à-dire le langage de données

$$\{(a_1,d_1)\ldots(a_n,d_n) \mid n \in \mathbb{N} \text{ et } \forall 1 \leq i < j \leq n,\ (a_i,d_i) \neq (a_j,d_j)\}$$

peut être décrit par la formule de $\mathcal{L}_c$ suivante :

$$\forall x.\forall y. \left(x < y \implies Q_{g_{\neq}}(x,y) \vee \bigwedge_{a\in\Sigma} (\neg R_a x \vee \neg R_a y) \right)$$

où $g_{\neq} = \{(d,d') \mid d \neq d'\}$.

Cependant, ce langage n'est accepté par aucun automate de données. Donc, même si $\mathcal{L}_c$ apparaît comme un bon candidat pour exprimer les langages de données, l'exemple précédent montre qu'il est trop expressif pour être équivalent au modèle des automates de données. Nous allons donc restreindre ce langage.

6.1.3 Définition de notre langage logique

La restriction du langage que nous allons considérer est similaire à la restriction qu'avait dû faire Thomas Wilke [Wil94] pour les automates temporisés.

Nous proposons de définir le langage logique $\mathcal{L}$ comme étant l'ensemble des formules closes $\exists X_1 \ldots \exists X_k.\psi$ où les $(X_i)_{1\leq i\leq k}$ sont des variables du second ordre et ψ est défini inductivement par la grammaire :

$$\psi ::= x < y \mid x = y \mid \exists x.\psi \mid \forall x.\psi \mid R_a x \mid Xx \mid P_g(X_1,\ldots,X_\ell,x) \mid \neg\psi \mid \psi \wedge \psi \mid \psi \vee \psi$$

où g est une contrainte définie sur $\mathcal{D}^\ell$ et $a \in \Sigma$.

La seule différence avec le langage $\mathcal{L}_c$ est contenue dans le prédicat $P_g(X_1, \ldots, X_\ell, x)$. Intuitivement, la variable X_i représente l'ensemble des dates où le registre r_i est mis à jour ; à la date représentée par la variable x, les données contenues dans les registres $(r_i)_{i=1\ldots\ell}$ vérifient la contrainte g.

Soit $u = (a_1, d_1) \ldots (a_n, d_n)$ un mot de données, soient μ un assignation des variables du premier ordre et ν une assignation des variables du second ordre, alors

$$(u; \mu.\nu) \models P_g(X_1, \ldots, X_\ell, x) \quad \Longleftrightarrow \quad \left(d^{(\mu,\nu)}_{\max(X_1 \leq x)}, \ldots, d^{(\mu,\nu)}_{\max(X_\ell \leq x)}\right) \in g$$

où pour tout $1 \leq i \leq \ell$

$$d^{(\mu,\nu)}_{\max(X_i \leq x)} = \begin{cases} d_\lambda \text{ si } \lambda \in \nu(X_i) \text{ et } \forall j,\ (j \in \nu(X_i) \text{ et } j \leq \mu(x) \text{ implique } j \leq \lambda) \\ \perp \text{ si } \forall j,\ j \in \nu(X_i) \text{ implique } \mu(x) < j \end{cases}$$

***Exemple* 6.3** Le langage $\{(a, \tau) \ldots (a, n\tau) \mid \tau > 0 \text{ et } n \text{ entier}\}$ peut être décrit par la formule

$$\begin{array}{ll} \exists X_0 \exists X_1 \exists X_2. & [\exists x_0. X_0 x_0 \wedge \forall y.((X_0 y \implies y = x_0) \wedge y \geq x_0) \wedge X_2 x_0 \wedge P_{g_0}(X_0, x_0)] \\ & \wedge\ \forall y \forall x. \left[\begin{array}{l} (X_1 x \wedge y = x+1) \implies (X_2 y \wedge \neg X_1 y \wedge P_g(X_0, X_1, X_2, y)) \\ \wedge\ (X_2 x \wedge y = x+1) \implies (X_1 y \wedge \neg X_2 y \wedge P_g(X_0, X_2, X_1, y)) \end{array} \right] \end{array}$$

où :

- la « macro » $y = x + 1$ remplace la formule $\forall z.(x < z \implies y \leq z) \wedge (z < y \implies z \leq x)$,
- $g_0 = \{d \in \mathcal{D} \mid d > 0\}$ et
- $g = \{(d_1, d_2, d_3) \in \mathcal{D}^3 \mid d_2 - d_3 = d_1\}$.

Lemme 6.4 *Le langage logique $\mathcal{L}$ n'est pas plus expressif que le langage logique $\mathcal{L}_c$, c'est-à-dire que pour toute formule φ dans $\mathcal{L}$, il existe une formule ψ dans $\mathcal{L}_c$ qui est logiquement équivalente à φ.*

PREUVE : Il est facile de vérifier que la formule $P_g(X_1, \ldots, X_\ell, x)$ est équivalente à la formule :

$$\bigvee_{\substack{J \subseteq \{1,\ldots,\ell\} \\ J = \{j_1,\ldots,j_h\} \\ j_1 < j_2 \cdots < j_h}} \exists x_{j_1} \ldots \exists x_{j_h}. \quad \left(\bigwedge_{i \in J} \Big((X_i x_i) \wedge (x_i \leq x) \wedge (\forall y.\ x_i \leq y \leq x \wedge X_i y \implies x_i = y) \Big) \right.$$
$$\left. \wedge \left(\bigwedge_{i \notin J} \forall y.(X_i y \implies x < y) \right) \wedge P_{\overline{g}}(x_{j_1}, \ldots, x_{j_h}) \right)$$

où $(d_1, \ldots d_h) \in \overline{g} \iff (d'_1, \ldots, d'_\ell) \in g$ si $\begin{cases} d'_i = d_r & \text{si } i = j_r \\ d'_i = \perp & \text{si } i \neq j_r \text{ pour tout } r. \end{cases}$ □

Le principal résultat de ce chapitre est le théorème suivant qui établit que le langage $\mathcal{L}$ est équivalent au formalisme des automates de données. Ce théorème apparaît alors comme une extension du théorème de Büchi [Büc62] dans le cadre formel et du résultat de Thomas Wilke dans le cadre des automates temporisés [AD94].

Théorème 6.5 (Caractérisation logique) *Un langage de données est reconnu par un automate (non déterministe) de données si et seulement s'il est l'ensemble des modèles d'une formule close de $\mathcal{L}$.*

Le reste de ce chapitre sera consacré à la preuve de ce théorème d'équivalence, ainsi qu'à une petite discussion sur la décidabilité et l'expressivité du langage $\mathcal{L}$. Cette preuve est assez proche de celle proposée par Thomas Wilke dans [Wil94]. Cependant, elle diffère sur les points techniques. Par exemple, dans [Wil94], les formules logiques sont transformées en des formules qui n'ont qu'une seule variable du premier ordre alors qu'ici, nous allons transformer nos formules en des formules mises sous forme prénexe.

6.2 Une « forme simplifiée » pour les formules de $\mathcal{L}$

Une *formule atomique* est une formule de la forme

$$x < y \mid x = y \mid R_a x \mid Xx \mid P_g(Y_1, \ldots, Y_\ell, x)$$

Une formule atomique $P_g(Y_1, \ldots, Y_\ell, x)$ est appelée un *prédicat de données atomique*.
De manière classique (voir par exemple [CL93]), chaque formule close de $\mathcal{L}$ peut être mise sous forme prénexe :

$$\varphi = \exists X_1 \ldots \exists X_k . Q_1 x_1 \ldots Q_p x_p \left(\bigvee_i \bigwedge_j p_{i,j} \right)$$

où pour tout i, $Q_i \in \{\exists, \forall\}$ et pour tout $i,\ j$, $p_{i,j}$ est une formule atomique ou sa négation.
Nous considérons une formule φ sous forme prénexe. Pour toute variable du second ordre Y apparaissant dans φ, il existe un entier $1 \leq i_Y \leq k$ tel que $Y = X_{i_Y}$. Nous définissons alors la relation d'équivalence $\sim_\varphi$ sur $\mathcal{D}^k$ par :

$$(\theta_i)_{i=1\ldots k} \sim_\varphi (\theta'_i)_{i=1\ldots k} \iff \text{pour tout prédicat de données atomique } P_g(Y_1, \ldots, Y_\ell, x) \text{ de } \varphi,$$
$$\Big((\theta_i)_{i=i_{Y_1}\ldots i_{Y_\ell}} \in g \iff (\theta'_i)_{i=i_{Y_1}\ldots i_{Y_\ell}} \in g \Big)$$

Cette relation d'équivalence est d'index fini. Si $\theta \in \mathcal{D}^k$, la classe d'équivalence de θ (modulo $\sim_\varphi$) est notée $cl_\varphi(\theta)$. Soit g une classe d'équivalence de $\sim_\varphi$, le prédicat de données atomique de la forme $P_g(X_1, \ldots X_k, x)$ est dit *φ-primitif*. Il est immédiat de vérifier que tout prédicat de données atomique apparaissant dans φ peut être transformé en une disjonction de prédicats de données atomiques φ-primitifs.

Nous venons donc de montrer le résultat suivant :

Lemme 6.6 *Toute formule close φ de $\mathcal{L}$ est équivalente à une formule de la forme :*

$$\exists X_1 \ldots \exists X_k . Q_1 x_1 \ldots Q_p x_p \left(\bigvee_i \bigwedge_j p_{i,j} \right)$$

où tout $p_{i,j}$ est une formule atomique ou sa négation et où tout prédicat de données atomique est φ-primitif. De plus, la transformation est effective.

6.3 De la logique aux automates...

L'idée est de transformer une formule de notre langage logique $\mathcal{L}$ en une formule d'une logique monadique du second ordre (sur un autre alphabet, fini) pour laquelle nous construirons un automate fini équivalent (en utilisant le théorème de Büchi [Büc62]). Nous montrerons ensuite qu'il est possible de transformer cet automate fini en un automate de données équivalent à la formule initiale de $\mathcal{L}$.

Soit φ une formule de $\mathcal{L}$. Par le lemme 6.6, nous pouvons supposer que φ est de la forme $\exists X_1 \ldots \exists X_k . Q_1 x_1 \ldots Q_p x_p . \bigvee_i \bigwedge_j p_{i,j}$ où pour chaque h, $Q_h \in \{\exists, \forall\}$ et pour tout i, j, $p_{i,j}$ est une formule atomique ou sa négation et tout prédicat de données atomique est φ-primitif. Soit $\psi = \bigvee_i \bigwedge_j p_{i,j}$. Toutes les variables de ψ sont libres et appartiennent à $\{X_1, \ldots, X_k\} \cup \{x_1, \ldots, x_p\}$. Nous transformons la formule ψ en une formule $\overline{\psi}$ définie inductivement par :

$$\begin{array}{lcl} \overline{x<y} & = & x<y \\ \overline{x=y} & = & x=y \\ \overline{R_a x} & = & R_a x \\ \overline{Xx} & = & R_X x \end{array} \qquad \begin{array}{lcl} \overline{P_g(X_1 \dots X_k, x)} & = & R_g x \\ \overline{\psi_1 \wedge \psi_2} & = & \overline{\psi_1} \wedge \overline{\psi_2} \\ \overline{\psi_1 \vee \psi_2} & = & \overline{\psi_1} \vee \overline{\psi_2} \\ \overline{\neg\psi} & = & \neg\overline{\psi} \end{array}$$

Ces formules sont des formules du premier ordre et sont interprétées sur les mots de l'alphabet $(\Sigma \times \mathcal{D}^k/\sim_\varphi \times \mathcal{P}(\{X_1 \dots X_k\}))$ avec une assignation de variables du premier ordre. Nous noterons encore μ une assignation de variables du premier ordre et nous considérerons un mot sur l'alphabet $(\Sigma \times \mathcal{D}^k/\sim_\varphi \times \mathcal{P}(\{X_1 \dots X_k\}))$, $(a_1, g_1, \mathcal{X}_1) \dots (a_n, g_n, \mathcal{X}_n)$:

$$\begin{array}{rcl} ((a_1, g_1, \mathcal{X}_1) \dots (a_n, g_n, \mathcal{X}_n), \mu) \models R_g x & \iff & g_{\mu(x)} = g \\ ((a_1, g_1, \mathcal{X}_1) \dots (a_n, g_n, \mathcal{X}_n), \mu) \models R_X x & \iff & X \in \mathcal{X}_{\mu(x)} \\ ((a_1, g_1, \mathcal{X}_1) \dots (a_n, g_n, \mathcal{X}_n), \mu) \models R_a x & \iff & a_{\mu(x)} = a \end{array}$$

Les autres formules sont interprétées comme d'habitude.

Remarque : Notons que les formules que nous venons de définir sont bien des formules de la logique du premier ordre pour l'alphabet $\Gamma = (\Sigma \times \mathcal{D}^k/\sim_\varphi \times \mathcal{P}(\{X_1 \dots X_k\}))$. Par exemple, le prédicat $R_g x$ est équivalent à l'union des prédicats $Q_\gamma x$ pour γ lettre de Γ dont la deuxième composante est un g.

Soit $u = (a_1, d_1) \dots (a_n, d_n) \in (\Sigma \times \mathcal{D})^*$ un mot de données et soit ν une assignation pour les variables du second ordre $X_1, \dots, X_k$ (pour tout X_i, $\nu(X_i)$ est donc un ensemble fini d'entiers). Nous associons à u les mots u'_ν et u_ν de la manière suivante :

$$u'_\nu = (a_1, d_1, \theta_1, \mathcal{X}_1) \dots (a_n, d_n, \theta_n, \mathcal{X}_n) \in (\Sigma \times \mathcal{D} \times \mathcal{D}^k \times \mathcal{P}(\{X_1 \dots X_k\}))^*$$

avec $\theta_0 = \perp^k$, $\theta_{i+1}[j] = \begin{cases} \theta_i[j] & \text{si } i \notin \nu(X_j) \\ d_i & \text{si } i \in \nu(X_j) \end{cases}$ et $\mathcal{X}_i = \{X_j \mid i \in \nu(X_j)\}$, et

$$u_\nu = (a_1, cl_\varphi(\theta_1), \mathcal{X}_1) \dots (a_n, cl_\varphi(\theta_n), \mathcal{X}_n) \in (\Sigma \times \mathcal{D}^k / \sim \times \mathcal{P}(\{X_1 \dots X_k\}))^*.$$

La relation entre u, ν et u_ν est établie par le lemme suivant :

Lemme 6.7 $(u, \nu) \models Q_1 x_1 \dots Q_p x_p.\psi \iff u_\nu \models Q_1 x_1 \dots Q_p x_p.\overline{\psi}$

PREUVE : Soit μ une valuation pour les variables du premier ordre. Nous allons montrer que pour toute formule atomique ρ, $(u, \mu.\nu) \models \rho$ si et seulement si $(u_\nu, \mu) \models \overline{\rho}$. Nous obtiendrons alors l'équivalence annoncée car ψ est une combinaison booléenne de formules atomiques.

– si $\rho = R_a x$, alors $\overline{\rho} = R_a x$ et

$$(u, \mu.\nu) \models R_a x \iff a_{\mu(x)} = a \iff (u_\nu, \mu) \models R_a x.$$

– si $\rho = Xx$, alors $\overline{\rho} = R_X x$ et

$$(u, \mu.\nu) \models Xx \iff \mu(x) \in \nu(X) \iff X \in \mathcal{X}_{\mu(x)} \iff (u_\nu, \mu) \models R_X x.$$

– si $\rho = g(X_1, \dots, X_k, x)$, alors $\overline{\rho} = R_g x$ et

$$(u, \mu.\nu) \models P_g(X_1, \dots, X_k, x) \iff \left(d^{(\mu,\nu)}_{\max(X_1 \leq x)}, \dots, d^{(\mu,\nu)}_{\max(X_k \leq x)}\right) \in g.$$

Par construction, $\theta_{\mu(x)} = \left(d^{(\mu,\nu)}_{\max(X_1 \leq x)}, \dots, d^{(\mu,\nu)}_{\max(X_k \leq x)}\right)$. Nous obtenons donc que

$$(u_\nu, \mu) \models R_g x \iff cl_\varphi(\theta_{\mu(x)}) = g \iff (u, \mu.\nu) \models P_g(X_1, \dots, X_k, x).$$

□

Appliquant le résultat fondamental de Büchi [Büc62], nous pouvons construire un automate fini $\overline{\mathcal{A}}$ sur l'alphabet $(\Sigma \times \mathcal{D}^k/\!\sim_\varphi \times \mathcal{P}(\{X_1 \dots X_k\}))^*$ qui reconnaît le langage défini par la formule $Q_1x_1 \dots Q_px_p.\overline{\psi}$. À partir de $\overline{\mathcal{A}}$, nous construisons alors un automate de données $\mathcal{A}$ qui utilise un registre pour chaque variable du second ordre X_i de la manière suivante :

$$\text{si } q \xrightarrow{a,g,\mathcal{X}} q' \text{ est une transition de } \overline{\mathcal{A}} \text{ alors } q \xrightarrow{*,a,up,g} q' \text{ est une transition de } \mathcal{A}$$

où $up = \{i \mid X_i \in \mathcal{X}\}$ et $*$ est une classe d'équivalence quelconque de $\sim_\varphi$. Les états initiaux et finals de $\mathcal{A}$ correspondent aux états initiaux et finals de $\overline{\mathcal{A}}$. L'automate $\mathcal{A}$ est précisément celui que nous cherchions à construire :

Lemme 6.8 *Un mot de données u est accepté par l'automate $\mathcal{A}$ si et seulement si*

$$u \models \exists X_1 \dots \exists X_k.Q_1x_1 \dots Q_px_p.\psi.$$

PREUVE : Nous supposons que $u = (a_1, d_1) \dots (a_n, d_n)$ est accepté par $\mathcal{A}$ sur le chemin

$$q_0 \xrightarrow{*,a_1,up_1,g_1} q_1 \xrightarrow{*,a_2,up_2,g_2} \dots \xrightarrow{*,a_{n-1},up_{n-1},g_{n-1}} q_{n-1} \xrightarrow{*,a_n,up_n,g_n} q_n$$

Soit ν une assignation des variables du second ordre définie par $\nu(X_i) = \{j \mid i \in up_j\}$. Le mot u_ν qui correspond à u peut être lu (sur le chemin correspondant) dans $\overline{\mathcal{A}}$. Ainsi,

$$u_\nu \models Q_1x_1 \dots Q_px_p.\overline{\psi}.$$

Par le lemme 6.7, nous en déduisons que $(u, \nu) \models Q_1x_1 \dots Q_px_p.\psi$.

Réciproquement, supposons que $(u, \nu) \models Q_1x_1 \dots Q_px_p.\psi$. Par ce qui précède, ceci implique que $u_\nu \models Q_1x_1 \dots Q_px_p.\overline{\psi}$. Ainsi, u_ν peut être lu dans $\overline{\mathcal{A}}$. Le mot u peut être lu sur le chemin correspondant dans $\mathcal{A}$. □

Ce lemme termine ainsi la preuve de la première implication du Théorème 6.5.

6.4 ... et *vice-versa*

Nous montrons maintenant la deuxième implication du théorème, c'est-à-dire que tout langage de données accepté par un automate de données peut être défini par une formule de $\mathcal{L}$.
Soit $\mathcal{A} = (Q, q_0, F, k, \sim, T)$ un automate de données. Nous numérotons les transitions de $\mathcal{A}$:

$$T = \{t_i = (q_i, g_i, a_i, up_i, g'_i, q'_i) \mid i = 1 \dots q\}.$$

La « macro » $x + 1$ a été définie à l'exemple 6.3, elle remplace la formule

$$\forall z.(x < z \implies y \leq z) \wedge (z < y \implies z \leq x)$$

La « macro » $x \leq y$ représente la formule $(x < y \vee x = y)$.

Nous utilisons plusieurs variables du second ordre qui joueront deux rôles différents :
- pour toute transition t_i $(1 \leq i \leq q)$, nous utilisons une variable du second ordre X_i qui contiendra toutes les positions où la transition t_i est franchie,
- pour tout registre r_j $(1 \leq j \leq k)$, nous utilisons une variable du second ordre Y_j qui contiendra toutes les positions où le registre r_j est mis à jour.

Soit φ la formule $\exists X_1 \dots \exists X_q . \exists Y_1 \dots \exists Y_k . \Big(\varphi_{pos} \wedge \varphi_{init} \wedge \varphi_{final} \wedge \varphi_{succ} \wedge \varphi_{up} \wedge \varphi_{act} \wedge \varphi_{guards} \Big)$ où les sous-formules sont définies dans le tableau 6.2 qui suit.

· $\varphi_{pos} = \forall x. \left(\bigwedge_{i \neq j} (\neg X_i x \vee \neg X_j x) \wedge \bigvee_i X_i x \right)$

φ_{pos} exprime le fait que chaque position correspond exactement à une transition,

· $\varphi_{init} = \exists x. \left(\left(\bigvee_{i \text{ t.q. } q_i = q_0} X_i x \right) \wedge \forall y. (x \leq y) \right)$

φ_{init} exprime le fait qu'un calcul commence dans un état initial,

· $\varphi_{final} = \exists x. \left(\left(\bigvee_{i \text{ t.q. } q'_i \in F} X_i x \right) \wedge \forall y. (y \leq x) \right)$

φ_{final} exprime le fait qu'un calcul termine dans un état final,

· $\varphi_{succ} = \forall x. \forall y. \left(y = x + 1 \implies \bigvee_{(i,j) \text{ t.q. } q'_i = q_j} (X_i x \wedge X_j y) \right)$

φ_{succ} exprime le fait que, sur un chemin, deux transitions consécutives ont un état en commun,

· $\varphi_{up} = \forall x. \left(\bigwedge_i \left(X_i x \implies \bigwedge_{j \text{ t.q. } j \in up_i} Y_j x \right) \right) \wedge \forall x. \left(\bigwedge_j \left(Y_j x \implies \bigvee_{i \text{ t.q. } j \in up_i} X_i x \right) \right)$

φ_{up} certifie que chaque transition met à jour les bons registres,

· $\varphi_{act} = \forall x. (\bigwedge_i X_i x \implies R_{a_i} x)$

φ_{act} certifie que chaque transition permet d'effectuer la bonne action,

· $\varphi_{guards} = \forall x. (\bigwedge_i (X_i x \implies g_i(Y_1, \dots, Y_k, x)))$

φ_{guards} certifie que les registres vérifient les bonnes contraintes.

Tableau 6.2: Formules décrivant un automate de données

Nous allons maintenant montrer le résultat suivant :

Lemme 6.9 *Pour chaque mot de données u, u est accepté par $\mathcal{A}$ si et seulement si $u \models \varphi$.*

PREUVE : Supposons que le mot de données $u = (a_1, d_1) \dots (a_n, d_n)$ soit accepté par l'automate de données $\mathcal{A}$. Il existe un chemin dans $\mathcal{A}$ qui accepte u :

$$q_0 \xrightarrow{g_1, a_1, up_1, g'_1} q_1 \xrightarrow{g_2, a_2, up_2, g'_2} \dots \xrightarrow{g_n, a_n, up_n, g'_n} q_n$$

Nous définissons l'assignation des variables du second ordre ν par

$$\begin{cases} \nu(X_i) = \{ j \mid q_{j-1} \xrightarrow{g_j, a_j, up_j, g'_j} q_j = t_i \}, \\ \nu(Y_i) = \{ j \mid i \in up_j \}. \end{cases}$$

Il est facile de montrer que

$$(u, \nu) \models \varphi_{pos} \wedge \varphi_{init} \wedge \varphi_{final} \wedge \varphi_{succ} \wedge \varphi_{up} \wedge \varphi_{act} \wedge \varphi_{guards} \qquad (*)$$

Réciproquement, supposons qu'il existe une assignation des variables du second ordre ν telle que $(*)$ soit vérifiée. Alors nous pouvons construire un chemin

$$q_0 \xrightarrow{g_1, a_1, up_1, g'_1} q_1 \xrightarrow{g_2, a_2, up_2, g'_2} \dots \xrightarrow{g_n, a_n, up_n, g'_n} q_n$$

tel que $q_{i-1} \xrightarrow{g_i, a_i, up_i, g'_i} q_i$ est la transition t_j si et seulement si $i \in \nu(X_j)$. Il est alors facile de montrer que le mot de données u peut être lu sur ce chemin dans $\mathcal{A}$ et qu'il est donc accepté par l'automate. □

Nous pouvons constater que, comme les contraintes des automates de données sont des classes d'équivalence d'une relation d'équivalence particulière, les prédicats de données de la formule φ que nous construisons sont φ-primitifs. Ceci conclut la preuve du Théorème 6.5.

6.5 Décidabilité de la logique

Supposons que φ soit une formule de $\mathcal{L}$. Comme nous l'avons écrit dans la partie 6.2, φ définit une relation d'équivalence $\sim_\varphi$ sur $\mathcal{D}^k$ pour un certain k qui dépend de φ. Supposons que la condition suivante soit vérifiée :

$$\text{pour tout } up \subseteq \{1, \ldots, k\},\ \theta_1 \sim \theta_2 \in \mathcal{D}^k,\ d_1 \in \mathcal{D},\ \text{il existe } d_2 \in \mathcal{D} \text{ t.q. } up(\theta_1, d_1) \sim up(\theta_2, d_2) \quad (\ddagger)$$

pour une relation d'équivalence $\sim$ qui raffine la relation $\sim_\varphi$. Cette condition (‡) correspond à la condition (†) présentée dans la partie 5.3. Ainsi, si φ est une formule qui vérifie (‡), alors il est possible de décider si elle a ou non un modèle. Nous obtenons donc le résultat suivant :

Théorème 6.10 *La restriction du langage logique $\mathcal{L}$ aux formules pour lesquelles la condition* (‡) *est vérifiée est décidable.*

Même si, en général, la condition (‡) n'est pas décidable, elle est vérifiée pour de nombreuses classes de relations d'équivalence sur $\mathcal{D}^k$.

***Exemple* 6.11** Reprenons l'exemple 6.3. il est facile de vérifier que la formule proposée pour décrire le langage temporisé $L = \{(a, \tau) \ldots (a, n\tau) \mid \tau > 0 \text{ et } n \text{ entier}\}$ vérifie la condition (‡). Par exemple, nous pouvons considérer la relation d'équivalence $\sim$ défini sur $\mathcal{D}^3$ par :

$$(\theta_i)_{i=1,2,3} \sim (\theta'_i)_{i=1,2,3} \iff \begin{cases} \theta_1 = \theta_3 - \theta_2 \iff \theta'_1 = \theta'_3 - \theta'_2 \\ \theta_1 = \theta_2 - \theta_3 \iff \theta'_1 = \theta'_2 - \theta'_3 \\ \theta_i > 0 \iff \theta'_i > 0 \text{ pour tout } i = 1, 2, 3 \end{cases}$$

Comme dans le cas de la condition (†) pour les automates (voir par exemple l'exemple 5.17 à la page 122), il est sans difficulté de montrer que, dans le cas présent, la condition (†) est vérifiée.

Ainsi, comme L est accepté par aucun automate temporisé classique, il n'y a aucune formule de la logique présentée dans [Wil94] équivalente à la formule de l'exemple 6.3. De plus, chaque automate temporisé classique peut être transformé en un automate de données équivalent qui vérifie la condition (†) (voir l'exemple 5.18), donc en une formule de $\mathcal{L}$ qui vérifie la condition (‡) : le fragment décidable de notre logique est donc strictement plus expressif que la logique monadique de distances proposée dans [Wil94], même lorsque nous nous restreignons aux comportements temporisés (c'est-à-dire même si nous restreignons notre domaine à un domaine de temps).

6.6 Conclusion

Dans ce chapitre, nous avons proposé un langage logique qui permet d'exprimer exactement les langages de données qui sont acceptés par des automates de données. Grâce à toutes ces équivalences (automates, logique, monoïdes), les langages de données nous semblent être un bon candidat pour modéliser les comportements temporisés. En outre, un large fragment de cette logique est décidable et ce fragment décidable est strictement plus expressif que, par exemple, le langage proposé par Thomas Wilke dans [Wil94].

En rassemblant les résultats obtenus dans les chapitres 5 et 6, nous obtenons un parallèle assez intéressant entre les langages formels sur un alphabet fini et les langages de données, ce parallèle est présenté dans le tableau 6.3.

Langages formels	**Langages de données**
Automates finis	Automates de données
Expressions rationnelles	**?**
Caractérisation logique MSO(<), LTL	Logique basée sur MSO(<)
Caractérisation algébrique (en utilisant des monoïdes finis)	Mécanisme algébrique

Tableau 6.3: Comparaison entre les langages formels et les langages de données

En outre, la case qui reste marquée d'un point d'interrogation ne semble pas très difficile à combler. Le travail que nous avons fait à propos du théorème de Kleene pour les langages temporisés (chapitre 4) semble pouvoir s'adapter assez bien aux automates de données : en nous plaçant par exemple dans un cadre « langage de registres » (c'est-à-dire que les lettres seraient des uplets constitués des lettres habituelles et des valeurs des différents registres), nous pourrions prendre pour éléments de base, par exemple, $\langle g, a, up, g'\rangle$, dont la sémantique serait

$$\{\theta.(a, d, \theta') \mid \theta \in g,\ \theta' = up(\theta, d) \text{ et } \theta' \in g'\}.$$

Les langages de données apparaissent vraiment intéressants à la fois pour faire de la théorie des langages sur des alphabets infinis et pour exprimer des comportements temporisés.

Une autre direction possible de recherche après ce travail consiste à étudier la logique plus générale $\mathcal{L}_c$ en terme de décidabilité et d'expressivité.

Troisième partie

Algorithmes et étude de cas

Chapitre 7

Des algorithmes de model-checking

Dans l'introduction de la thèse, nous avons présenté de manière assez informelle le problème du model-checking et nous avons proposé plusieurs axes de recherche complémentaires et sous-jacents à celui-ci. Dans les chapitres précédents, nous avons étudié l'un des axes proposés, à savoir le développement de modèles pour représenter les systèmes que l'on souhaite vérifier.

Nous allons maintenant nous intéresser aux deux autres axes, à savoir le développement d'algorithmes et les études de cas. Ce chapitre introductif de contient pas de résultats originaux, il présente de manière plus précise le cadre dans lequel se situeront nos travaux.

Dans ce chapitre, nous supposons que le modèle de base pour le système est celui des automates temporisés, ou l'une de ses variantes (*cf* chapitre 2). Nous commençons ce chapitre en décrivant les propriétés que nous souhaitons vérifier puis nous présentons deux formalismes logiques qui permettent de les exprimer. Dans un second temps, nous exposons plusieurs algorithmes de model-checking pour lesquels nous précisons les domaines d'application. Nous continuons cette présentation par la description de plusieurs outils de model-checking qui ont déjà été utilisés sur de nombreux exemples industriels. Enfin, nous concluons ce chapitre en décrivant quels sont les travaux que nous présenterons dans cette partie III.

7.1 Les propriétés que nous cherchons à vérifier

Les propriétés qui servent à décrire les comportements des systèmes réels n'apparaissent pas toutes de la même nature. De cette nature va parfois dépendre la technique de vérification qui sera utilisée. Nous reprenons ici la classification de [SBB$^+$01].

7.1.1 Les types de propriétés

La classification suivante nous permet de mieux cerner quelles sont les propriétés qui peuvent être décrites dans tel ou tel langage de spécification.

- Les propriétés d'**accessibilité** (*reachability*) indiquent qu'un état du système peut être atteint.

  ***Exemple* 7.1** Considérons un système avec un état critique et la propriété suivante :

  « L'état critique du système peut être atteint. »

  C'est une propriété d'accessibilité.

- Les propriétés de **sûreté** (*safety*) expriment que, sous certaines conditions, quelque chose ne peut jamais arriver.

***Exemple* 7.2** Considérons le problème de l'exclusion mutuelle. La propriété

« Deux systèmes ne seront jamais simultanément en section critique. »

est une propriété de sûreté. Cette propriété peut par exemple s'instancier en

« Deux trains ne peuvent pas être simultanément sur le pont. »

***Exemple* 7.3** Considérons par exemple une carte bancaire et un distributeur d'argent. La propriété

« Si le code PIN n'est pas correct, le distributeur ne donne pas d'argent. »

est aussi une propriété de sûreté.

Nous pouvons aussi constater que la négation d'une propriété d'accessibilité est en fait une propriété de sûreté. En effet, dire qu'un état n'est pas accessible revient à dire que partout où on peut aller, on n'est pas dans cet état.

- Les propriétés de **vivacité** (*liveness*) expriment que, sous certaines conditions, quelque chose finit par avoir lieu.

***Exemple* 7.4** Considérons un ascenseur. La propriété

« Si l'ascenseur est appelé au 6ème étage, alors il s'arrêtera, un jour, au 6ème étage. »

est une propriété de vivacité.

- Les propriétés de **vivacité bornée** (*bounded liveness*) sont des propriétés de vivacité mais avec des délais maximaux d'attente.

***Exemple* 7.5** Revenons à l'ascenseur de l'exemple précédent. La propriété

« Si l'ascenseur est appelé au 6ème étage, alors il s'arrêtera dans moins de 5 minutes au 6ème étage. »

est une propriété de vivacité bornée.

Les propriétés de vivacité bornée sont en fait des propriétés de sûreté car l'expression « une propriété doit être vraie avant un délai donné » peut se traduire en « il est toujours vrai que soit le délai n'est pas passé, soit la propriété a eu lieu », ce qui correspond bien à une propriété de sûreté.

- Les propriétés d'**équité** (*fairness*) expriment que, sous certaines conditions, quelque chose aura lieu infiniment souvent.

***Exemple* 7.6** Considérons la barrière d'un passage à niveau. La propriété

« La barrière est ouverte infiniment souvent. »

est une propriété d'équité

- Les propriétés d'**absence de blocage** (*no deadlock*) expriment que le système ne se retrouve jamais dans un état qu'il n'a plus la possibilité de quitter.

***Exemple* 7.7** Considérons à nouveau un ascenseur. La propriété

« L'ascenseur peut toujours changer d'étage. »

est une propriété d'absence de blocage.

Maintenant que nous avons décrit une classification possible des différents types de propriétés, nous allons voir comment il est possible de décrire ces propriétés « de manière mathématique ». Pour cela, il est nécessaire d'avoir des langages de spécification adéquats.
Dans le cadre non temporisé, de nombreuses logiques ont été définies et utilisées. Le but n'est pas ici de les présenter, nous pouvons cependant en citer quelques-unes, par exemples des logiques temporelles comme LTL [Pnu77] ou CTL [CES86], des logiques provenant de l'arithmétique [Büc62] ou bien le μ-calcul [Koz83].

Dans le cadre temporisé, les logiques qui sont utilisées sont souvent des logiques provenant du cadre temporisé dans lesquelles des notions de temporisation ont été rajoutées.
Nous présenterons dans ce qui suit deux types de langages de spécification pour les systèmes temporisés, l'extension temporisée de la logique temporelle du temps arborescent, TCTL et une logique modale L_ν. Nous avons choisi parmi beaucoup d'autres ces deux logiques car elles nous semblent relativement représentatives.

7.1.2 Les langages de spécification

La logique TCTL. La logique TCTL (TCTL signifie *Timed Computation Tree Logic*) est une extension temporisée de CTL [CES86]. Elle a été définie dans [ACD90]. Des syntaxes légèrement différentes sont proposées, par exemple dans [HNSY94, Yov97, Daw98, Yov98, TY01]. C'est une logique arborescente qui s'interprète sur les systèmes de transitions temporisés, donc en particulier sur les automates temporisés. Nous utilisons la présentation de TCTL faite dans [TY01], qui nous semble la plus simple. Les autres présentations sont bien entendu équivalentes.

Si $\mathcal{I}$ est l'ensemble des intervalles à bornes entières ou infinies de $\mathbb{R}^+$ et si P est un ensemble de propositions atomiques, les formules de TCTL bâties sur P sont définies inductivement par la grammaire suivante :

$$\varphi \ ::= \ p \mid \mathtt{tt} \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \mathsf{E}\,\varphi\,\mathsf{U}_I\,\varphi \mid \mathsf{A}\,\varphi\,\mathsf{U}_I\,\varphi$$

où $p \in P$ et $I \in \mathcal{I}$.

Rappelons (voir la partie 3.4.1) qu'un système de transitions est un 4-uplet $\mathcal{T} = (S, \Gamma, s_0, \longrightarrow)$ où S est un ensemble d'états, Γ est un alphabet, $s_0 \in S$ est l'état initial et $\longrightarrow \subseteq S \times \Gamma \times S$ est un ensemble de transitions. On dit que ce système de transitions est *étiqueté* (par P) s'il existe une fonction λ qui associe à chaque état du système de transitions considéré un sous-ensemble de P. Intuitivement, si $p \in P$ étiquette un état, cela signifie que p est vraie dans cet état.

Les formules de TCTL sont interprétées sur les états des systèmes de transitions étiquetés. La relation de satisfaction $\models$ est définie dans le tableau 7.1.

$s \models \mathtt{tt}$	si	toujours vrai
$s \models p$	si	s étiqueté par p (*i.e.* $p \in \lambda(s)$)
$s \models \neg\varphi$	si	$s \not\models \varphi$
$s \models \varphi_1 \vee \varphi_2$	si	$s \models \varphi_1$ ou $s \models \varphi_2$
$s \models \mathsf{E}\,\varphi_1\,\mathsf{U}_I\,\varphi_2$	si	il existe un chemin $s \rightsquigarrow s'$ tel que $s' \models \varphi_2$,
		la durée du chemin est dans I et
		tout au long du chemin, φ_1 est vérifiée
$s \models \mathsf{A}\,\varphi_1\,\mathsf{U}_I\,\varphi_2$	si	sur tout chemin partant de s,
		il existe $s \rightsquigarrow s'$ tel que $s' \models \varphi_2$,
		la durée du chemin $s \rightsquigarrow s'$ est dans I et
		tout au long du chemin, φ_1 est vérifiée

Tableau 7.1: Relation de satisfaction pour TCTL

Intuitivement, une formule $\mathsf{E}\ \varphi_1\ \mathsf{U}_I\ \varphi_2$ exprime qu'à partir de la configuration courante, il est possible de suivre un chemin dont un préfixe, de durée dans l'intervalle I, vérifie φ_1 à chaque instant et la dernière configuration de ce préfixe vérifie φ_2. La formule $\mathsf{A}\ \varphi_1\ \mathsf{U}_I\ \varphi_2$, elle, exprime la propriété précédente pour tout chemin partant de la configuration courante.

Les macros suivantes sont très souvent utilisées :

$$\begin{array}{ll} \mathsf{EF}_I\ \varphi \equiv \mathsf{E}\ (\mathtt{tt}\ \mathsf{U}_I\ \varphi) & \mathsf{AF}_I\ \varphi \equiv \mathsf{A}\ (\mathtt{tt}\ \mathsf{U}_I\ \varphi) \\ \mathsf{EG}_I\ \varphi \equiv \neg\Big(\mathsf{AF}_I(\neg\varphi)\Big) & \mathsf{AG}_I\ \varphi \equiv \neg\Big(\mathsf{EF}_I(\neg\varphi)\Big) \\ (\varphi_1 \implies \varphi_2) \equiv (\neg\varphi_1) \vee \varphi_2 & \end{array}$$

Par exemple, $\mathsf{EF}_{[0,5]}\ \varphi$ exprime qu'il est possible d'atteindre, en moins de 5 unités de temps, une configuration qui vérifie φ alors que $\mathsf{AG}\ \varphi$ exprime que toute configuration qu'il est possible d'atteindre vérifie φ. Les indices « $_I$ » indiquent en outre que φ doit être vérifiée dans une durée qui est incluse dans I. Si l'intervalle I vaut $[0; +\infty[$, l'indice « $_I$ » est omis.

Nous revenons aux exemples de la partie 7.1.1 et nous allons voir que cette logique permet d'exprimer à peu près tous les types de propriétés. Nous reprenons nos exemples un par un :

- ***Exemple 7.1*** La propriété décrite peut s'exprimer par la formule

  $$\mathsf{EF}(\mathsf{état\_critique})$$

  En effet, cette formule indique qu'il y a un chemin qui mène à l'état critique.
- ***Exemple 7.2*** La propriété décrite peut s'exprimer par la formule

  $$\mathsf{AG}\Big((\neg\mathsf{section\_critique1}) \vee (\neg\mathsf{section\_critique2})\Big)$$

  En effet, cette formule indique qu'il n'est pas possible d'être simultanément dans la section critique 1 et dans la section critique 2.
- ***Exemple 7.3*** La propriété décrite peut s'exprimer par la formule

  $$\mathsf{A}\Big((\neg\mathsf{donne\_argent})\ \mathsf{U}\ (\mathsf{code\_correct})\Big)$$

  En effet, cette formule indique qu'aucun argent n'est donné tant que le code correct n'est pas tapé.
- ***Exemple 7.4*** La propriété décrite peut s'exprimer par la formule

  $$\mathsf{AG}\Big(\mathsf{appel\_6} \implies (\mathsf{AF\ arrêt\_6})\Big)$$

  En effet, cette formule indique que, à n'importe quel moment, si l'ascenseur est appelé à l'étage 6, alors, un jour, l'ascenseur s'arrêtera à cet étage.
- ***Exemple 7.5*** La propriété décrite peut s'exprimer par la formule

  $$\mathsf{AG}\Big(\mathsf{appel\_6} \implies (\mathsf{AF}_{[0;5]}\ \mathsf{arrêt\_6})\Big)$$

  En effet, cette formule indique la même chose que précédemment, mais elle impose en outre que l'ascenseur s'arrêtera à l'étage 6 avant 5 unités de temps.
- ***Exemple 7.6*** La propriété décrite peut s'exprimer par la formule

  $$\mathsf{AG}\ \mathsf{AF}(\mathsf{barrière\_ouverte})$$

  Pour certaines propriétés d'équité, il est nécessaire d'introduire de nouveaux opérateurs, $\overset{\infty}{\mathsf{F}}$ et $\overset{\infty}{\mathsf{G}}$, qui indiquent qu'infiniment souvent, une propriété sera vérifiée. Nous renvoyons à [SBB+99] pour plus de détails sur ces opérateurs.

• ***Exemple 7.7*** La propriété décrite peut s'exprimer par la formule

$$\mathsf{AG}\left(\bigvee_i\left(\mathsf{état\_i} \implies \left(\mathsf{état\_i}\ \mathsf{U} \bigvee_{j\neq i}\mathsf{état\_j}\right)\right)\right)$$

Dans cette formule, nous supposons que chaque état est étiqueté par une proposition atomique différente état_i. Cette formule indique alors que si nous nous trouvons dans un état i, alors plus tard, nous nous trouverons dans un état j distinct de i.

La logique TCTL est une logique très expressive, la plupart des propriétés pouvant s'y exprimer assez aisément. Le model-checking de TCTL, en prenant comme modèle les automates temporisés, a été montré PSPACE-complet [ACD90, ACD93]. L'outil KRONOS utilise TCTL comme langage de spécification. Nous parlerons de l'algorithme implémenté dans KRONOS dans la partie 7.2.3.

Les logiques modales temporisées comme L_ν. Nous allons maintenant présenter une logique modale temporisée, L_ν. Il en existe beaucoup d'autres, voir par exemple [AL99, AL02]. La logique L_ν sur un ensemble d'horloges K et un ensemble de variables $\mathcal{V}$ est définie par la grammaire suivante :

$$\varphi \ ::= \ \mathtt{tt} \ | \ \mathtt{ff} \ | \ g \ | \ \varphi\wedge\varphi \ | \ \varphi\vee\varphi \ | \ \exists\varphi \ | \ \forall\!\!\!\forall\varphi \ | \ \langle a\rangle\varphi \ | \ [a]\varphi \ | \ x \ \mathsf{in}\ \varphi \ | \ Z \ | \max(Z,\varphi)$$

où $g \in \mathcal{C}(K)$, $x \in K$, $Z \in \mathcal{V}$ et a est une action.

Les formules de L_ν sont interprétées sur les couples (s, v) où s est un état d'un système de transitions temporisé et v une valuation des horloges de K. À cause des formules récursives $\max(Z, \varphi)$, nous définissons la relation de satisfaction $\models$ comme étant la plus grande relation satisfaisant les implications décrites dans le tableau 7.2.

$(s,v)\models \mathtt{tt}$	$\implies$	toujours vrai
$(s,v)\models \mathtt{ff}$	$\implies$	toujours faux
$(s,v)\models g$	$\implies$	$v\models g$
$(s,v)\models \varphi_1\wedge\varphi_2$	$\implies$	$(s,v)\models\varphi_1$ et $(s,v)\models\varphi_2$
$(s,v)\models \varphi_1\vee\varphi_2$	$\implies$	$(s,v)\models\varphi_1$ ou $(s,v)\models\varphi_2$
$(s,v)\models \exists\varphi$	$\implies$	$\exists s \xrightarrow{\epsilon(d)} s',\ (s',v+d)\models\varphi$
$(s,v)\models \forall\!\!\!\forall\varphi$	$\implies$	$\forall s \xrightarrow{\epsilon(d)} s',\ (s',v+d)\models\varphi$
$(s,v)\models \langle a\rangle\varphi$	$\implies$	$\exists s \xrightarrow{a} s',\ (s',v)\models\varphi$
$(s,v)\models [a]\varphi$	$\implies$	$\forall s \xrightarrow{a} s',\ (s',v)\models\varphi$
$(s,v)\models x\ \mathsf{in}\ \varphi$	$\implies$	$(s,v[x\leftarrow 0])\models\varphi$
$(s,v)\models \max(Z,\varphi)$	$\implies$	$(s,v)\models\varphi\{Z/\max(Z,\varphi)\}$[a]

[a] $\varphi\{Z/\max(Z,\varphi)\}$ représente la formule φ dans laquelle la variable Z a été remplacée par $\max(Z,\varphi)$.

Tableau 7.2: Relation de satisfaction pour L_ν

Intuitivement, la formule $\exists\varphi$ permet d'exprimer qu'il est possible de laisser s'écouler du temps puis de vérifier φ. La formule $\forall\!\!\!\forall\varphi$ exprime que φ reste vérifiée tant que nous attendons. La formule $\langle a\rangle\varphi$ exprime qu'il est possible de faire une action a puis de vérifier φ alors que la formule $[a]\varphi$ exprime que quelque soit la manière de faire un a, φ est vérifiée. Par exemple, la formule $[a]\mathtt{ff}$ exprime qu'il n'est pas possible de faire un a. La formule $\max(Z, \varphi)$ représente le plus grand point fixe de l'équation $Z = \varphi$. Il se calcule par exemple (voir [Tar55] ou les notes de cours [Ace94] pour un peu de théorie des points fixes) en définissant $\varphi^{(0)} = \varphi\{Z/\mathtt{tt}\}$, puis en définissant inductivement $\varphi^{(l)} = \varphi\{Z/\varphi^{(l-1)}\}$ pour tout $l \geq 1$. Si $[\![\psi]\!]$ représente l'ensemble des états vérifiant la formule ψ (*i.e.* c'est l'*ensemble caractéristique* de ψ), alors l'ensemble des états qui vérifient $\max(Z, \varphi)$ est

$$[\![\max(Z, \varphi)]\!] = \bigcap_{l \geq 0} [\![\varphi^{(l)}]\!]$$

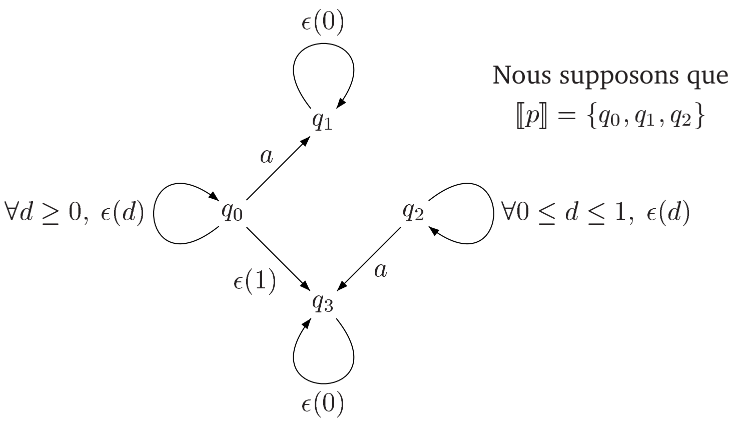


Figure 7.3: Système de transitions $\mathcal{T}$

***Exemple* 7.8** Soit ψ une propriété.

– La macro $\max(Z, \bigwedge_a [a]Z \wedge \psi)$ signifie que n'importe quel état accessible par une succession d'actions (aucune attente entre les actions n'est autorisée) doit vérifier ψ,

  Par exemple, considérons le système de transitions $\mathcal{T}$ dessiné sur la figure 7.3 pour lequel nous supposons que les états $q_0,\ q_1,\ q_2$ sont étiquetés par une proposition p, mais q_3 ne l'est pas. Le calcul de l'ensemble des états qui satisfont $\max(Z, [a]Z \wedge p)$ se fait de la manière suivante :

$$\begin{array}{rcl} [\![[a]\mathtt{tt} \wedge p]\!] & = & [\![p]\!] \\ & = & \{q_0, q_1, q_2\} \\ [\![[a]\,([a]\mathtt{tt} \wedge p) \wedge p]\!] & = & [\![[a]p \wedge p]\!] \\ & = & \{q_0, q_1\} \\ [\![[a]\,([a]\,([a]\mathtt{tt} \wedge p) \wedge p) \wedge p]\!] & = & [\![[a][a]p \wedge [a]p \wedge p]\!] \\ & = & \{q_0, q_1\} \end{array}$$

  Le calcul du point fixe se stabilise au bout de la troisième étape de calcul. Nous obtenons alors que

$$[\![\max(Z, [a]Z \wedge p)]\!] = \{q_0, q_1\}.$$

– La macro $\max(Z, \forall Z \wedge \bigwedge_a [a]Z \wedge \psi)$ signifie que n'importe quel état accessible doit vérifier la propriété ψ. En effet, la conjonction « $\forall Z \wedge \bigwedge_a [a]Z$ » permet d'atteindre tous les états accessibles du système (elle permet d'alterner les attentes et les actions).

  Si l'on revient au système de transitions $\mathcal{T}$, le calcul de l'ensemble des états qui vérifient $\max(Z, \forall Z \wedge [a]Z \wedge p)$ se fait de la même façon que pour $\max(Z, [a]Z \wedge p)$. Nous obtenons alors que

$$[\![\max(Z, \forall Z \wedge [a]Z \wedge p)]\!] = \{q_1\}$$

  car lorsque l'on est dans q_0, on peut arriver à l'état q_3 en laissant passer une unité de temps, et q_3 ne vérifie pas p.

La logique L_ν est une logique modale temporisée avec plus grand point fixe, mais pas de plus petit point fixe. Elle permet ainsi d'exprimer essentiellement des propriétés de sûreté et de vivacité bornée. Revenons aux exemples de la partie 7.1.1.

- ***Exemple 7.2*** La propriété peut s'exprimer par

$$\max\left(Z, (\neg\mathsf{section\_critique1} \vee \neg\mathsf{section\_critique2}) \wedge \bigwedge_a [a]Z \wedge \forall\!\!\!\forall Z\right)$$

  En effet, cette formule signifie bien que tous les états accessibles sont tels que l'on n'est jamais simultanément dans les deux sections critiques.

- ***Exemple 7.3*** La propriété peut s'exprimer par

$$\max\left(Z, \mathsf{code\_correct} \vee \left(\neg\mathsf{donne\_argent} \wedge \bigwedge_a [a]Z \wedge \forall\!\!\!\forall Z\right)\right)$$

  Cette formule est un peu plus compliquée que la précédente, mais elle exprime bien ce que l'on souhaite : elle exprime que tant que l'on n'arrive pas dans un état où le code est correct, on continue le parcours (macro « $\bigwedge_a [a]Z \wedge \forall\!\!\!\forall Z$ ») en ne donnant pas d'argent.

- ***Exemple 7.4*** La propriété peut s'exprimer localement par

$$\psi = \mathsf{appel\_6} \implies \left(x \text{ in } \max\left(Z, \mathsf{arrêt\_6} \vee (x \leq 5 \wedge \bigwedge_a [a]Z \wedge \forall\!\!\!\forall Z)\right)\right)$$

  donc globalement par

$$\max\left(Y, \psi \wedge \bigwedge_a [a]Y \wedge \forall\!\!\!\forall Y\right).$$

  La première formule reprend l'exemple précédent : la sous-formule

$$\max\left(Z, \mathsf{arrêt\_6} \vee (x \leq 5 \wedge \bigwedge_a [a]Z \wedge \forall\!\!\!\forall Z)\right)$$

  s'interprète comme « tant que l'ascenseur ne s'arrête pas au $6^{\text{ème}}$ étage, alors l'horloge x est plus petite que 5 », ce qui est équivalent à « l'ascenseur s'arrête au $6^{\text{ème}}$ étage avant que l'horloge x ne dépasse 5 ». La formule ψ exprime alors que si l'ascenseur est appelé au $6^{\text{ème}}$ étage, alors il s'y arrêtera avant 5 unités de temps car l'horloge x est remise à zéro juste au moment où l'ascenseur est appelé. La formule globale revient à dire que de tout état du système, la propriété ψ est vérifiée.

- L'analyse d'accessibilité peut aussi être spécifiée par le langage L_ν en exprimant en fait la négation de la propriété, qui est, comme nous l'avons déjà dit, une propriété de sûreté.
  Exemple 7.1 La propriété inverse, qui dit qu'aucun état critique n'est accessible peut s'exprimer par la formule :

$$\max\left(Z, \neg\mathsf{état\_critique} \wedge \bigwedge_a [a]Z \wedge \forall\!\!\!\forall Z\right)$$

  Notons que si nous disposions d'un plus petit point fixe, la propriété d'accessibilité pourrait s'exprimer directement par

$$\min\left(Z, \mathsf{état\_critique} \vee \bigvee_a \langle a\rangle Z \vee \exists Z\right)$$

La logique L_ν est intéressante : elle permet par exemple d'exprimer la notion de bisimulation pour les automates temporisés [LLW95, AIPP00] *via* la notion de *formules caractéristiques*. Le model-checking de L_ν est EXPTIME-complet, mais de nombreuses sous-classes de L_ν sont d'une complexité moindre pour le model-checking [AL99, AL02]. Des algorithmes de model-checking pour L_ν ont été décrits et étudiés dans [LL95, LPY95]. Nous décrirons l'un de ces algorithmes dans la partie 7.2.4. Enfin, cette logique est le langage de spécification de l'outil de model-checking CMC [LL98].

Importance du problème de l'accessibilité. Les propriétés d'accessibilité sont souvent les plus simples à vérifier et les plus importantes. Elles ont montré leur intérêt dans une multitude d'études de cas (voir par exemple [BGK+96, HSLL97]). Il est en outre possible de ramener l'étude de certaines propriétés à de l'analyse d'accessibilité [Wol97]. Une technique d'*automate de test* est utilisée dans [JLS96] pour réduire la vérification d'une propriété de vivacité bornée à la vérification d'une propriété d'accessibilité.

7.2 Des algorithmes de model-checking

Jusqu'à présent, dans la partie II, la seule véritable question de vérification à laquelle nous nous sommes intéressés est le problème de la décidabilité du vide. Ce problème étant strictement équivalent à celui de l'accessibilité d'un état donné, montrer la décidabilité du vide des modèles est un premier pas important. Toutes nos preuves de décidabilité reposent sur la construction d'un automate fini ou de Büchi assez proche de l'automate des régions [AD90, AD94] (voir la partie 2.3.2). Cependant, en pratique, cette construction de l'automate des régions n'est pas implémentée car un tel algorithme ne serait pas efficace et il n'y a pas de structures de données vraiment adaptées aux régions. En pratique, d'autres algorithmes sont implémentés. Nous allons en décrire plusieurs.

7.2.1 L'accessibilité par analyse en avant

Description générale. L'idée de l'*analyse en avant* est de calculer **tous** les états qu'il est possible d'atteindre en partant d'une configuration donnée du système. Bien sûr, dans le cas temporisé comme dans beaucoup de cas où les configurations possibles du système sont en nombre infini, il n'est pas possible de calculer ces configurations une par une. La difficulté est alors contournée en utilisant des *représentations symboliques* d'ensembles de configurations, ces ensembles pouvant très bien être infinis. Le principe de l'analyse en avant est alors de calculer

$$\mathsf{Post}^*(\mathsf{init})$$

où init est la configuration initiale (ou l'ensemble des configurations initiales) du système et Post^* est la clôture réflexive et transitive de Post défini par

$$\mathsf{Post}(E) = \{\text{configurations accessibles en « un pas » à partir de } E\}$$

si E est un ensemble de configurations, la notion de « un pas » étant bien sûr à préciser (elle varie selon les modèles).

Ce calcul de Post^* permet alors de tester l'accessibilité d'un ensemble d'états F à partir des configurations initiales en intersectant l'ensemble calculé avec F.

Le principe décrit ci-dessus n'est pas propre aux automates temporisés. Il peut être utilisé par exemple pour des automates à compteurs, des réseaux de Pétri, des automates hybrides, etc... Ce genre d'algorithme est par exemple utilisé dans l'outil HYTECH [HHWT95a, HHWT95b, HHWT97] où les ensembles de configurations sont représentés par des *polyèdres convexes*, c'est-à-dire des intersections d'hyperplans de l'espace des temps.

Cas particulier des automates temporisés. L'algorithme que nous présentons dans cette partie est un exemple d'algorithme d'analyse en avant pour les automate temporisés. Il est parfois appelé l'*algorithme des zones* ou l'*algorithme de résolution de contraintes*.
Si $\mathcal{A}$ est un automate temporisé, une configuration de $\mathcal{A}$ est un couple (q, v) où q est un état de $\mathcal{A}$ et v une valuation des horloges de $\mathcal{A}$. Les ensembles de configurations que nous considérons sont de la forme

$$\bigcup_q \{(q, v) \mid v \in Z_q\} \tag{7.1}$$

où pour tout q, Z_q est une *zone*, c'est-à-dire un ensemble convexe de valuations défini par une contrainte d'horloges (avec des constantes entières).
La fonction Post est alors définie de la manière suivante :

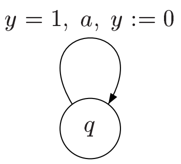
$$\mathsf{Post}((q, v)) = \left\{ (q', v') \;\middle|\; \begin{array}{l} \text{soit } \exists d \in \mathbb{T} \text{ t.q. } q = q' \text{ et } v' = v + d \\ \text{soit } \exists q \xrightarrow{g,a,C} q' \text{ t.q. } v \models g \text{ et } v' = [C \leftarrow 0]v^{1} \end{array} \right\}$$

Il est possible de montrer que l'image par Post de tout ensemble de configurations de la forme (7.1) est encore de la forme (7.1). Notons que des structures de données ont été proposées et sont adaptées au stockage d'ensembles de configurations du type (7.1) et aux différents calculs nécessaires. Parmi ces structures de données, nous mentionnons :

– les DBMs (Difference Bounded Matrices) [Dil89],
– les graphes réduits aux plus courts chemins [LLPY97],
– les CDDs (Clock Difference Diagrams) [BLP+99, LWYP99],
– les NDDs (Numerical Decision Diagrams) [ABK+97].

Cependant, le calcul de Post^* souffre d'un problème non négligeable : il peut ne pas terminer. Pour illustrer cela, nous présentons un petit contre-exemple :

***Contre-exemple* 7.9** Considérons l'automate suivant ayant un seul état mais deux horloges x et y et la configuration initiale $\mathsf{init} = \{(q, \mathbf{0})\}$. Alors, nous avons :



$$\begin{array}{rcl}
\mathsf{Post}(\mathsf{init}) & = & \{(q, v) \mid v(x) = v(y)\} \\
\mathsf{Post}^2(\mathsf{init}) \setminus \mathsf{Post}(\mathsf{init}) & = & \{(q, v) \mid v(x) = 1 \text{ et } v(y) = 0\} \\
\mathsf{Post}^3(\mathsf{init}) \setminus \mathsf{Post}^2(\mathsf{init}) & = & \{(q, v) \mid v(x) = v(y) + 1\} \\
& \vdots & \\
\mathsf{Post}^{2i}(\mathsf{init}) \setminus \mathsf{Post}^{2i-1}(\mathsf{init}) & = & \{(q, v) \mid v(x) = i \text{ et } v(y) = 0\} \\
\mathsf{Post}^{2i+1}(\mathsf{init}) \setminus \mathsf{Post}^{2i}(\mathsf{init}) & = & \{(q, v) \mid v(x) = v(y) + i\} \\
& \vdots &
\end{array}$$

Nous constatons alors que le calcul de $\mathsf{Post}^*(\mathsf{init})$ ne termine pas.

Pour pallier ce problème, il est possible de calculer à chaque étape un sur-ensemble des configurations accessibles au lieu de l'ensemble exact des configurations accessibles. Ceci peut être réalisé en utilisant une fonction Post′ à la place de Post telle que pour toute zone Z,

$$\mathsf{Post}(Z) \subseteq \mathsf{Post}'(Z)$$

[1]Rappelons que, comme il a été défini dans la partie 2.1.2, la valuation $[C \leftarrow 0]v$ est définie par $([C \leftarrow 0]v)(x) = 0$ si $x \in C$ et $([C \leftarrow 0]v)(x) = v(x)$ sinon.

et telle que le calcul de $\mathsf{Post}'^*(\mathsf{init})$ termine. Bien sûr, pour que le calcul réalisé corresponde bien à ce que nous souhaitons calculer, il faut s'assurer que l'intersection de $\mathsf{Post}'^*(\mathsf{init})$ avec l'ensemble d'états que nous testons corresponde à l'intersection de $\mathsf{Post}^*(\mathsf{init})$ avec ce même ensemble d'états. Dans l'article [DT98], plusieurs telles surapproximations (appelées alors *abstractions*) sont étudiées.

Ces algorithmes d'analyse en avant sont beaucoup utilisées pour les automates temporisés (*cf* par exemple [OY93, BL96, TY96, Yov97, Daw97, BTY97, Daw98, BDM$^+$98, Yov98, Alu99]).

Des algorithmes de ce type sont implémentés dans plusieurs outils, par exemple KRONOS [Yov97] ou UPPAAL [BL96]. Ces outils utilisent une surapproximation, ce qui leur permet de s'arrêter à chaque fois. En outre, le calcul s'arrête lorsqu'un état que nous cherchons à atteindre a été atteint, ce qui écourte souvent les calculs. Par contre, dans HYTECH [HHWT97], aucune surapproximation n'est utilisée : le calcul fait ne termine pas toujours. En outre, le test d'intersection avec l'ensemble que nous cherchons à atteindre n'est réalisé qu'après tout ce calcul.

7.2.2 L'accessibilité par analyse en arrière

L'idée de l'*analyse en arrière* est l'inverse de celle de l'analyse en avant. Au lieu de calculer les configurations qui succèdent aux configurations initiales, cet algorithme calcule les configurations qui précèdent les configurations que nous cherchons à atteindre. Si final est l'ensemble des configurations que nous souhaitons atteindre, l'analyse en arrière consiste alors à calculer $\mathsf{Pré}^*(\mathsf{final})$ où la fonction $\mathsf{Pré}$ est définie par

$$\mathsf{Pré}(S) = \{\text{configurations qui permettent d'atteindre } S \text{ « en un pas »}\}$$

Dans le cadre des automates temporisés, les mêmes structures de données que celles utilisées dans le cas de l'analyse en avant sont aussi adaptées à l'analyse en arrière. Cet algorithme est implémenté pour les automates temporisés dans l'outil KRONOS [Yov97, Daw98, Yov98].

Avant de poursuivre, remarquons que parfois, l'analyse en avant est meilleure (dans le sens où elle répond plus vite) que l'analyse en arrière, mais d'autres fois, c'est le contraire. Il n'y a pas de règle précise. Les deux approches apparaissent alors comme complémentaires, « il faut tester les deux ».

7.2.3 Un algorithme pour vérifier TCTL

Nous allons maintenant donner une idée sur comment il est possible de vérifier les propriétés de TCTL. La construction que nous allons décrire provient de l'article [Yov98], elle évite de réduire la vérification de TCTL à celle de CTL en passant par l'automate des régions, car cette réduction n'est pas très efficace, comme nous l'avons remarqué dans un autre contexte dans la partie 7.2. Notons tout de même que dans [ACD$^+$92b], un algorithme qui utilise une « minimisation » de l'automate des régions est proposé pour TCTL.

L'idée de l'algorithme présenté dans [Yov98] est de calculer pour chaque formule, son *ensemble caractéristique*, c'est-à-dire

$$[\![\psi]\!] = \{((q,v),w) \mid (q,v) \models \psi\}$$

où w est une valuation sur un certain nombre d'horloges, correspondant au nombre d'opérateurs U dans la formule (nous verrons l'utilité de ce w plus loin).

La construction se fait par induction sur la structure de la formule Les cas suivants sont simples :

$$\begin{aligned}
&[\![p]\!] = \{((q,v),w) \mid q \text{ étiqueté par } p\}\\
&[\![\mathsf{Vrai}]\!] = \{((q,v),w) \mid q \text{ état}\}\\
&[\![\neg\varphi]\!] = [\![\mathsf{Vrai}]\!] \setminus [\![\varphi]\!]\\
&[\![\varphi_1 \vee \varphi_2]\!] = [\![\varphi_1]\!] \cup [\![\varphi_2]\!]\\
&[\![\varphi_1 \wedge \varphi_2]\!] = [\![\varphi_1]\!] \cap [\![\varphi_2]\!]
\end{aligned}$$

Il reste à décrire les ensembles caractéristiques des formules qui ont un opérateur U. Nous considérons la formule $\mathsf{E}\ \varphi_1\ \mathsf{U}_I\ \varphi_2$.

$$[\![\mathsf{E}\ \varphi_1\ \mathsf{U}_I\ \varphi_2]\!] = EU\Big([z \leftarrow 0][\![\varphi_1]\!], [\![\varphi_2]\!] \cap \{((q,v),w) \mid w(z) \in I\}\Big) \tag{7.2}$$

où z est l'horloge correspondant à l'opérateur U que nous considérons et $EU\Big(R_1, R_2\Big) = \bigcup_{i\geq 0} E_i$ avec

$$\begin{cases} E_0 = R_2 \\ E_{i+1} = \mathsf{Pré}[R_1](E_i) \cup \mathsf{Pré}(E_i) \end{cases}$$

$\mathsf{Pré}[R_1](E_i)$ représente l'ensemble des configurations qui permettent d'atteindre E_i en laissant le temps s'écouler tout en restant dans R_1 alors que $\mathsf{Pré}(E_i)$ représente les configurations qui permettent d'atteindre E_i en prenant une transition.

Une horloge est attachée à chaque opérateur U de la formule. Son rôle apparaît dans la formule (7.2). Elle sert à assurer que le temps qui sépare le moment où φ_1 est vérifiée et celui où φ_2 est vérifiée se trouve bien dans l'intervalle I.

Nous remarquons que le calcul précédent est en fait un calcul en arrière. Nous ne décrivons pas le calcul de $[\![\mathsf{A}\ \varphi_1\ \mathsf{U}_I\ \varphi_2]\!]$ qui utilise aussi un calcul en arrière, mais qui un peu plus compliqué, il est par exemple décrit dans [HNSY94].

Le genre de calculs que nous venons de décrire pour les formules de TCTL peuvent aussi se faire pour des logiques telles que L_ν (voir [HNSY94] où une technique analogue est utilisée pour un sur-ensemble de L_ν). Une technique légèrement différente mais qui conserve la même idée, celle de calculer de manière inductive l'ensemble caractéristique des formules, est décrite pour L_ν dans [LPY95] ou juste pour des propriétés d'accessibilité dans [YPD94].

7.2.4 Le model-checking compositionnel

Étant donnés un réseau d'automates $(A_1 \parallel \cdots \parallel A_k)$ et une propriété φ, nous souhaitons vérifier que le système $(A_1 \parallel \cdots \parallel A_k)$ vérifie φ. La technique du *model-checking compositionnel* consiste à calculer une formule notée $\varphi_{/A_k}$ à partir de φ et de A_k telle que :

$$(A_1 \parallel \cdots \parallel A_k) \models \varphi \iff (A_1 \parallel \cdots \parallel A_{k-1}) \models \varphi_{/A_k}$$

La formule $\varphi_{/A_k}$ est appelée la *formule quotient* de φ par A_k. De manière similaire, nous pouvons quotienter par A_{k-1}... puis par A_1 et nous nous ramenons alors à tester que $\mathsf{Nil} \models \varphi_{/A_k \ldots /A_1}$ où Nil est un processus qui ne fait rien.

Cette approche compositionnelle a été étudiée principalement pour éviter le problème de l'explosion combinatoire qui apparaît dans des algorithmes d'analyse en avant. Dans certains cas, il s'est avéré que cette approche était assez efficace [LL98].

Ce genre de techniques a été étudié pour les systèmes temporisés par exemple dans [LL95, LPY95]. Le langage de spécification L_ν s'est avéré bien adapté à cette technique, car c'est un langage *compositionnel*, c'est-à-dire que les formules quotientées décrites plus haut appartiennent au langage L_ν dès que la formule initiale est dans L_ν. Ces techniques ont été implémentées dans l'outil de model-checking CMC [LL98], puis étendues aux automates hybrides dans HCMC [CL00a].

7.2.5 Bilan et extensions

Il existe de nombreux algorithmes de model-checking pour les automates temporisés pour différents langages de spécification. Nous en avons décrits quelques-uns ici, mais il y en a bien d'autres, par exemple pour d'autres logiques temporelles temporisées que TCTL. Le survey [AH91] et les articles [AH93, AH94] en présentent certaines et proposent des algorithmes de model-checking adaptés à ces logiques. Dans [BTY97], un algorithme de model-checking est proposé pour une extension de TCTL, cet algorithme ne fait pas les calculs en arrière comme dans la partie 7.2.3, mais en avant.

Nous avons pu aussi constater que l'étude d'algorithmes de vérification pour les propriétés d'accessibilité était souvent privilégiée. Nous pouvons alors faire la remarque suivante à propos des algorithmes en avant et en arrière présentés dans les parties précédentes.

Remarque : Calculer le Post^* ou le $\mathsf{Pré}^*$ d'une configuration revient en fait à construire un graphe de « simulation » du système initial. Dans [BFH90, BFHR92], un algorithme de minimisation de la taille du graphe généré est proposé. Cet algorithme permet de construire un graphe de taille minimale bisimilaire au système initial en « découpant » au minimum l'ensemble des configurations possibles. Dans le cas des automates temporisés, l'automate des régions est un exemple de tel graphe de simulation, mais ce n'est pas en général le graphe le plus petit. Une application de ces algorithmes de minimisation aux automates temporisés et au model-checking de TCTL est présentée dans [ACD+92b]. Dans [ACD+92a], l'implémentation de trois tels algorithmes de minimisation pour les automates temporisés est proposée. Dans [TY01], un algorithme de génération de graphe est aussi proposé pour trois simulations qui abstraient le temps, ce qui permet d'appliquer des méthodes de vérification non temporisées au graphe généré.

Dans un genre un peu différent, les travaux [CJ98, CJ99] ont montré que la relation d'accessibilité était exprimable dans la logique de Presburger, c'est-à-dire que pour tous les états q et q', il existe une formule de Presburger $\varphi_{q,q'}$ telle que pour toutes les valuations v et v',

$$(v, v') \models \varphi_{q,q'} \iff (q', v') \text{ est accessible à partir de } (q, v)$$

La preuve de ce résultat est extrêmement compliquée.

7.3 Les outils de model-checking

Dans tout ce qui précède, nous avons évoqué de temps en temps différents outils de model-checking pour les systèmes temporisés. Nous allons maintenant en présenter brièvement quatre, à savoir CMC, HyTech, Kronos et Uppaal.

7.3.1 CMC

L'outil CMC (CMC signifie *Compositional Model-Checking*) a été développé au Laboratoire Spécification et Vérification à l'ENS de Cachan par François Laroussinie. Cet outil vise à vérifier des réseaux d'automates temporisés. Il utilise comme langage de spécification la logique L_ν que nous avons présentée dans la partie 7.1.2. L'algorithme qui est implémenté est l'approche compositionnelle présentée dans la partie 7.2.4. Des stratégies de simplifications de formules ont été aussi étudiées. Ces travaux se trouvent dans les articles [LL95, KLL+97, LL98]. La page de l'outil CMC est

`http://www.lsv.ens-cachan.fr/~fl/cmcweb.html`

7.3.2 HyTech

L'outil HyTech (HyTech signifie HYbrid TECHnology Tool) a été développé à l'université de Berkeley par Tom Henzinger, Pei-Hsin Ho et Howard Wong-Toi. Cet outil vise à vérifier des réseaux d'automates hybrides très généraux. Il permet même d'analyser des systèmes paramétrés. Il vérifie principalement des propriétés d'accessibilité et de sûreté. La représentation symbolique qui est utilisée dans HyTech est celle des polyèdres convexes, c'est-à-dire des intersections d'hyperplans. Les algorithmes implémentés sont des algorithmes de calculs de Post^* et $\mathsf{Pré}^*$. Le guide d'utilisation de HyTech est [HHWT95b] alors qu'une description de l'outil peut aussi se trouver dans [HHWT95a, HHWT97]. La page de l'outil HyTech est

`http://www-cad.eecs.berkeley.edu/~tah/HyTech/`

7.3.3 Kronos

L'outil Kronos a été développé au laboratoire Vérimag à Grenoble principalement par Sergio Yovine, Alfredo Olivero, Conrado Daws et Stavros Tripakis. Cet outil vise à vérifier des réseaux d'automates temporisés. Il permet de vérifier des propriétés énoncées dans la logique TCTL présentée dans la partie 7.1.2. Dans cet outil, plusieurs algorithmes sont implémentés, tout d'abord l'analyse en avant, puis l'analyse en arrière, enfin l'algorithme pour TCTL décrit à la partie 7.2.3. De multiples articles parlent de Kronos, par exemple [OY93, BTY97, Daw97, Yov97, BDM+98, Yov98]. La page de l'outil se trouve à l'adresse

`http://www-verimag.imag.fr/TEMPORISE/kronos/`

7.3.4 Uppaal

L'outil Uppaal a été développé conjointement par l'université d'Aalborg au Danemark et l'université d'Uppsala en Suède. Les principales personnes ayant participé à son développement sont Wang Yi, Kim G. Larsen, Paul Pettersson, Johan Bengtsson, Gerd Behrmann, Kåre J. Kristoffersen. Il permet de vérifier des réseaux d'automates temporisés avec variables entières bornées, actions urgentes, etc... Les propriétés qui peuvent être vérifiées sont principalement des propriétés d'accessibilité, de vivacité (ces propriétés sont apparues dans les dernières versions d'Uppaal [BLL+98]) ou d'états bloquants. La logique utilisée par Uppaal est un fragment de TCTL qui n'autorise pas les emboîtements d'opérateurs temporels. L'algorithme implémenté dans Uppaal est essentiellement un algorithme d'analyse en avant, avec quelques variantes, dont l'utilisation d'approximations. De nombreux articles décrivent l'outil, par exemple [BLL+95, BL96, LPY97, HSLL97, BLL+98, ABB+01]. La page de l'outil est la suivante

`http://www.uppaal.com/`

Remarque : Les gros avantages d'Uppaal par rapport aux autres outils que nous avons décrits avant sont d'une part son interface graphique, très conviviale et d'autre part, son module de simulation qui permet, lors de la phase de modélisation de faire des tests du modèle et de détecter ainsi éventuellement des erreurs dans la modélisation.

Tous les outils dont nous venons de parler ont été « validés » par des études de cas convaincantes. Nous renvoyons aux pages web des outils pour la description de ces études de cas.

7.4 Ce que nous allons faire

L'algorithme implémentable que nous présentons pour le modèle des automates temporisés avec mises à jour est un algorithme d'analyse en avant (*cf* la partie 7.2.1). Cet algorithme utilise une approximation du type de celles étudiées dans [DT98]. Nous décrirons une implémentation possible de cet algorithme en utilisant une structure de données que nous avons évoquée dans ce chapitre, les DBM (*Difference Bounded Matrice*). Ce travail est présenté dans le chapitre 8.

L'étude des automates de test que nous menons au chapitre 9 se situe dans plusieurs des points dont nous avons parlé dans ce chapitre. Pour bien s'en rendre compte, nous en rappelons brièvement le principe. Étant donnée une propriété ϕ, nous construisons un automate (de test) T_ϕ qui vérifie la propriété suivante : pour tout automate temporisé $\mathcal{A}$,

$$\mathcal{A} \models \phi \iff (\mathcal{A} \parallel T_\phi) \text{ ne permet pas d'atteindre un état rejetant de } T_\phi$$

Nous proposons un langage logique, proche de L_ν, qui est testable de cette façon et qui caractérise en outre l'ensemble des propriétés qui se testent de cette façon. Pour montrer cela, des techniques de model-checking compositionnel (*cf* la partie 7.2.4) seront utilisées, car la propriété ϕ ci-dessus peut apparaître comme le quotient de la propriété d'accessibilité dans la composition parallèle par l'automate T_ϕ. En outre, nous avons vu dans la partie 7.3.4, que l'outil UPPAAL permet essentiellement de vérifier des propriétés d'accessibilité. Avec ces résultats, nous avons alors caractérisé l'ensemble des propriétés que UPPAAL « est capable » de vérifier.

Dans le chapitre 10, nous menons une étude de cas sur un exemple réel de protocole de communication. La modélisation que nous proposons sera ensuite implémentée dans l'outil UPPAAL. Celui-ci nous permettra de réaliser des simulations et des vérifications.

Chapitre 8

Une approche algorithmique des automates temporisés avec mises à jour

Ce chapitre provient de l'article [Bou01].

Même si le modèle général des automates temporisés avec mises à jour est indécidable, de nombreuses sous-classes ont été montrées décidables (voir le chapitre 3). Les résultats théoriques de décidabilité résultent d'une extension de la construction de l'automate des régions décrite dans [AD90, AD94] et rappelée dans la partie 2.3.2. Comme nous l'avons déjà dit à plusieurs reprises, même dans le cas des automates temporisés classiques, cette construction n'est pas utilisée pour l'implémentation, mais des outils tels que UPPAAL et KRONOS utilisent un algorithme d'analyse en avant, l'*algorithme des zones*. Cet algorithme calcule à la volée une sur-approximation (ou une abstraction suivant la terminologie de [DT98]) de l'ensemble des états accessibles, c'est-à-dire des paires (q, Z) où q est un état de contrôle et Z une zone. L'un des avantages majeurs des zones est qu'elles peuvent être facilement implémentées en utilisant des structures de données telles que les DBMs [Dil89] ou les CDDs [BLP$^+$99, LWYP99].

Notre but est ici de proposer un algorithme d'accessibilité implémentable pour les automates temporisés avec mises à jour. Il s'agit d'un algorithme d'analyse en avant qui est une extension de l'algorithme des zones. Après avoir décrit cet algorithme, nous proposons une description d'une implémentation qui utilise aussi des DBMs comme structure de données, c'est-à-dire que nous montrons que toutes les opérations sur les zones qui apparaissent dans l'algorithme peuvent se calculer en utilisant les DBMs. Nous proposons de plus une preuve complète de la correction de cet algorithme. Cette preuve utilise différentes propriétés des DBMs et des régions. Ce résultat a pour principale conséquence la correction de l'algorithme des zones pour les automates classiques. À notre connaissance, bien que ce dernier algorithme soit très connu, nous n'en avons trouvé aucune preuve complète et correcte dans la littérature.

Nous commençons ce chapitre par décrire précisément l'algorithme des zones que nous avons évoqué dans la partie 7.2.1 pour les automates temporisés classiques et nous montrons comment les DBMs sont utilisés pour son implémentation. Après cette brève présentation, nous revenons au modèle des automates temporisés avec mises à jour et présentons notre algorithme. Le reste du chapitre est dévolu à une possible implémentation de notre algorithme en utilisant aussi les DBMs ainsi qu'à une preuve complète de sa correction.

Dans tout ce chapitre, nous considérons des automates temporisés avec uniquement des constantes entières, comme nous autorise à le faire le lemme 3.13 page 55.

8.1 Automates temporisés classiques, état de l'art

Le principe général de l'analyse en avant a été décrit dans la partie 7.2.1. Dans cette partie, nous allons maintenant présenter de manière précise l'algorithme des zones, algorithme implémenté dans certains outils pour la vérification des automates temporisés classiques. Nous allons aussi expliquer comment l'implémentation utilise les DBMs comme structure de données.

8.1.1 Zones

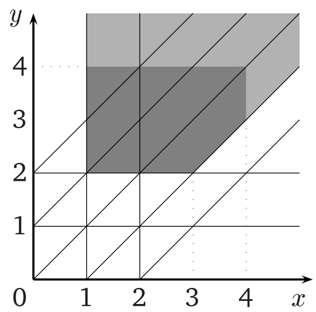
Une *zone* est un sous-ensemble de $\mathbb{T}^n$ défini par une contrainte d'horloges. Soit k une constante. Une zone est *k-bornée* si elle peut être définie par une contrainte k-bornée (voir page 30). Soit Z une zone. La *k-approximation* de Z est définie comme étant la plus petite zone k-bornée Z_k telle que $Z \subseteq Z_k$. Nous notons $\mathsf{Approx}_k(Z) = Z_k$.

***Exemple* 8.1** Considérons la zone Z dessinée en gris foncé sur la figure de droite : Z est définie par la contrainte

$$1 < x < 4 \wedge 2 < y < 4 \wedge x - y < 1.$$

Prenant $k = 2$, la k-approximation de Z est dessinée en gris clair (et gris foncé) ; elle est définie par la contrainte

$$1 < x \wedge 2 < y \wedge x - y < 1.$$



La k-approximation peut être utilisée dans des algorithmes d'analyse en avant (voir la partie 7.2.1) pour que les calculs terminent. Considérons par exemple l'automate du contre-exemple 7.9, page 151. Nous avons vu que le calcul exact d'une analyse en avant ne terminait pas sur cet automate. Il est facile de voir que si l'on utilise l'abstraction[1] donnée par la 1-approximation (c'est-à-dire si l'on calcule l'itérée de $\mathsf{Approx}_1 \circ \mathsf{Post}$ au lieu de l'itérée de Post, *cf* la partie 7.2.1), alors les calculs terminent.
C'est le rôle principal que vont jouer les surapproximations dans ce chapitre.

8.1.2 L'algorithme

Nous avons vu que le calcul exact des états accessibles par un calcul en avant pouvait ne pas terminer (partie 7.2.1) et qu'il était alors possible de pallier cette difficulté en calculant non pas les états accessibles, mais une surapproximation de ceux-ci. Bien sûr, comme nous l'avons déjà dit, cette surapproximation doit rester adaptée au problème que nous voulons résoudre, à savoir le test d'accessibilité : la surapproximation doit être suffisamment fine pour que nous ne calculions pas d'états qui ne sont pas accessibles dans le système réel. Dans [DT98], plusieurs surapproximations (alors appelées abstractions) sont présentées, dont celle dont nous allons parler ici.

Soit $\mathcal{A}$ un automate temporisé classique. Si $\mathtt{e} = (\mathtt{q} \xrightarrow{\mathtt{g,a,C:=0}} \mathtt{q}')$ est une transition de $\mathcal{A}$ et si $\mathtt{Z}$ est une zone, alors $\mathtt{Post}(\mathtt{Z},\mathtt{e})$ représente l'ensemble $[\mathtt{C} \leftarrow 0](\mathtt{g} \cap \overrightarrow{\mathtt{Z}})$ où $\overrightarrow{\mathtt{Z}}$ représente le *futur* de $\mathtt{Z}$ et est défini par

$$\overrightarrow{\mathtt{Z}} = \{v + t \mid v \in \mathtt{Z} \text{ et } t \geq 0\}$$

[1] selon la terminologie de [DT98]

$\texttt{Post}(\texttt{Z},\texttt{e})$ est l'ensemble des valuations qui peuvent être atteintes à partir de Z en attendant dans l'état courant, q, puis en franchissant e. Nous associons à $\mathcal{A}$ la plus grande constante k apparaissant dans $\mathcal{A}$ (*i.e.* la plus grande constante c telle qu'il y a une contrainte $x \sim c$ pour une horloge x ou $x - y \sim c$ pour des horloges x et y). Une constante maximale pourrait être calculée pour chaque horloge x (de la même façon), mais pour ce que nous faisons, cela ne change rien, la présentation en est juste moins compliquée.

L'algorithme des zones pour les automates classiques est alors présenté comme l'algorithme 8.1 ($\texttt{Z}_0$ représente la zone initiale, c'est souvent la valuation **0** où toutes les horloges valent zéro).

Algorithme 8.1 Algorithme des zones pour les automates temporisés classiques

```
Algorithme des zones (𝒜:TA) {
  Définir k;
  Visités := ∅;                    (* Visités contient les états visités *)
  Attente := {(q0,Approx_k(Z0))};
  Répéter
    Prendre (q,Z) dans Attente et le retirer;
    Si q est final alors {Retourner «~Oui~»;}
                 sinon {S'il n'y pas de (q,Z')∈ Visités t.q. Z ⊆ Z'
                        alors {Visités := Visités ∪ {(q,→Z)};
                               Successeur := {(q',Approx_k(Post(Z,e))) |
                                                     e transition de q à q'};
                               Attente := Attente ∪ Successeur;}}
  Jusqu'à (Attente = ∅);
  Retourner «~Non~»; }
```

Cet algorithme calcule pas à pas une sur-approximation de l'ensemble des états accessibles et teste si l'approximation intersecte ou non l'ensemble des états finals. Ainsi, si la réponse de l'algorithme est « Non », alors aucun état final ne peut être atteint. Cependant, si la réponse est « Oui », il peut *a priori* arriver que la sur-approximation intersecte les états finals alors que l'ensemble des états accessibles ne les intersecte pas. Nous discuterons de la correction de cet algorithme en détails dans la partie 8.3.

8.1.3 L'implémentation : les DBMs

Pour implémenter l'algorithme 8.1, nous avons besoin d'une structure de données pour représenter les zones et cette structure de données doit permettre de tester l'inclusion de deux zones et de calculer aisément les différentes opérations utilisées dans l'algorithme, c'est-à-dire l'intersection de deux zones, le futur d'une zone, l'image d'une zone par une remise à zéro et la k-approximation d'une zone. Des outils comme UPPAAL ou KRONOS utilisent la structure de données proposée par David Dill dans [Dil89], les DBMs. Une présentation assez détaillée de cette structure de données peut être trouvée dans [CGP99].

Une *matrice de différences bornées*, ce que nous abrégerons en *DBM* (qui signifie « Difference Bounded Matrice »), pour n horloges est une matrice carrée $(m_{i,j}, \prec_{i,j})_{i,j=0\ldots n}$ de taille $n+1$ de paires

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

Une DBM $M = (m_{i,j}, \prec_{i,j})_{i,j=0\ldots n}$ définit l'ensemble suivant de $\mathbb{T}^n$ (l'horloge x_0 est supposée être constamment égale à zéro, *i.e.* pour toute valuation v, $v(x_0) = 0$) :

$$\{v : \{x_1, \dots, x_n\} \longrightarrow \mathbb{T} \mid \forall\, 0 \le i, j \le n,\ v(x_i) - v(x_j) \prec_{i,j} m_{i,j}\}$$

où $\gamma < \infty$ signifie que γ est un réel quelconque.

Ce sous-ensemble de $\mathbb{T}^n$ est une zone et est noté $[\![M]\!]$. Chaque DBM sur n horloges représente une zone de $\mathbb{T}^n$. Remarquons que plusieurs DBMS peuvent représenter la même zone.

***Exemple* 8.2** La zone définie par les équations $x_1 > 3 \wedge x_2 \le 5 \wedge x_1 - x_2 < 4$ peut être représentée par les deux DBMs

$$\begin{pmatrix} (0;\le) & (-3;<) & (\infty;<) \\ (\infty;<) & (0;\le) & (4;<) \\ (5;\le) & (\infty;<) & (0;\le) \end{pmatrix} \quad \text{et} \quad \begin{pmatrix} (\infty;<) & (-3;<) & (\infty;<) \\ (\infty;\le) & (\infty;<) & (4;<) \\ (5;\le) & (\infty;<) & (0;\le) \end{pmatrix}$$

Ainsi, les DBMs ne sont pas une représentation canonique des zones. De plus, il n'est pas possible de tester de manière syntaxique si $[\![M]\!] = \varnothing$ ou $[\![M_1]\!] = [\![M_2]\!]$. Une forme normale pour ces DBMs a donc été définie. Son calcul utilise l'algorithme de Floyd et plusieurs réécritures syntaxiques (voir par exemple [Dil89, CGP99] pour une description de ce calcul). Dans ce qui suit, nous noterons $\phi(M)$ la *forme normale* de M. Avant d'établir quelques propriétés très importantes des formes normales, nous définissons un ordre total sur $\mathbb{V}$ de la manière suivante : si $(m;\prec), (m';\prec') \in \mathbb{V}$, alors

$$(m;\prec) \le (m';\prec') \iff \begin{cases} m < m' \\ \text{ou} \\ m = m' \text{ et soit } \prec = \prec' \text{ ou } \prec' = \le . \end{cases}$$

avec pour tout $m \in \mathbb{Z}, m < \infty$. Nous définissons alors $>$, $\ge$ et $<$ de manière naturelle.

Ces ordres sont étendus en des ordres partiels sur les DBMs. Soient $M = (m_{i,j};\prec_{i,j})_{i,j=0\dots n}$ et $M' = (m'_{i,j};\prec'_{i,j})_{i,j=0\dots n}$ deux DBMs, alors

$$M \le M' \iff \text{pour tous } i, j = 0 \dots n,\ (m_{i,j};\prec_{i,j}) \le (m'_{i,j};\prec'_{i,j}).$$

Nous pouvons maintenant établir quelques propriétés très utiles des formes normales. Si M et M' sont des DBMs, alors :

(i) $[\![M]\!] = [\![\phi(M)]\!]$ et $\phi(M) \le M$,

(ii) $[\![M]\!] \subseteq [\![M']\!] \iff \phi(M) \le M' \iff \phi(M) \le \phi(M')$.

Le dernier point exprime le fait que le test d'inclusion pour les zones peut se faire syntaxiquement sur les formes normales des DBMs (qui représentent les zones bien sûr).

En outre, les formes normales des DBMs peuvent être caractérisées d'une manière naturelle. Si $M = (m_{i,j};\prec_{i,j})_{i,j=0\dots n}$ est une DBM telle que $[\![M]\!] \ne \varnothing$, alors les deux propriétés suivantes sont équivalentes :

(i) M est en forme normale,

(ii) pour chaque $i, j = 0 \dots n$, pour chaque réel $-m_{j,i} \prec_{j,i} r \prec_{i,j} m_{i,j}$, il existe une valuation $v \in [\![M]\!]$ telle que $v(x_j) - v(x_i) = r$ (en supposant toujours que $v(x_0) = 0$).

Cette propriété exprime le fait que si une DBM est sous forme normale, aucune de ses contraintes ne peut être resserrée[2] en utilisant l'algorithme de Floyd. Cette propriété sera utilisée dans la partie 8.3.4.

[2] Ceci signifie qu'il n'y a aucun moyen de remplacer un des coefficients de la DBM par une valeur plus petite sans modifier la zone définie.

Calcul de quelques opérations sur les DBMs.

Comme nous le disions au début de cette partie, la structure de données utilisée pour représenter les zones doit aussi être appropriée pour calculer les quatre opérations sur les zones qui sont utilisées dans l'algorithme 8.1, plus précisément le futur, l'intersection, l'image par une remise à zéro et la k-approximation. Ces opérations sur les DBMs sont présentées en détails dans [CGP99], nous ne faisons que les rappeler ici.

Intersection. Supposons que $M = (m_{i,j}; \prec_{i,j})_{i,j=1\dots n}$ et $M' = (m'_{i,j}; \prec'_{i,j})_{i,j=1\dots n}$ soient deux DBMs en forme normale. Alors, définissant $M'' = (m''_{i,j}; \prec''_{i,j})_{i,j=1\dots n}$ par

$$\text{pour tout } i,j = 1\dots n, \quad (m''_{i,j}; \prec''_{i,j}) = \min((m_{i,j}; \prec_{i,j}), (m'_{i,j}; \prec'_{i,j}))$$

Nous obtenons alors que $[\![M'']\!] = [\![M]\!] \cap [\![M']\!]$. Notons qu'il peut arriver que M'' ne soit pas en forme normale.

Futur. Supposons que $M = (m_{i,j}; \prec_{i,j})_{i,j=1\dots n}$ soit une DBM en forme normale. Nous définissons la DBM $\overrightarrow{M} = (m'_{i,j}; \prec'_{i,j})_{i,j=1\dots n}$ par :

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

Nous obtenons alors que $[\![\overrightarrow{M}]\!] = \overrightarrow{[\![M]\!]}$. De plus, la DBM $\overrightarrow{M}$ est en forme normale.

Image par remise à zéro. Supposons que $M = (m_{i,j}; \prec_{i,j})_{i,j=1\dots n}$ soit une DBM en forme normale. Nous définissons la DBM $M_{x_k:=0} = (m'_{i,j}; \prec'_{i,j})_{i,j=1\dots n}$ par :

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

Nous obtenons que $[\![M_{x_k:=0}]\!] = [x_k \leftarrow 0][\![M]\!]$. De plus, la DBM $M_{x_k:=0}$ est en forme normale.

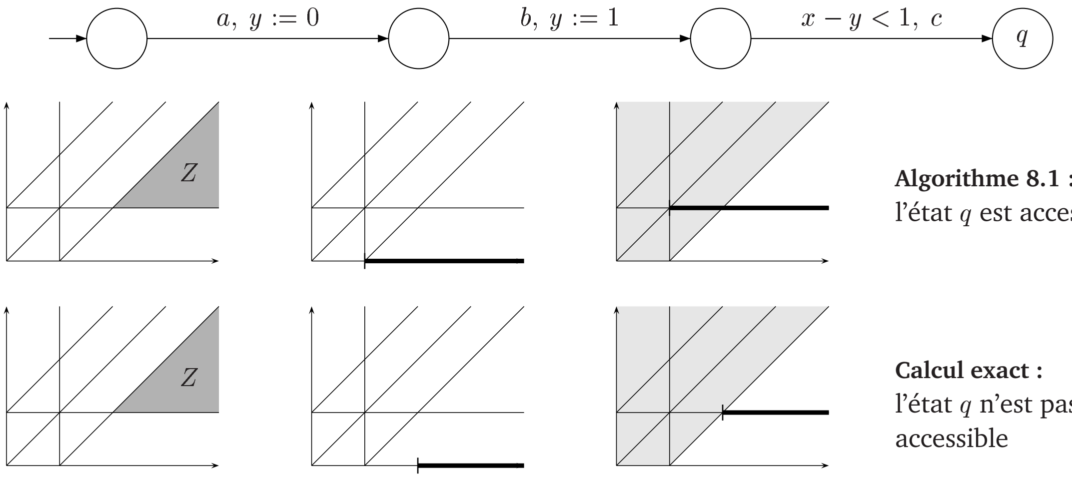
k-approximation. Supposons que $M = (m_{i,j}; \prec_{i,j})_{i,j=1\dots n}$ soit une DBM en forme normale. Nous définissons la DBM $M_k = (m'_{i,j}; \prec'_{i,j})_{i,j=1\dots n}$ par :

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

Nous obtenons que $[\![M_k]\!] = \mathsf{Approx}_k([\![M]\!])$, mais M_k n'est pas nécessairement en forme normale.

8.2 Retour aux automates temporisés avec mises à jour

L'algorithme 8.1 ne s'adapte pas aussi facilement aux automates temporisés avec mises à jour : l'approximation utilisée pour les automates temporisés classiques (celle utilisant les constantes maximales apparaissant dans l'automate) n'est pas correcte pour nos automates temporisés avec mises à jour. Par exemple, considérons l'automate temporisé suivant où nous autorisons une unique mise à jour spéciale, $y := 1$. La constante maximale apparaissant dans cet automate est 1.



Algorithme 8.1 :
l'état q est accessible

Calcul exact :
l'état q n'est pas accessible

Les dessins ci-dessus décrivent d'une part ce que l'algorithme 8.1 calcule et d'autre part le calcul exact des états accessibles, en partant dans les deux cas de la zone Z définie par la contrainte d'horloges $y > 1 \wedge x - y > 1$. Le dernier état de l'automate n'est donc pas accessible alors que l'algorithme 8.1 le dit accessible : l'approximation utilisée dans l'algorithme n'est pas correcte vis-à-vis de l'accessibilité (sur la troisième colonne des deux calculs, la zone $x - y < 1$ est représentée en gris clair).

Avant de présenter notre algorithme, nous rappelons nos résultats précis de décidabilité obtenus dans la partie 3.3. Ces résultats, ainsi que d'autres définitions vont être nécessaires dans la suite de ce chapitre.

8.2.1 Rappels sur les résultats de décidabilité

Nous reprenons la définition de régions donnée dans la partie 3.3 de la page 61, celle de la page 55 en étant un cas particulier. Ces derniers ensembles de régions seront appelés des ensembles de régions non diagonaux.

Nous distinguons à nouveau les ensembles de contraintes non diagonales et les ensembles de contraintes générales.

Contraintes non diagonales

Soient $\mathcal{C} \subseteq \mathcal{C}_{df}(X)$ un ensemble de contraintes non diagonales et $\mathcal{U} \subseteq \mathcal{U}(X)$ un ensemble de mises à jour tel que

$$up = \bigwedge_{x \in X} up_x \in \mathcal{U} \implies \text{pour tout } x,\ up_x \in \{det_x, inf_x, sup_x, int_x\} \text{ où :} \qquad (\diamondsuit_{df})$$

$$\begin{cases} det_x ::= x := c \mid x := z + d \text{ avec } c \in \mathbb{N},\ d \in \mathbb{Z} \text{ et } z \in X \\ inf_x ::= x :\lhd c \mid x :< z + d \mid inf_x \wedge inf_x \text{ avec } \lhd \in \{<, \leq\},\ c \in \mathbb{N},\ d \in \mathbb{Z} \text{ et } z \in X \\ sup_x ::= x :\rhd c \mid x :> z + d \mid sup_x \wedge sup_x \text{ avec } \rhd \in \{>, \geq\},\ c \in \mathbb{N},\ d \in \mathbb{Z} \text{ et } z \in X \\ int_x ::= x :\in (c; d) \mid x :\in (c; z + d) \mid x :\in (z + c; d) \mid x :\in (z + c; z + d) \\ \qquad\qquad \text{où (et) sont soit [soit], } z \text{ est une horloge et } c,\ d \text{ sont dans } \mathbb{Z}. \end{cases}$$

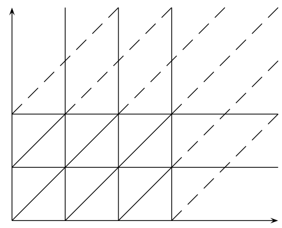
Si le système Diophantien d'inéquations linéaires sur les variables $(\max_x)_{x \in X}$

$$\{c \leq \max_x \mid x \sim c \in \mathcal{C} \text{ ou } x :\sim c \in \mathcal{U}\} \cup \{\max_z \leq \max_y + c \mid z :\sim y + c \in \mathcal{U}\} \qquad (\mathcal{S}_{df})$$

a une solution, alors la classe d'automates $Aut(\mathcal{C},\mathcal{U})$ est décidable. De plus, si $\alpha = (\max_x)_{x\in X}$ est solution du système, et si $\mathcal{A} \in Aut(\mathcal{C},\mathcal{U})$, alors nous pouvons décider du vide de $L(\mathcal{A})$ grâce à l'automate des régions $\Gamma_{\mathcal{R}_\alpha}(\mathcal{A})$ (α est vu ici comme un ensemble de régions non diagonales). Notons bien qu'il est possible de décider de l'existence d'un tel α et que s'il existe, il peut être calculé (voir [Dom91]).

À partir de l'ensemble de régions α que nous venons de construire, nous définissons un autre ensemble de régions, $\beta = ((\max'_x)_{x\in X}; (\max'_{x,y})_{x,y\in X})$ où pour toutes les horloges $x,y \in X$, $\max'_x = \max_x$ et $\max'_{x,y} = \max_x$.

***Exemple* 8.3** Les ensembles de régions α et β peuvent être représentés comme sur la figure de droite, où α est dessiné en lignes pleines alors que son raffinement β utilise aussi les lignes en pointillés.



Contraintes générales

Soient $\mathcal{C} \subseteq \mathcal{C}(X)$ un ensemble fini de contraintes générales et $\mathcal{U} \subseteq \mathcal{U}(X)$ un ensemble fini de mises à jour telles que :

$$up = \bigwedge_{x\in X} up_x \in \mathcal{U} \quad \Longrightarrow \quad \forall x \in X,\ up_x \in \begin{array}{l} \{x := c, x :< c, x :\leq c \mid c \in \mathbb{N}\} \\ \cup\ \{x := y \mid y \in X\} \end{array} \qquad (\lozenge_{gen})$$

Soit $\alpha = ((\max_x)_{x\in X}; (\max_{x,y})_{x,y\in X})$ une solution du système Diophantien d'inéquations linéaires suivant :

$$\begin{array}{ll} & \{c \leq \max_x \mid x \sim c \in \mathcal{C}\} \\ \cup & \{c \leq \max_{x,y} \mid x - y \sim c \in \mathcal{C}\} \\ \cup & \{c \leq \max_x,\ \max_z \geq c + \max_{z,x} \mid x :< c \text{ ou } x :\leq c \text{ ou } x := c\ \in \mathcal{U},\ \text{et } z \in X\} \\ \cup & \{\max_x \leq \max_y,\ \max_{z,y} \geq \max_{z,x},\ \max_{x,z} \leq \max_{x,y} \mid x := y \in \mathcal{U} \text{ et } z \in X\} \end{array} \qquad (\mathcal{S}_{gen})$$

Alors le vide de tout automate de $Aut(\mathcal{C},\mathcal{U})$ peut être testé en construisant l'automate des régions $\Gamma_{\mathcal{R}_\alpha}(\mathcal{A})$.

À partir de α, nous définissons l'ensemble de régions $\beta = ((\max'_x)_{x\in X}; (\max'_{x,y})_{x,y\in X})$ par :

$(\max'_x)_{x\in X}$ sont les plus petites constantes telles que $\left\{ \begin{array}{l} \max'_x \geq \max_x \\ x := y \in \mathcal{U} \implies \max'_x = \max'_y \end{array} \right.$

et pour toutes les horloges $x,y \in X$, $\max'_{x,y} = \max'_x$.

Comme dans le cas des automates temporisés avec contraintes non diagonales, l'ensemble des régions décrit par β est un raffinement de celui décrit par α.

Dans tout ce qui suit, nous considérerons uniquement des automates temporisés qui appartiennent à une classe d'automates décidable décrite ci-avant. Nous associons à chaque tel automate deux ensembles de régions α et β comme nous l'avons décrit dans cette partie.

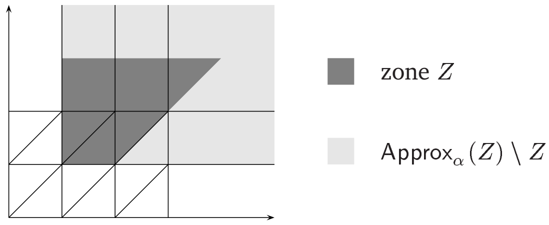
8.2.2 Zones

Nous complétons les définitions de la partie 8.1.1.

Soit $\mathcal{R}_\alpha$ un ensemble de régions. Chaque union convexe de régions de $\mathcal{R}_\alpha$ est une zone, l'ensemble de ces zones est noté $\mathcal{Z}_\alpha$. Si Z est une zone, nous définissons $\mathsf{Approx}_\alpha(Z)$ la plus petite zone de

$\mathcal{Z}_\alpha$ qui contient Z. Nous disons que $\mathsf{Approx}_\alpha(Z)$ est l'*α-approximation* de Z. La k-approximation d'une zone que nous avons définie dans la partie 8.1.1 est un cas particulier d'α-approximation où $\alpha = (k, \ldots, k)$.

***Exemple* 8.4** Considérons par exemple l'ensemble de régions non diagonal $\mathcal{R}_\alpha$ dessiné à droite ($\max_x = 3$ et $\max_y = 2$) et la zone Z en gris foncé. Son α-approximation $\mathsf{Approx}_\alpha(Z)$ est alors dessinée en gris clair.



8.2.3 L'algorithme

L'algorithme que nous proposons pour tester le vide des automates temporisés avec mises à jour décidables est une extension de l'algorithme 8.1 pour les automates temporisés classiques. Nous considérons donc un automate temporisé avec mises à jour $\mathcal{A}$ qui appartient à une classe décidable $Aut(\mathcal{C},\mathcal{U})$. Notre algorithme est présenté comme l'algorithme 8.2.

Algorithme 8.2 Algorithme des zones pour les automates temporisés avec mises à jour décidables

```
Algorithme des zones (𝒜:UTA) {
  Calculer β en utilisant 𝒰 et 𝒞 ;
  Visités := ∅;                    (* Visités contient les états visités *)
  Attente := {(q0,Approxβ(Z0))};
  Répéter
    Prendre (q,Z) dans Attente et le Retirer;
    Si q est final alors {Retourner «~Oui~»;}
                   sinon {S'il n'y a pas de (q,Z')∈ Visités t.q. Z ⊆ Z'
                          alors {Visités := Visités ∪ {(q, Z⃗)};
                                 Successeur := {(q',Approxβ(Post(Z,e))) |
                                                   e transition de q à q'};
                                 Attente := Attente ∪ Successeur;}}
  Jusqu'à (Attente = ∅);
  Retourner «~Non~»; }
```

Cet algorithme est très semblable à celui pour les automates temporisés classiques, la seule différence résidant dans le calcul de β, l'ensemble des régions utilisé pour calculer les approximations. Nous pouvons remarquer que si $\mathcal{A}$ est un automate temporisé classique, alors, définissant $\beta = (k, \ldots, k)$, nous obtenons qu'un tel β pourrait être associé à $\mathcal{A}$ comme dans la partie 8.2.1 et que $\mathsf{Approx}_k = \mathsf{Approx}_\beta$. L'algorithme 8.2 apparaît donc bien comme une extension de l'algorithme 8.1 pour les automates temporisés avec mises à jour.

Comme Approx_β est une sur-approximation, l'algorithme 8.2 calcule, tout comme l'algorithme 8.1, une surapproximation de l'ensemble d'états accessibles. Ainsi, il nous faut maintenant étudier deux problèmes principaux :

- nous devons étudier la **correction** de l'algorithme 8.2 : nous souhaitons montrer que si l'algorithme 8.2 répond qu'un état est accessible, alors il est vraiment accessible. L'étude de la correction sera présentée en détails dans la partie 8.3.
- dans l'algorithme 8.2, de nouvelles opérations sur les zones sont apparues, par exemple l'image d'une zone par une mise à jour ou bien la surapproximation Approx_β d'une zone. Nous présen-

terons dans la partie 8.2.4 une **implémentation** possible de cet algorithme en utilisant aussi les DBMs comme structure de données.

Remarque : L'ensemble exact des états accessibles peut être calculé si aucune sur-approximation des zones n'est utilisée. Cependant, les calculs peuvent ne pas terminer. Dans [MP99], des sous-classes des automates temporisés classiques, pour lesquels la terminaison est assurée, sont décrites. Notre but, ici, est très différent, nous ne sommes pas intéressés par le calcul exact des états accessibles, mais par la décision du problème de l'accessibilité (d'un état de contrôle).

8.2.4 Implémentation de l'algorithme

Pour montrer que la structure de données des DBMs est adaptée à l'implémentation de l'algorithme 8.2, nous décrivons comment il est possible de calculer les nouvelles opérations sur les zones qui apparaissent dans l'algorithme 8.2 en utilisant les DBMs. Les nouvelles opérations sont de deux types distincts : image par une mise à jour et sur-approximation Approx_β.

Image par une mise à jour. Soit up une mise à jour et M une DBM (en forme normale). La DBM M représente une contrainte d'horloges φ sur l'ensemble d'horloges X. À partir de up, nous construisons une contrainte φ_{up} sur l'ensemble d'horloges $X \cup X'$ (où X' est une copie disjointe de X) par induction sur up de la manière suivante :

$$\begin{cases} \varphi_{x:\sim c} = x' \sim c \\ \varphi_{x:\sim y+c} = x' \sim y + c \\ \varphi_{up_1 \wedge up_2} = \varphi_{up_1} \wedge \varphi_{up_2}. \end{cases}$$

Nous définissons maintenant la formule $\varphi'_{up} = \varphi \wedge \varphi_{up}$. Cette formule représente la mise à jour up dans le sens où si V est une valuation des horloges $X \cup X'$, alors

$$V \models \varphi'_{up} \iff V_{|X} \models \varphi \text{ et } V_{|X'} \in up(V_{|X}).$$

Nous supposons que $up = \bigwedge_{i=1}^n up_i$ où up_i est une mise à jour de l'horloge x_i. Nous définissons $N = (\eta_{i,j})_{i,j=0\dots 2n}$ comme étant la DBM de taille $2n+1$ (sur l'ensemble d'horloges $X \cup X'$) définie par :

- pour $i, j = 0 \dots n$, $\eta_{i,j} = (m_{i,j}; \prec_{i,j})$,
- pour chaque mise à jour up_i ($1 \leq i \leq n$), nous distinguons les différents cas possibles pour up_i :

si up_i est :	alors nous posons :
$x_i := c$	$\eta_{i+n,0} = (c, \leq)$ et $\eta_{0,i+n} = (-c; \leq)$
$x_i := x_j + c$	$\eta_{i+n,j} = (c; \leq)$ et $\eta_{j,i+n} = (-c; \leq)$
$x_i :\prec c$ avec $\prec \in \{<; \leq\}$	$\eta_{i+n,0} = (c; \prec)$ et $\eta_{0,i+n} = (0; \leq)$
$x_i :\succ c$ avec $\succ \in \{>; \geq\}$	$\eta_{0,i+n} = (-c; \prec)$
	(où $\prec$ est $<$ si $\succ$ est $>$ et $\prec$ est $\leq$ si $\succ$ est $\geq$)
$x_i :\prec x_j + c$ avec $\prec \in \{<; \leq\}$	$\eta_{i+n,j} = (c; \prec)$ et $\eta_{j,i+n} = (\infty; <)$
$x_i :\succ x_j + c$ avec $\succ \in \{>; \geq\}$	$\eta_{j,i+n} = (-c; \prec)$
	(où $\prec$ est $<$ si $\succ$ est $>$ et $\prec$ est $\leq$ si $\succ$ est $\geq$)

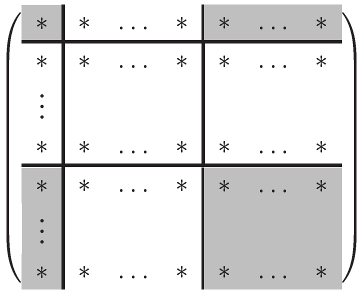
  Si up_i est une conjonction de mises à jour simples, nous prenons les plus petites valeurs calculées par l'un de ces up_i,
- pour chaque $i, j = 0 \dots 2n+1$, si le coefficient $\eta_{i,j}$ n'a pas encore été affecté, nous posons $\eta_{i,j} = (+\infty; <)$.

Par construction, la DBM N représente la contrainte d'horloges φ'_{up} (les indices $1 \leq i \leq n$ correspondent à l'ensemble des horloges X alors que les indices $n+1 \leq i \leq 2n+1$ correspondent à l'ensemble des horloges X') :

$$[\![N]\!] = \{V \text{ valuation sur } X \cup X' \mid V \models \varphi'_{up}\}.$$

Nous définissons la DBM $(\phi(N))_{|X'}$ comme étant la sous-matrice carrée de N de taille $n+1$ où nous effaçons tout ce qui concerne l'ensemble d'horloges X. Ceci peut être représenté par le dessin à droite (la partie blanche correspond aux horloges de X). Nous obtenons alors :

$$[\![(\phi(N))_{|X'}]\!] = up([\![M]\!]).$$



***Exemple* 8.5** Considérons la zone Z définie par la DBM dans laquelle nous ne conservons que les entiers, et pas les opérateurs de comparaison (pous simplifier)

$$\begin{pmatrix} 0 & -2 & -1 \\ 4 & 0 & 3 \\ 2 & -1 & 0 \end{pmatrix}$$

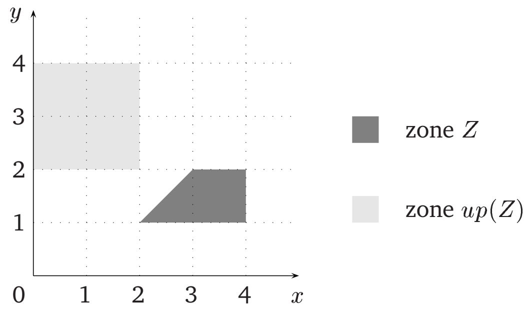
ainsi, que la mise à jour $up = (x :< 2 \wedge y := x)$.

La matrice N qui est calculée est :

$$\begin{pmatrix} 0 & -2 & -1 & \mathbf{2} & \infty \\ 4 & 0 & 3 & \infty & \mathbf{0} \\ 2 & -1 & 0 & \infty & \infty \\ \mathbf{0} & \infty & \infty & \infty & \infty \\ \infty & \mathbf{0} & \infty & \infty & \infty \end{pmatrix}$$

ce qui, après application de l'algorithme de Floyd et projection, donne la DBM (sous forme normale) suivante pour représenter $up(Z)$:

$$\begin{pmatrix} 0 & -2 & -2 \\ 2 & 0 & 0 \\ 2 & 2 & 0 \end{pmatrix}$$



β-sur-approximation. Soit $\beta = ((\max_x)_{x\in X}; (\max_{x,y})_{x,y\in X})$ un uplet définissant un ensemble de régions. Nous supposons que $X = \{x_1, \ldots, x_n\}$. Une DBM $M = (m_{i,j}; \prec_{i,j})_{i,j}$ est dite β-*bornée* si :

$$\begin{cases} 0 \leq m_{i,0} \leq \max_{x_i} \quad \text{ou} \quad m_{i,0} = \infty & \text{pour tout } 1 \leq i \leq n \\ -\max_{x_i} \leq m_{0,i} \leq 0 & \text{pour tout } 1 \leq i \leq n \\ (-\max_{x_j,x_i} \leq m_{i,j} \leq \max_{x_i,x_j} \quad \text{ou} \quad m_{i,j} = \infty) & \\ \qquad\qquad \text{si} \quad m_{i,0} = \infty \quad \text{ou} \quad m_{j,0} = \infty & \text{pour tout } 1 \leq i,j \leq n \\ -\max_{x_j} \leq m_{i,j} \leq \max_{x_i} \quad \text{si} \quad m_{i,0} < \infty \text{ et } m_{j,0} < \infty & \text{pour tout } 1 \leq i,j \leq n \end{cases}$$

Nous avons alors le résultat suivant :

Lemme 8.6 *Soit Z une zone. Alors*

$$Z \in \mathcal{Z}_\beta \iff \textit{il existe une DBM } \beta\textit{-bornée } M_\beta \textit{ telle que } Z = [\![M_\beta]\!].$$

PREUVE : Si M_β est une DBM β-bornée, alors $[\![M_\beta]\!]$ est dans $\mathcal{Z}_\beta$.
Une région $R \in \mathcal{R}_\beta$ est β-bornée. Supposons que Z soit une zone, union (convexe) de deux zones Z_1 et Z_2 qui appartiennent à $\mathcal{Z}_\beta$ et telles qu'il existe deux DBMs β-bornées M_{Z_1} et M_{Z_2} telles que $Z_1 = [\![M_{Z_1}]\!]$ et $Z_2 = [\![M_{Z_2}]\!]$. Nous supposons que pour $l = 1, 2$, $M_{Z_l} = (m^{(l)}_{i,j}; \prec^{(l)}_{i,j})_{i,j=1\dots n}$ et nous construisons $M = (m_{i,j}; \prec_{i,j})_{i,j=1\dots n}$ telle que :

$$(m_{i,j}; \prec_{i,j}) = \max((m^{(1)}_{i,j}; \prec^{(1)}_{i,j}), (m^{(2)}_{i,j}; \prec^{(2)}_{i,j})).$$

Nous commençons par remarquer que M est une DBM β-bornée. Nous voulons montrer que $[\![M]\!] = Z$. Nous notons que pour $l = 1, 2$, $M_{Z_l} \leq M$, donc $Z_l \subseteq [\![M]\!]$ et ceci implique que $Z \subseteq [\![M]\!]$.
Soit r un réel tel que $-m_{j,i} \prec_{j,i} r \prec_{i,j} m_{i,j}$. Il existe k tel que $(m_{i,j}; \prec_{i,j}) = (m^{(k)}_{i,j}; \prec^{(k)}_{i,j})$. Nous obtenons donc que $(m^{(k)}_{j,i}; \prec^{(k)}_{j,i}) \leq (m_{j,i}; \prec_{j,i})$, ainsi $-m^{(k)}_{j,i} \prec^{(k)}_{j,i} r \prec^{(k)}_{i,j} m^{(k)}_{i,j}$. Ceci implique qu'il existe une valuation $v \in Z_k$ telle que $v(i) - v(j) = r$. Comme conséquence, nous ne pouvons resserrer aucune constante de M. Donc, $Z = [\![M]\!]$. Ceci termine la preuve. □

Nous considérons l'algorithme suivant (où $M = (m_{i,j}; \prec_{i,j})$ et $M' = (m'_{i,j}; \prec'_{i,j})$) :

```
Algorithme de β-approximation (M:n-DBM) {
  M' = φ (M);                (* Forme normale de M *)
  Pour i de 1 à n faire
    Pour j de 1 à n faire
      Si ((m'i,0;≺i,0)>(maxi;<)) ou ((m'j,0;≺j,0)>(maxj;<)) alors {
        Si (m'i,j;≺i,j)>(maxi,j;≤) alors (m'i,j;≺i,j) devient (+∞;<);
        Si (m'i,j;≺i,j)<(mini,j;<) alors (m'i,j;≺i,j) devient (mini,j;<); };
    Findupour;
  Findupour;
  Pour i de 0 à n faire
    Si (m'i,0;≺i,0)>(maxi;≤) alors (m'i,0;≺i,0) devient (+∞;<);
    Si (m'0,i;≺0,i)<(-maxi;<) alors (m'0,i;≺0,i) devient (-maxi,<);!
  Findupour;
  Retourner M';}
```

Si M est une DBM, en lui appliquant cet algorithme, nous obtenons une DBM que nous notons $\phi_\beta(M)$. Nous pouvons remarquer que $\phi_\beta(M)$ est β-bornée et que $[\![\phi_\beta(M)]\!]$ est dans $\mathcal{Z}_\beta$.

Propriété 8.7 *Si M est une DBM, alors nous avons que $[\![\phi_\beta(M)]\!] = \mathsf{Approx}_\beta([\![M]\!])$.*

PREUVE : Soit M une DBM en forme normale. Nous obtenons immédiatement l'inclusion

$$\mathsf{Approx}_\beta([\![M]\!]) \subseteq [\![\phi_\beta(M)]\!]$$

car $\mathsf{Approx}_\beta([\![M]\!])$ est la plus petite zone de $\mathcal{Z}_\beta$ qui contient $[\![M]\!]$ et $[\![\phi_\beta(M)]\!]$ est une telle zone.
Il existe une DBM β-bornée M_β telle que $\mathsf{Approx}_\beta([\![M]\!]) = [\![M_\beta]\!]$. Nous obtenons alors que $M \leq M_\beta$. Supposons que $M = (m_{i,j}; \prec_{i,j})_{i,j=1\dots n}$ et $M_\beta = (m^{(\beta)}_{i,j}; \prec^{(\beta)}_{i,j})_{i,j=1\dots n}$. Nous montrons, coefficient par coefficient, que $\phi_\beta(M) \leq M_\beta$. Par exemple, supposons que

$$(m_{i,0}; \prec_{i,0}) > (\max_i; \leq) \text{ et } (m_{i,j}; \prec_{i,j}) > (\max_{i,j}; \leq).$$

Alors $M \leq M_\beta$ implique que $(m_{i,j}^{(\beta)}; \prec_{i,j}^{(\beta)}) = (\infty; <)$. Nous ne détaillons pas les autres cas (qui sont nombreux), mais, de la même manière, nous montrons que $\phi_\beta(M) \leq M_\beta$, ce qui implique que $[\![\phi_\beta(M)]\!] \subseteq \mathsf{Approx}_\beta([\![M]\!])$. Ceci termine la preuve. $\square$

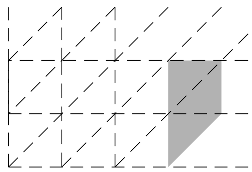
L'`Algorithme de` β`-approximation` appliqué à M calcule donc une DBM qui représente la β-approximation de $[\![M]\!]$. En outre, la complexité du calcul d'une telle β-sur-approximation n'est pas très élevée (sa complexité est en $\mathcal{O}(n^3)$ si n est le nombre d'horloges).

Utilisant les résultats de la partie 8.1.3, nous obtenons que chaque opération de l'algorithme 8.2 peut être calculée en utilisant les DBMs comme structure de données : les DBMs dont donc appropriées pour implémenter l'algorithme 8.2. De plus, la complexité de chaque étape de l'algorithme est, comme pour les automates temporisés classiques, polynômiale en le nombre d'horloges.

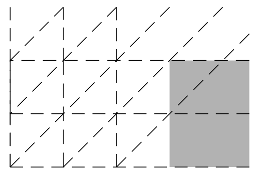
8.3 Preuve de la correction de l'algorithme

Comme nous l'avons annoncé dans les parties 8.1.2 et 8.2.3, la correction des algorithmes 8.1 et 8.2 est loin d'être évidente. En effet, ces algorithmes calculent une sur-approximation de l'ensemble des états accessibles, ce qui pourrait être strictement plus grand que l'ensemble des états réellement accessibles. Nous commençons par discuter la correction et montrer que, même pour l'algorithme 8.1, la preuve de correction n'est pas si évidente. Pourtant, bien que cet algorithme ait été plusieurs fois implémenté, nous n'avons trouvé aucune preuve convaincante dans la littérature. Bien sûr, il y a plusieurs articles qui prétendent de telles preuves, mais toutes ces preuves, si nous les avons bien comprises, sont incomplètes et même parfois incorrectes.

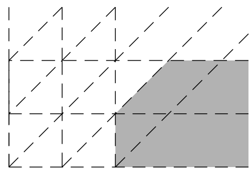
– Dans la thèse [Tri98], l'implémentation de la « *c-closure* » proposée à la page 127 correspond à notre approximation Approx_c. L'algorithme « Yes/No » devrait correspondre à l'algorithme 8.1. Cependant, la définition de la « c-closure » donnée à la page 21 ne correspond pas à l'implémentation. Par exemple, considérons le graphe des régions où c vaut 2 :



zone Z



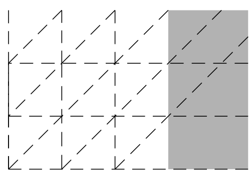
« *c-closure* » de Z



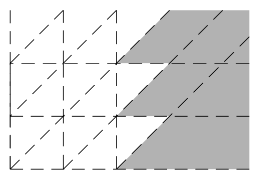
ce que l'implémentation calcule

La preuve de la correction de l'algorithme « Yes/No » (Lemme 5.9 page 58) est faite en utilisant la « c-closure » qui est définie et non celle qui est implémentée. La correction de l'algorithme implémenté n'est donc pas complète.

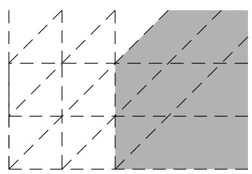
– Dans la thèse [WT94], si Z est une zone, alors *round*(Z) correspond à notre sur-approximation Approx_k. Le théorème 4.8 de la page 90 vise à montrer la correction de l'algorithme 8.1. La preuve est faite de la manière suivante : considérons une zone Z, elle est décomposée en une union finie de régions. Le calcul de l'opérateur *round* est d'abord fait sur les sous-zones (cette étape est assez facile) ; *round*(Z) est alors obtenu en calculant l'union du *round* de toutes ces sous-zones. Cependant, *round* n'est pas compatible avec l'union, par exemple, considérons la zone suivante :



zone Z



round appliqué aux sous-zones de Z



round(Z)

En plus de cela, cette preuve ne vise à montrer la correction de l'algorithme 8.1 que pour les automates temporisés avec contraintes non diagonales.

Nous revenons maintenant à la preuve de correction de l'algorithme 8.2. Lorsque cela sera fait, un simple corollaire sera la correction de l'algorithme 8.1 pour les automates temporisés classiques. Ce corollaire est très important car il est la base des algorithmes de UPPAAL et KRONOS.

La preuve se décompose en plusieurs étapes. L'algorithme 8.2 utilise une sur-approximation des zones, Approx_β. Il serait plutôt difficile de prouver la correction directement en utilisant cette approximation. Nous allons donc utiliser un algorithme intermédiaire qui utilise une autre sur-approximation. Malheureusement, cet algorithme ne peut pas être implémenté facilement, mais sa correction peut, elle, être prouvée assez aisément.

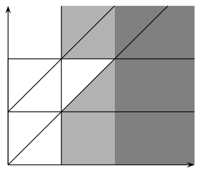
8.3.1 L'algorithme modifié

Pour chaque zone Z, pour chaque uplet α, nous définissons la *clôture par régions* de $\mathcal{R}_\alpha$ de Z par

$$\mathsf{Closure}_\alpha(Z) = \bigcup \{R \in \mathcal{R}_\alpha \mid R \cap Z \neq \varnothing\}.$$

Nous disons que $\mathsf{Closure}_\alpha(Z)$ est l'α-*clôture* de Z. Notons tout d'abord que $\mathsf{Closure}_\alpha(Z)$ peut être différent de $\mathsf{Approx}_\alpha(Z)$: $\mathsf{Approx}_\alpha(Z)$ est convexe alors que $\mathsf{Closure}_\alpha(Z)$ peut ne pas être convexe.

***Exemple* 8.8** Considérons l'ensemble de régions défini comme sur le dessin à côté. Si nous considérons la zone en gris foncé Z, alors la clôture par régions $\mathsf{Closure}_\alpha(Z)$ est dessinée en gris clair (et gris foncé) et n'est pas convexe, comme nous pouvons le constater sur le dessin.



L'algorithme modifié est présenté comme étant l'algorithme 8.3.

Algorithme 8.3 Algorithme des zones modifié pour les automates temporisés avec mises à jour

```
Algorithme des zones modifié (𝒜:UTA) {
  Calculer α en utilisant 𝒰 et 𝒞 ;
  Visités := ∅;                    (* Visités contient les états visités *)
  Attente := {(q0,Closureα(Z0))};
  Répéter
    Prendre (q,Z) dans Attente et le Retirer;
    Si q est final alors {Retourner «~Oui~»;}
                   sinon {S'il n'y a pas de (q,Z')∈ Visités t.q. Z ⊆ Z'
                          alors {Visités := Visités ∪ {(q, Z⃗)};
                                 Successeur := {(q',Closureα(Post(Z,e))) |
                                                   e transition de q à q'};
                                 Attente := Attente ∪ Successeur;}}
  Jusqu'à (Attente = ∅);
  Retourner «~Non~»; }
```

La différence entre cet algorithme et l'algorithme 8.2 tient dans l'approximation qui est utilisée. Bien entendu, l'algorithme que nous venons de décrire termine (si α est fixé, quand Z varie, il y a un nombre fini de $\mathsf{Closure}_\alpha(Z)$ différents). Nous commençons par montrer la correction de cet algorithme.

Soit $\mathcal{U}$ un ensemble de mises à jour et $\mathcal{C}$ un ensemble de contraintes d'horloges. Soit α un uplet définissant un ensemble de régions $\mathcal{R}_\alpha$ comme dans la partie 8.2.1. Ceci implique en particulier (voir la partie 3.3) que pour toute contrainte g de $\mathcal{C}$, pour chaque mise à jour up de $\mathcal{U}$, pour chaque région R de $\mathcal{R}_\alpha$, pour chaque valuation v de R,

$$\begin{cases} R \subseteq g \quad \text{ou} \quad R \subseteq \neg g \\ up(v) \cap R \neq \varnothing \implies \forall v' \in [v]_\alpha,\ up(v') \cap R \neq \varnothing \end{cases} \tag{8.1}$$

En outre, ceci assure que le problème du vide est décidable pour la classe $Aut(\mathcal{C},\mathcal{U})$ (c'est le sujet de toute la partie 3.3).

Proposition 8.9 *L'*`Algorithme des zones modifié` *teste la vacuité de tous les automates temporisés avec mises à jour appartenant à une classe décidable.*

La preuve de cette proposition est faite en utilisant plusieurs résultats intermédiaires. Nous commençons par le résultat suivant :

Lemme 8.10 *Les deux conditions suivantes sont vérifiées :*

$$\begin{cases} \mathsf{Closure}_\alpha(g) = g & \textit{si } g \in \mathcal{C} \\ up(\mathsf{Closure}_\alpha(Z)) \subseteq \mathsf{Closure}_\alpha(up(Z)) & \textit{si } Z \textit{ zone et } up \in \mathcal{U} \end{cases} \tag{8.2}$$

PREUVE : Soit g une contrainte dans $\mathcal{C}$. Alors $\mathsf{Closure}_\alpha(g) = \bigcup\{R \in \mathcal{R}_\alpha \mid R \cap g \neq \varnothing\} = g$ car pour chaque région $R \in \mathcal{R}_\alpha$, si $R \cap g \neq \varnothing$ alors $R \subseteq g$.

Nous montrons maintenant la deuxième condition, c'est-à-dire que pour toute zone Z, pour chaque mise à jour $up \in \mathcal{U}$,

$$up(\mathsf{Closure}_\alpha(Z)) \subseteq \mathsf{Closure}_\alpha(up(Z)) \tag{8.3}$$

Considérons une zone Z. Nous pouvons écrire Z comme une union finie $\bigcup_f Z_i$ où il existe des régions R_i telles que $Z_i = Z \cap R_i$ et Z_i est non vide. Si nous arrivons à montrer que la condition (8.3) est vraie pour chaque zone Z_i, alors elle sera vraie aussi pour Z :

$$\begin{array}{rcll} up(\mathsf{Closure}_\alpha(Z)) & = & up(\mathsf{Closure}_\alpha(\cup_f Z_i)) & \\ & = & up(\cup_f \mathsf{Closure}_\alpha(Z_i)) & \\ & = & \cup_f up(\mathsf{Closure}_\alpha(Z_i)) & \\ & \subseteq & \cup_f \mathsf{Closure}_\alpha(up(Z_i)) & \text{si la condition (8.3) est vraie pour chaque } Z_i \\ & \subseteq & \mathsf{Closure}_\alpha(\cup_f up(Z_i)) & \\ & \subseteq & \mathsf{Closure}_\alpha(up(\cup_f Z_i)) & \\ & \subseteq & \mathsf{Closure}_\alpha(up(Z)) & \end{array}$$

Ainsi, il nous suffit de montrer la condition (8.3) pour les zones incluses dans une région de $\mathcal{R}_\alpha$. Soit Z une zone incluse dans une région. Nous souhaitons montrer que

$$up(\mathsf{Closure}_\alpha(Z)) \subseteq \mathsf{Closure}_\alpha(up(Z))$$

Supposons qu'il existe une valuation $v' \in up(\mathsf{Closure}_\alpha(Z)) \setminus \mathsf{Closure}_\alpha(up(Z))$. Il existe alors une valuation $v \in \mathsf{Closure}_\alpha(Z)$ telle que $v' \in up(v)$. Ainsi, nous obtenons que $up(v) \cap [v']_\alpha \neq \varnothing$. Soit v_0 dans Z, alors $[v]_\alpha = [v_0]_\alpha$ et, par hypothèse, $up(v_0) \cap [v']_\alpha \neq \varnothing$ et donc $[v']_\alpha \subseteq \mathsf{Closure}_\alpha(up(Z))$. C'est une contradiction avec le fait que $v' \notin \mathsf{Closure}_\alpha(up(Z))$. □

Nous montrons ensuite le lemme suivant :

Lemme 8.11 *Pour chaque zone* Z*,* $\overrightarrow{\mathsf{Closure}_\alpha(Z)} \subseteq \mathsf{Closure}_\alpha(\overrightarrow{Z})$ *et pour chaque contrainte d'horloges* g *telle que* $\mathsf{Closure}_\alpha(g) = g$*,* $g \cap \mathsf{Closure}_\alpha(Z) \subseteq \mathsf{Closure}_\alpha(g \cap Z)$*.*

PREUVE : Soit $v_0 \in \overrightarrow{\mathsf{Closure}_\alpha(Z)}$. Il existe une valuation $v_1 \in \mathsf{Closure}_\alpha(Z)$ et un réel $t \geq 0$ tels que $v_0 = v_1 + t$. Par définition de $\mathsf{Closure}_\alpha$, il existe une valuation $v_2 \in Z$ telle que $[v_1]_\alpha = [v_2]_\alpha$. Ainsi, $[v_0]_\alpha$ est un successeur temporel de $[v_2]_\alpha$: il existe $v_3 \in [v_0]_\alpha$ et $t' \geq 0$ tels que $v_3 = v_2 + t'$. Ainsi, $v_3 \in \overrightarrow{Z}$ et il s'ensuit que $v_0 \in \mathsf{Closure}_\alpha(\overrightarrow{Z})$ car $[v_3]_\alpha = [v_0]_\alpha$. Ainsi, nous obtenons la propriété que nous souhaitions.

La deuxième propriété est vraie grâce à la relation $\mathsf{Closure}_\alpha(g) = g$: prenons $v_0 \in g \cap \mathsf{Closure}_\alpha(Z)$. Nous avons que $[v_0]_\alpha \subseteq g$ et qu'il existe $v_1 \in [v_0]_\alpha \cap Z$. Nous obtenons que $v_1 \in g \cap Z$ et donc que $v_0 \in \mathsf{Closure}_\alpha(g \cap Z)$. □

En utilisant ces deux lemmes, nous pouvons maintenant montrer la proposition. Sous les conditions (8.2), si $e = (q, g, a, up, q')$ est une transition, nous calculons :

$$\begin{array}{rcll}
\mathtt{Post}(\mathsf{Closure}_\alpha(Z), e) & = & up(g \cap \overrightarrow{\mathsf{Closure}_\alpha(Z)}) & \\
& \subseteq & up(g \cap \mathsf{Closure}_\alpha(\overrightarrow{Z})) & \text{par le lemme 8.11} \\
& \subseteq & up(\mathsf{Closure}_\alpha(g \cap \overrightarrow{Z})) & \text{par le lemme 8.11} \\
& \subseteq & \mathsf{Closure}_\alpha(up(g \cap \overrightarrow{Z})) & \text{par la condition (8.2)} \\
& \subseteq & \mathsf{Closure}_\alpha(\mathtt{Post}(Z, e)) &
\end{array}$$

En particulier, si $\mathtt{Post}(Z, e)$ est vide, $\mathtt{Post}(\mathsf{Closure}_\alpha(Z), e)$ est vide (car $\mathsf{Closure}_\alpha(Z)$ est vide si et seulement si Z est vide).

En fait, l'`Algorithme des zones modifié` calcule, pour une suite de transitions $(e_i)_{i=1\ldots p}$ consécutives, la zone

$$\mathsf{Closure}_\alpha(\mathtt{Post}(\mathsf{Closure}_\alpha(\mathtt{Post}(\mathsf{Closure}_\alpha(\mathtt{Post}(\ldots, e_{p-2})), e_{p-1})), e_p)).$$

Utilisant de manière itérative la relation ci-dessus et le fait que $\mathsf{Closure}_\alpha$ est involutive, nous obtenons que :

$$\mathsf{Closure}_\alpha(\mathtt{Post}(\mathsf{Closure}_\alpha(\mathtt{Post}(\ldots, e_{p-1})), e_p)) \subseteq \mathsf{Closure}_\alpha(\mathtt{Post}(\mathtt{Post}(\ldots, e_{p-1}), e_p))$$

Nous obtenons alors, si $R(\langle q_0, Z_0\rangle)$ et $MZA(\langle q_0, Z_0\rangle)$ représentent respectivement l'ensemble exact des états accessibles et l'ensemble des états calculés par l'`Algorithme des zones modifié`,

$$R(\langle q_0, Z_0\rangle) \quad \subseteq \quad MZA(\langle q_0, Z_0\rangle) \quad \subseteq \quad \mathsf{Closure}_\alpha(R(\langle q_0, Z_0\rangle))$$

Ainsi, $MZA(\langle q_0, Z_0\rangle) = \varnothing \iff R(\langle q_0, Z_0\rangle) = \varnothing$, et ceci conclut la preuve de la proposition. □

Pour chaque automate temporisé « décidable » $\mathcal{A}$ (c'est-à-dire appartenant à une sous-classe décidable décrite dans la partie 8.2.1), nous pouvons tester le vide de $L(\mathcal{A})$ en utilisant l'`Algorithme des zones modifié`. Cependant, cet algorithme ne peut pas être implémenté facilement (car les ensembles de valuations qui sont manipulés ne sont pas toujours convexes), mais sa correction nous aidera à montrer celle de l'algorithme 8.2. Pour terminer la preuve, nous allons utiliser quelques propriétés des DBMs.

8.3.2 Quelques propriétés des DBMs

Avant d'établir quelques propriétés des DBMs et des zones, nous nous attardons sur quelques définitions.

Règles syntaxiques de réécriture. Une *règle élémentaire syntaxique de réécriture* sur les DBMs du coefficient (i_0, j_0) est une fonction ψ qui associe à chaque DBM M une DBM M' de la manière suivante : supposons que $M = (m_{i,j}; \prec_{i,j})_{i,j}$. Alors $M' = (m'_{i,j}; \prec'_{i,j})_{i,j}$ est telle que pour tous i, j,

$$(m'_{i,j}; \prec'_{i,j}) = (m_{i,j}; \prec_{i,j}) \quad \text{si } (i,j) \neq (i_0, j_0).$$

Une *règle générale syntaxique de réécriture* sur les DBMs est une fonction ψ qui peut s'écrire comme une composition

$$\psi = \psi_{i_0,j_0} \circ \cdots \circ \psi_{i_l,j_l}$$

où ψ_{i_h,j_h} est une règle élémentaire syntaxique de réécriture du coefficient (i_h, j_h). Par exemple, ϕ_β, présenté dans la partie 8.2.4, est une règle générale syntaxique de réécriture.

Si Z est une zone et si ψ est une règle élémentaire ou générale syntaxique de réécriture sur les DBMs, alors nous notons $\psi(Z)$ la zone $[\![\psi(\phi(M_Z))]\!]$ où M_Z est une DBM qui représente la zone Z.

Projections. Soit $M = (m_{i,j}; \prec_{i,j})_{i,j}$ une DBM. Nous définissons sa *projection*, $\pi_{i_0,j_0}(M)$, sur le plan $(0, x_{i_0}, x_{j_0})$ (avec $i_0 < j_0$ et $i_0 \neq 0$) comme la restriction de M aux coefficients i_0 et j_0 :

$$\pi_{i_0,j_0}(M) = \begin{pmatrix} m_{0,0} & m_{i_0,0} & m_{j_0,0} \\ m_{0,i_0} & m_{i_0,i_0} & m_{j_0,i_0} \\ m_{0,j_0} & m_{i_0,j_0} & m_{j_0,j_0} \end{pmatrix}$$

Nous définissons aussi la projection sur le plan $(0, x_{i_0}, x_{j_0})$ des zones d'une manière géométrique (nous considérons alors la zone comme un polyèdre) et nous notons aussi $\pi_{i_0,j_0}(Z)$ cette projection de Z. Bien sûr, si M est une DBM, la relation suivante est vérifiée :

$$\pi_{i_0,j_0}([\![M]\!]) = [\![\pi_{i_0,j_0}(\phi(M))]\!].$$

Applications. Nous allons maintenant établir plusieurs propriétés des DBMs qui seront très utiles pour la fin de la preuve. Le lemme et le corollaire qui suivent ont pour but de rendre plus faciles certains points de la preuve par le fait qu'il va être suffisant de montrer les résultats pour la dimension 2.

Lemme 8.12 *Soit ψ une règle élémentaire syntaxique de réécriture sur les DBMs (avec n horloges) pour le coefficient (i, j). Soient Z et Z' des zones en dimension n . Supposons que :*

$$\pi_{i,j}(Z) \cap \psi(\pi_{i,j}(Z')) \neq \varnothing \implies \pi_{i,j}(Z) \cap \pi_{i,j}(Z') \neq \varnothing$$

Alors,

$$Z \cap \psi(Z') \neq \varnothing \implies Z \cap Z' \neq \varnothing.$$

PREUVE : Nous définissons la zone $Z'' = Z \cap \psi(Z')$. Alors, $Z'' \subseteq \psi(Z')$. Soit $M_{Z''}$ une DBM sous forme normale qui représente la zone Z''. Si $M_{Z'}$ est une DBM sous forme normale qui représente la zone Z', alors

$$M_{Z''} \leq \psi(M_{Z'}).$$

En projetant sur le plan $(0, x_i, x_j)$, nous obtenons que :

$$\pi_{i,j}(M_{Z''}) \leq \pi_{i,j}(\psi(M_{Z'})) = \psi(\pi_{i,j}(M_{Z'})).$$

Par hypothèse, ($\pi_{i,j}(M_{Z'})$ est sous forme normale car $M_{Z'}$ est sous forme normale),

$$\pi_{i,j}([\![M_{Z''}]\!]) \cap \pi_{i,j}([\![M_{Z'}]\!]) \neq \varnothing.$$

Nous en déduisons que :

$$\pi_{i,j}(Z \cap \psi(Z')) \cap \pi_{i,j}(Z') \neq \varnothing.$$

Soit $v \in Z \cap \psi(Z')$ une valuation telle que $v_{|\{i,j\}} \in \pi_{i,j}(Z \cap \psi(Z')) \cap \pi_{i,j}(Z')$. Comme ψ est une règle élémentaire syntaxique de réécriture pour le coefficient (i,j), nous obtenons que

$$v \in Z' \iff v_i \text{ et } v_j \text{ vérifient la contrainte indexée par } i \text{ et } j.$$

Comme $\pi_{i,j}(v)$ est dans $\pi_{i,j}(Z')$, nous obtenons que v est dans Z'. Le résultat s'en suit immédiatement. □

Corollaire 8.13 (Réduction à la dimension deux) *Soit ψ une règle générale syntaxique de réécriture sur les DBMs (avec n horloges). Supposons que pour tout $0 \leq i,j \leq n$, pour toutes les zones Z et Z',*

$$\pi_{i,j}(Z) \cap \psi(\pi_{i,j}(Z')) \neq \varnothing \implies \pi_{i,j}(Z) \cap \pi_{i,j}(Z') \neq \varnothing.$$

Alors, pour toutes les zones Z et Z',

$$Z \cap \psi(Z') \neq \varnothing \implies Z \cap Z' \neq \varnothing.$$

PREUVE : ψ peut être décomposé en $\psi = \psi_1 \circ \cdots \circ \psi_l$ où ψ_i est une règle élémentaire syntaxique de réécriture pour un coefficient. Appliquant l fois le lemme 8.12, nous obtenons que :

$$\begin{aligned} Z \cap \psi_1 \circ \cdots \circ \psi_l(Z') \neq \varnothing &\implies Z \cap \psi_2 \circ \cdots \circ \psi_l(Z') \neq \varnothing \\ &\implies Z \cap \psi_3 \circ \cdots \circ \psi_l(Z') \neq \varnothing \\ &\ \ \vdots \\ &\implies Z \cap \psi_l(Z') \neq \varnothing \\ &\implies Z \cap Z' \neq \varnothing. \end{aligned}$$

Ceci conclut bien la preuve du corollaire. □

Nous appliquerons cette réduction dans les preuves qui suivent.

8.3.3 Application aux automates temporisés avec contraintes non diagonales

Soit $\mathcal{A}$ un automate temporisé avec contraintes non diagonales, décidable. Comme décrit dans la partie 8.2.1, nous associons à $\mathcal{A}$ un ensemble de régions non diagonales $\mathcal{R}_\alpha$ et un ensemble de régions $\mathcal{R}_\beta$. Nous supposons que $\beta = ((\max_x)_{x \in X}, (\max_{x,y})_{x,y \in X})$.

Proposition 8.14 *Pour chaque zone Z, ce qui suit est vérifié :*

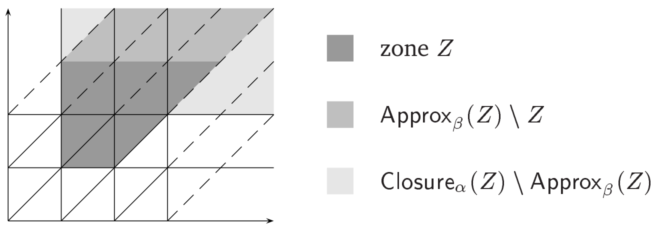
$$Z \subseteq \mathit{Approx}_\beta(Z) \subseteq \mathsf{Closure}_\alpha(Z).$$

PREUVE : Nous pouvons faire la preuve sur les DBMs et remarquer que si M est une DBM, alors $\mathsf{Approx}_\beta([\![M]\!]) = [\![\phi_\beta(M)]\!]$. De plus, ϕ_β est une règle générale syntaxique de réécriture, nous pouvons donc utiliser la réduction proposée dans le corollaire 8.13 et supposer que nous sommes dans un espace bidimensionnel. Il nous faudrait distinguer beaucoup de cas, mais il est suffisant de présenter en détails un seul des cas, car tous les autres sont semblables.

Nous supposons que

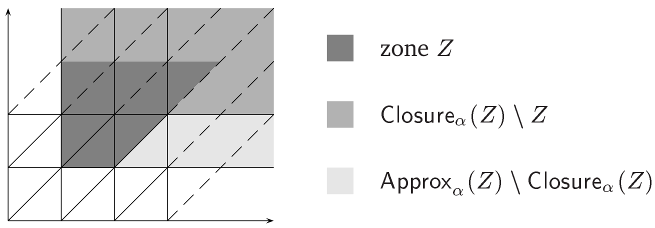
$$\begin{cases} m_{i,0} \leq \max_i \\ m_{j,0} > \max_j \\ m_{0,j} > -\max_j \end{cases}$$

et nous considérons la zone Z sur la figure de droite.



Il n'est pas difficile de montrer que $\mathsf{Approx}_\beta(Z) \subseteq \mathsf{Closure}_\alpha(Z)$. Ceci reste vrai pour n'importe quelle zone Z. $\square$

Remarque : Nous pouvons remarquer que si nous considérons Approx_α au lieu de Approx_β, la propriété précédente n'est plus vraie. Il suffit par exemple de regarder la figure à droite pour s'en convaincre.



L'algorithme 8.3 est correct vis-à-vis de l'accessibilité et la sur-approximation utilisée dans l'algorithme 8.2 est plus fine que celle utilisée dans l'algorithme 8.3, ce qui peut se décrire par le schéma suivant :

$$\begin{array}{ccccc} Z & \subseteq & \mathsf{Approx}_\beta(Z) & \subseteq & \mathsf{Closure}_\alpha(Z) \\ \uparrow & & \uparrow & & \uparrow \\ \text{Ensemble exact} & & \text{Algorithme 8.2} & & \text{Algorithme 8.3} \end{array}$$

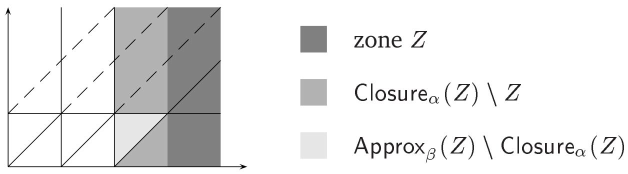
nous obtenons alors le théorème suivant :

Théorème 8.15 *L'algorithme 8.2 est correct, vis-à-vis de l'accessibilité, pour la classe des automates temporisés avec mises à jour et contraintes non diagonales.*

8.3.4 Application aux automates temporisés avec mises à jour générales

La propriété « *pour toute zone Z, $\mathsf{Approx}_\beta(Z) \subseteq \mathsf{Closure}_\alpha(Z)$* » n'est pas toujours vraie pour les ensembles de régions quelconques.

Par exemple, considérons l'ensemble de régions dessiné sur la figure de droite. La zone Z ne vérifie pas la propriété $\mathsf{Approx}_\beta(Z) \subseteq \mathsf{Closure}_\alpha(Z)$.



Au lieu de prouver cette propriété très forte, nous allons montrer que pour toute zone Z,

$$\mathsf{Approx}_\beta\left(up(\overrightarrow{\mathsf{Approx}_\beta(Z)} \cap g)\right) \subseteq \mathsf{Closure}_\alpha\left(up(\overrightarrow{\mathsf{Approx}_\beta(Z)} \cap g)\right) \tag{8.4}$$

ce qui sera la conséquence du fait que pour toute zone Z de $\mathcal{Z}_\beta$,

$$\mathsf{Approx}_\beta(up(Z)) \subseteq \mathsf{Closure}_\alpha(up(Z)) \tag{8.5}$$

car si une zone Z est dans $\mathcal{Z}_\beta$, alors $\overrightarrow{Z}$ est aussi dans $\mathcal{Z}_\beta$ et si deux zones Z_1 et Z_2 sont dans $\mathcal{Z}_\beta$, alors $Z_1 \cap Z_2$ est aussi dans $\mathcal{Z}_\beta$.
Si nous montrons la condition (8.5), alors la condition (8.4) sera aussi vérifiée, et, utilisant le fait que l'algorithme 8.3 est correct pour les automates temporisés décidables, nous obtiendrons

que l'ensemble des états calculés par l'algorithme 8.2 est inclus dans l'ensemble des états que l'algorithme 8.3 calcule, ce qui montre qu'aucun état de contrôle en trop n'est calculé.

Soit up une mise à jour comme décrite dans la partie 8.2.1 (cas général). Elle peut s'écrire comme une conjonction

$$up = \bigwedge_{x \in Y} (x := y_x) \wedge \bigwedge_{x \in X \setminus Y} (x :\sim c_x).$$

L'image d'une zone par up peut donc être calculée en commençant par appliquer la mise à jour

$$\bigwedge_{x \in Y} (x := y_x) \text{ (équivalente à la mise à jour } \bigwedge_{x \in Y} (x := y_x) \wedge \bigwedge_{x \in X \setminus Y} (x := x)\text{)}$$

et en appliquant ensuite successivement chaque mise à jour $x :\sim c_x$ (pour $x \in X \setminus Y$).

Les deux lemmes suivants ont pour but de montrer la condition (8.5).

Lemme 8.16 *Considérons la mise à jour* $up = \bigwedge_{i=1}^{n} (x_i := x_{h_i})$ $(X = \{x_i \mid i = 1 \dots n\})$ *et supposons que* $M = (m_{i,j}; \prec_{i,j})_{i,j=1\dots n}$ *est une DBM sous forme normale telle que* $[\![M]\!] = [\![\phi_\beta(M)]\!]$. *Si nous définissons la DBM* $M' = (m'_{i,j}; \prec'_{i,j})_{i,j=1\dots n}$ *par :*

$$(m'_{i,j}; \prec'_{i,j}) = (m_{h_i,h_j}; \prec_{h_i,h_j}) \text{ pour tout } i, j = 1 \dots n$$

alors M' est en forme normale, $[\![M']\!] = up([\![M]\!])$ *et* $[\![M']\!] = [\![\phi_\beta(M')]\!]$.

En particulier, si Z est une zone telle que $Z = \mathsf{Approx}_\beta(Z)$, alors

$$up(Z) = \mathsf{Approx}_\beta(up(Z)) \subseteq \mathsf{Closure}_\alpha(up(Z)).$$

Preuve : Supposons que r est un réel tel que $-m'_{h_j,h_i} \prec'_{h_j,h_i} r \prec'_{h_i,h_j} m'_{h_i,h_j}$. Alors nous avons que $-m_{j,i} \prec_{j,i} r \prec_{i,j} m_{i,j}$ et comme M est sous forme normale, il existe $v \in [\![M]\!]$ telle que $v_i - v_j = r$. Alors, v' définie par $up(v)$ vérifie $v'_i - v'_j = r$. Ceci implique que M' est sous forme normale (aucune contrainte ne peut être resserrée).

Le deuxième point du lemme est très facile : $[\![M']\!] = up([\![M]\!])$ par construction.

Nous allons maintenant montrer le dernier point du lemme, $[\![M']\!] = [\![\phi_\beta(M')]\!]$. Supposons donc que $[\![M']\!] \neq [\![\phi_\beta(M')]\!]$ et prenons $v' \in [\![\phi_\beta(M')]\!] \setminus [\![M']\!]$. Il y a une contrainte de M', disons celle représentée dans M' par $(m'_{i,j}; \prec_{i,j})$, qui n'est pas vérifiée par v'. Il existe donc des indices $i \neq j$ tels que $v'_{|\{i,j\}} \notin \pi_{i,j}([\![M']\!])$ alors que $v'_{|\{i,j\}} \in \pi_{i,j}([\![\phi_\beta(M')]\!])$. Comme

$$\begin{cases} \pi_{i,j}([\![M']\!]) = [\![\pi_{i,j}(M')]\!] & \text{car } M' \text{ est sous forme normale} \\ \pi_{i,j}([\![\phi_\beta(M')]\!]) \subseteq [\![\phi_\beta(\pi_{i,j}(M'))]\!] & \text{car } \phi_\beta \text{ est une règle de réécriture} \end{cases}$$

alors $[\![\pi_{i,j}(M')]\!] \neq [\![\phi_\beta(\pi_{i,j}(M'))]\!]$. Comme $h_i \neq h_j$ (sinon $x_i = x_j$ est la contrainte vérifiée dans M'), il y a une isométrie entre $[\![\pi_{h_i,h_j}(M)]\!]$ et $[\![\pi_{i,j}(M')]\!]$. Comme β est « symétrique » en i/h_i (par exemple, $\max_{i,j} = \max_i = \max_{h_i} = \max_{h_i,h_j}$) et en j/h_j, nous obtenons que $[\![\pi_{h_i,h_j}(M)]\!] \neq [\![\phi_\beta(\pi_{h_i,h_j}(M))]\!]$ et donc $\pi_{h_i,h_j}([\![M]\!]) \neq \pi_{h_i,h_j}([\![\phi_\beta(M)]\!])$. Ainsi, nous obtenons que $[\![M]\!] \neq [\![\phi_\beta(M)]\!]$, ce qui contredit nos hypothèses. Ceci conclut la preuve, $[\![M']\!] = [\![\phi_\beta(M')]\!]$. □

Lemme 8.17 *Considérons la mise à jour* $up = (x_k :\sim c)$ *où* $\sim \in \{<; \leq; =\}$ *et considérons aussi une DBM sous forme normale* $M = (m_{i,j}; \prec_{i,j})_{i,j=1\dots n}$ *telle que* $[\![M]\!] = [\![\phi_\beta(M)]\!]$. *Supposons que*

$M' = (m'_{i,j}; \prec'_{i,j})_{i,j}$ avec :

$$\left\{\begin{array}{l}(m'_{i,j}; \prec'_{i,j}) = (m_{i,j}; \prec_{i,j}) \ \textit{ si } i,j \neq k \\ (m'_{k,0}; \prec'_{k,0}) = \left\{\begin{array}{l}(c; \leq) \ \textit{ si } \ \sim \in \{\leq; =\} \\ (c; <) \ \textit{ si } \ \sim \in \{<\}\end{array}\right. \\ (m'_{0,k}; \prec'_{0,k}) = \left\{\begin{array}{l}(-c; \leq) \ \textit{ si } \ \sim \in \{=\} \\ (0; \leq) \ \textit{ si } \ \sim \in \{<; \leq\}\end{array}\right. \\ (m'_{k,i}; \prec'_{k,i}) = (m'_{k,0}; \prec'_{k,0}) + (m_{0,i}; \prec_{0,i}) \ \textit{ si } i \neq k \\ (m'_{i,k}; \prec'_{i,k}) = (m_{i,0}; \prec_{i,0}) + (m'_{0,k}; \prec'_{0,k}) \ \textit{ si } i \neq k\end{array}\right.$$

Alors M' est en forme normale, $[\![M']\!] = up([\![M]\!])$ et $[\![\phi_\beta(M')]\!] \subseteq \mathsf{Closure}_\alpha([\![M']\!])$.

Ainsi, si Z est une zone telle que $Z = \mathsf{Approx}_\beta(Z)$, alors $\phi_\beta(up(Z)) \subseteq \mathsf{Closure}_\alpha(up(Z))$.

PREUVE : Soit r un réel tel que $-m'_{j,i} \prec'_{j,i} r \prec'_{i,j} m'_{i,j}$. Nous distinguons plusieurs cas :

– Si i et j sont tous les deux différents de k, alors il existe une valuation $v \in [\![M]\!]$ (M est sous forme normale) telle que $v_i - v_j = r$. Toute valuation $v' \in up(v)$ vérifie $v'_i - v'_j = r$.

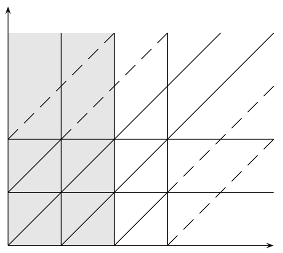
– Si $i = k$ et $j = 0$, alors le réel r vérifie $r \sim c$. Donc, si nous prenons la valuation $v \in [\![M]\!]$, $v' = [v_k \leftarrow r]v$ est dans $[\![M']\!]$.
Le cas où $i = 0$ et $j = k$ est identique.

– Si $i = k$ et $j \neq 0, k$, nous pouvons trouver $r' \sim c$ tel que $-m_{0,j} \prec_{0,j} r' - r \prec_{j,0} m_{j,0}$ (par construction de M'). Il existe une valuation $v \in [\![M]\!]$ telle que $v_j = r' - r$. Alors la valuation v' définie par $v'_k = r'$, $v'_j = r' - r$ et pour tout $i \neq j, k$, $v'_i = v_i$. La valuation v' est dans $up(v)$.
Le cas $i \neq 0, k$ et $j = k$ est identique.

Nous déduisons de ceci que M' est sous forme normale et il est facile de voir que $[\![M']\!] = up([\![M]\!])$. Nous pouvons montrer que $[\![\phi_\beta(M')]\!] \subseteq \mathsf{Closure}_\alpha(M')$. Nous supposons qu'il existe une région R de $\mathcal{R}_\beta$ (c'est donc une sous-région de $\mathcal{R}_\alpha$) telle que $R \cap \mathsf{Closure}_\alpha(M') = \varnothing$ et $R \subseteq [\![\phi_\beta(M')]\!]$. Il existe une contrainte de R (indexée par i, j) qui est plus grande que la contrainte de $\mathsf{Closure}_\alpha(M')$. Nous obtenons donc

$$\begin{array}{ccc}\pi_{i,j}(R) \cap \pi_{i,j}(\mathsf{Closure}_\alpha(M')) & = & \varnothing \\ \| & & \\ \pi_{i,j}(R) \cap \mathsf{Closure}_\alpha(\pi_{i,j}(M')) & & \end{array}$$

En outre, comme $R \subseteq [\![\phi_\beta(M')]\!]$, nous en déduisons que $\pi_{i,j}(R) \subseteq [\![\phi_\beta(\pi_{i,j}(M'))]\!]$.



Nous obtenons alors une contradiction : la zone $[\![\pi_{i,j}(M')]\!]$ est incluse dans la zone grisée de la figure de gauche et la condition $(\diamondsuit_{gen})$ présentée dans la partie 8.2.1 à la page 163 implique qu'il n'est pas possible d'avoir en même temps l'inclusion $\pi_{i,j}(R) \subseteq [\![\phi_\beta(\pi_{i,j}(M'))]\!]$ et $\pi_{i,j}(R) \cap \mathsf{Closure}_\alpha(\pi_{i,j}(M')) = \varnothing$ si R est une région de $\mathcal{R}_\beta$ □

Appliquant tout d'abord le lemme 8.16 puis plusieurs fois le lemme 8.17, nous obtenons la condition (8.5), c'est-à-dire que pour toute zone Z de $\mathcal{Z}_\alpha$, $\mathsf{Approx}_\beta(up(Z)) \subseteq \mathsf{Closure}_\alpha(up(Z))$. Comme nous le disions au début de cette partie, ceci implique que l'algorithme 8.2 est correct pour les automates temporisés décidables :

Théorème 8.18 *L'algorithme 8.2 est correct (vis-à-vis de l'accessibilité) pour les automates temporisés décidables (avec des contraintes générales).*

Comme un important corollaire de ce théorème, nous obtenons la correction de l'algorithme 8.1 pour les automates temporisés classiques (voir la partie 8.2.3 pour une explication).

Corollaire 8.19 (Application aux automates temporisés (classiques)) *L'algorithme 8.1 est correct (vis-à-vis de l'accessibilité) pour les automates temporisés classiques.*

Ce résultat montre que l'algorithme utilisé dans des outils tels que UPPAAL et KRONOS est correct, ce dont personne ne doutait bien que, à notre connaissance, aucune preuve correcte ne soit publiée.

8.4 Conclusion

Dans ce chapitre, nous avons donc proposé un algorithme d'analyse en avant qui permet de décider de la vacuité des automates temporisés avec mises à jour décidables. De plus, nous avons donné une description précise d'une implémentation possible de notre algorithme en utilisant les DBMs, la même structure de données que celle utilisée dans les outils UPPAAL ou KRONOS. Le travail qu'il reste à faire pour implémenter n'est donc plus très important car nous avons décrit avec précision les différentes étapes de calculs de notre algorithme. Il faut bien remarquer que notre algorithme apparaît réellement comme une extension de l'algorithme des zones implémenté dans UPPAAL et KRONOS pour les automates temporisés classiques et sa complexité n'est pas plus importante.

Ce travail est donc le dernier maillon de notre étude sur les automates temporisés avec mises à jour. Il fait le lien entre le travail théorique qui a été fait sur le sujet (voir la partie 3), décidabilité et expressivité, et un outil qui permettrait de traiter ces mises à jour.

Dans le futur, ceci facilitera la modélisation des systèmes temporisés en autorisant l'utilisation de mises à jour. Comme nous venons de le voir, la vérification de ces modèles ne nécessite pas une première étape consistant à transformer les automates temporisés avec mises à jour en automates temporisés classiques, ce qui est très positif car la transformation que nous avons présentée au chapitre 3 souffre d'une explosion combinatoire.

Une conséquence (assez importante) de notre résultat est la correction de l'algorithme des zones pour automates temporisés classiques, implémenté dans UPPAAL et KRONOS, mais dont nous n'avons trouvé aucune preuve complète dans la littérature.

Chapitre 9

Les automates de test

Ce chapitre provient des articles [ABBL98] et [ABBL02].

Comme nous l'avons dit dans le chapitre précédent, les propriétés d'accessibilité sont les plus simples à vérifier et il y a des algorithmes assez efficaces pour les vérifier. Par exemple, dans l'outil UPPAAL présenté brièvement à la page 155, l'algorithme implémenté est essentiellement un algorithme d'accessibilité. En pratique, il peut arriver que nous ayons à vérifier des propriétés autres que de l'accessibilité. Il est alors relativement naturel d'essayer de ramener la vérification de ces propriétés un peu plus « élaborées » à de la vérification de propriétés d'accessibilité *via* une réduction adéquate. La réduction que nous nous proposons d'étudier est la suivante. Étant donnée une propriété ϕ, nous construisons un « observateur », appelé *automate de test* et noté T_ϕ. Cet automate vérifie alors que si $\mathcal{A}$ est un automate temporisé,

$$\mathcal{A} \text{ vérifie } \phi \iff \mathcal{A} \parallel T_\phi \text{ ne permet pas d'atteindre un état rejetant de } T_\phi$$

Ainsi, pour tester si $\mathcal{A}$ vérifie ϕ, il suffit de pouvoir tester l'accessibilité de certains états du système formé de la mise en parallèle de l'automate $\mathcal{A}$ avec l'observateur T_ϕ. Cette réduction a été utilisée pour traiter des études de cas [KP95, BGK+96, JLS96].

Il est alors naturel de se poser la question de savoir quelles sont les propriétés pouvant être vérifiées de cette manière. Dans ce chapitre, nous définissons un fragment d'un langage proche de L_ν (voir page 147) et nous proposons un algorithme qui permet de construire, de manière syntaxique, pour chaque formule ϕ de notre langage, un automate de test T_ϕ comme décrit ci-dessus. Nous montrons alors que ce fragment de langage logique caractérise complètement l'ensemble des propriétés dont la vérification peut se réduire à un test d'accessibilité dans le sens défini ci-dessus.

Nous considérons tout d'abord le langage de propriétés SBLL qui permet d'exprimer des propriétés de sûreté et de vivacité bornée pour les systèmes temporisés. Nous montrons que SBLL est *testable*, dans le sens où des automates de test peuvent être construits pour chaque propriété de SBLL. Cependant, SBLL n'est pas un langage *expressivement complet* vis-à-vis des automates de test, dans le sens où il ne permet pas d'exprimer toutes les propriétés qui peuvent être vérifiées *via* les automates de test. Nous présentons alors une extension de SBLL, appelée $\mathcal{L}_{\forall S}$, qui caractérise la limite précise de l'approche du model-checking par les automates de test. Plus précisément, nous montrons que :

- toute propriété ψ de $\mathcal{L}_{\forall S}$ est testable, dans le sens où il existe un automate de test T_ψ tel que tout automate temporisé S satisfait ψ si et seulement si $S \parallel T_\psi$ ne permet pas d'atteindre un état rejetant de T_ψ,

– tout automate de test T peut s'exprimer dans $\mathcal{L}_{\forall S}$, dans le sens où il existe une formule ψ_T de $\mathcal{L}_{\forall S}$ telle que, pour chaque automate temporisé S, le système $S \parallel T$ ne permet pas d'atteindre un état rejetant de T si et seulement si S satisfait ψ_T.

Ce chapitre est organisé de la façon suivante : après avoir présenté les spécificités du modèle à la UPPAAL que nous allons considérer (partie 9.1), légèrement différent de celui d'Alur et Dill, nous définissons la notion d'*automates de test* et nous décrivons comment ils peuvent être utilisés pour vérifier des propriétés (partie 9.2). Nous présentons ensuite les langages de propriétés SBLL et $\mathcal{L}_{\forall S}$ et nous prouvons leur testabilité (partie 9.3). Dans la partie 9.4, nous montrons que le langage de propriétés $\mathcal{L}_{\forall S}$ est complet vis-à-vis des automates de test.

9.1 Le modèle à la UPPAAL

Nous commençons par quelques définitions provenant des spécificités du modèle utilisé dans UPPAAL [LPY97].

9.1.1 Systèmes de transitions temporisés avec urgence

Pour reprendre les notations classiques utilisées dans les articles traitant d'UPPAAL, nous noterons Act l'ensemble des actions. Celui-ci sera décomposé en deux ensembles disjoints, un ensemble classique d'actions Σ et un ensemble d'actions *urgentes* Σ_u. Nous supposerons que Act est doté d'une fonction $\bar{\cdot}$: Act $\rightarrow$ Act telle que pour tout $a \in$ Act, $\overline{\overline{a}} = a$. En outre, nous imposons que $\overline{a} \in \Sigma \iff a \in \Sigma$ pour toute action $a \in$ Act (remarquons que comme Σ et Σ_u sont disjoints, cela est vrai aussi pour Σ_u). Nous noterons τ l'action silencieuse (ou *action interne*) au lieu de ε et nous poserons $\mathsf{Act}_\tau = \mathsf{Act} \cup \{\tau\}$.

Soit $\mathcal{T} = (S, \Gamma, s_0, \longrightarrow)$ un système de transitions temporisé où $\Gamma = \mathsf{Act}_\tau \cup \{\epsilon(d) \mid d \in \mathbb{T}\}$ comme présenté à la définition 3.34, page 69. Ce système de transitions est dit *avec urgence* si les deux hypothèses suivantes sont vérifiées :

– PERSISTANCE EN AVANT DES ACTIONS URGENTES : pour tous $s, s', s'' \in S$, $a \in \Sigma_u$ et $d \in \mathbb{R}^+$, si $s \xrightarrow{\epsilon(d)} s'$ et $s \xrightarrow{a} s''$, alors il existe $\overline{s} \in S$ tel que $s' \xrightarrow{a} \overline{s}$
– PERSISTANCE EN ARRIÈRE DES ACTIONS URGENTES : pour tous $s, s', s'' \in S$, $a \in \Sigma_u$ et $d \in \mathbb{R}^+$, si $s \xrightarrow{\epsilon(d)} s'$ et $s' \xrightarrow{a} s''$, alors il existe $\overline{s} \in S$ tel que $s \xrightarrow{a} \overline{s}$

Dans toute la suite de ce chapitre, nous considérerons uniquement ces systèmes de transitions temporisés « avec urgence ».

Nous noterons $s \xrightarrow{\alpha}$ pour représenter le fait qu'il existe un état s' tel que $s \xrightarrow{\alpha} s'$, et $s \not\xrightarrow{\alpha}$ s'il n'existe pas d'état s' tel que $s \xrightarrow{\alpha} s'$.

Soit $\mathcal{T} = (S, \Gamma, s_0, \longrightarrow)$ un système de transitions temporisé avec urgence. Nous définissons de nouvelles relations de transitions, paramétrées par un ensemble d'actions urgentes S :

$$
\begin{array}{lll}
s \xrightarrow{\mu}_S s' & \text{ssi} & s \xrightarrow{\mu} s' \text{ pour tout } \mu \in \mathsf{Act}_\tau \\
s \xrightarrow{\epsilon(d)}_S s' & \text{ssi} & s \xrightarrow{\epsilon(d)} s' \text{ et } \forall d' \in [0, d[,\ (s \xrightarrow{\epsilon(d')} s_{d'} \implies \forall a \in S,\ s_{d'} \not\xrightarrow{a}) \\
s \xRightarrow{a}_S s' & \text{ssi} & s \xRightarrow{a} s'^{1} \\
s \xRightarrow{\epsilon(d)}_S s' & \text{ssi} & \text{il existe une exécution d'attente}^2 \\
 & & s = s_0 \xrightarrow{\alpha_1}_S s_1 \xrightarrow{\alpha_2}_S \ldots \xrightarrow{\alpha_n}_S s_n = s' \quad (n \geq 0) \text{ où} \\
 & & d = \sum \{d_i \mid \alpha_i = \epsilon(d_i)\}
\end{array}
$$

Intuitivement, $s \xrightarrow{\epsilon(d)}_S s'$ est vrai si s peut attendre d unités de temps sans qu'aucune action de S ne devienne possible. Remarquons que comme S ne contient que des actions urgentes, les hypothèses de persistance des actions urgentes entraînent que soit $d = 0$, soit $s \not\xrightarrow{a}$, pour tout $a \in S$. De manière similaire, $s \overset{\epsilon(d)}{\Longrightarrow}_S s'$ est vrai s'il existe une exécution d'attente de durée d à partir de l'état s pour laquelle les transitions d'attente de durée positive ont seulement lieu dans des états où aucune action urgente de S n'est permise.

***Exemple* 9.1** Considérons un système de transitions temporisé dont les états sont s et s'. Les transitions d'actions sont $s \xrightarrow{a} s$, avec a une action urgente, et $s \xrightarrow{\tau} s'$. Les transitions d'attente sont $s \xrightarrow{\epsilon(d)} s$ et $s' \xrightarrow{\epsilon(d)} s'$ pour chaque $d \in \mathbb{R}^+$. Alors, pour chaque $d \in \mathbb{R}^+$, l'état s permet l'exécution d'attente suivante : $s \xrightarrow{\tau} s' \xrightarrow{\epsilon(d)}_{\{a\}} s'$. Ainsi, $s \overset{\epsilon(d)}{\Longrightarrow}_{\{a\}} s'$ pour chaque $d \in \mathbb{R}^+$.

Remarquons que, comme dans l'exemple ci-dessus, une exécution d'attente permettant de faire $s \overset{\epsilon(d)}{\Longrightarrow}_S s'$ peut traverser des états à partir desquelles des actions de S pourraient être faites, mais au niveau de ces états, aucune attente n'est autorisée.

Nous définissons définir un opérateur qui permet de restreindre l'ensemble des actions qui peuvent être effectuées.

Définition 9.2 *Soit* $\mathcal{T} = (S, \Gamma, s_0, \longrightarrow)$ *un système de transitions temporisé et soit* $L \subseteq \mathsf{Act}$ *un ensemble d'actions. La* restriction *de* $\mathcal{T}$ *par rapport à* L *est le système de transitions temporisé*

$$\mathcal{T}\backslash L = (S\backslash L, \Gamma, s_0\backslash L, \rightsquigarrow)$$

où $S\backslash L = \{s\backslash L \mid s \in S\}$ *et la relation de transitions* $\rightsquigarrow$ *est définie par les trois règles suivantes :*

(1)	$\dfrac{s \xrightarrow{\tau} s'}{s\backslash L \overset{\tau}{\rightsquigarrow} s'\backslash L}$	(2)	$\dfrac{s \xrightarrow{\epsilon(d)} s'}{s\backslash L \overset{\epsilon(d)}{\rightsquigarrow} s'\backslash L}$
(3)	$\dfrac{s \xrightarrow{a} s'}{s\backslash L \overset{a}{\rightsquigarrow} s'\backslash L}$		$a, \overline{a} \notin L$

où $s,\ s'$ sont des états de $\mathcal{T}$,
$L \subseteq \mathsf{Act}$, $a \in \mathsf{Act}$, et $d \in \mathbb{R}^+$.

Tableau 9.1: Règles définissant la relation de transition $\rightsquigarrow$ dans $\mathcal{T}\backslash L$

Définition 9.3 *Soient* $\mathcal{T}_i = (S_i, \Gamma, s_i^0, \longrightarrow_i)$ $(i \in \{1,2\})$ *deux systèmes de transitions temporisés. La* composition parallèle *de* $\mathcal{T}_1$ *et* $\mathcal{T}_2$ *est le système de transitions temporisé*

$$\mathcal{T}_1 \parallel \mathcal{T}_2 = (S_1 \times S_2, \Gamma, (s_1^0, s_2^0), \longrightarrow)$$

où la relation de transitions $\longrightarrow$ *est définie par les règles présentées dans le tableau 9.2.*

Un couple (s_1, s_2) sera noté par la suite $s_1 \parallel s_2$.

[2] La relation $\overset{a}{\Longrightarrow}_S$ a été définie à la page 69 et est définie par $s \overset{a}{\Longrightarrow} s' \iff \exists s'',\ s \xrightarrow{\varepsilon}{}^* s'' \xrightarrow{a} s'$.

[2] Rappelons (voir page 69) qu'une exécution d'attente est une suite de transitions $s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \ldots \xrightarrow{\alpha_n} s_n$ telle que $n \geq 0$, pour tout $1 \leq i \leq n$, $\alpha_i = \varepsilon$ ou bien $\alpha_i = \epsilon(d_i)$ pour un certain $d_i \in \mathbb{T}$.

(1) $\dfrac{s_1 \xrightarrow{\mu}_1 s_1'}{s_1 \| s_2 \xrightarrow{\mu} s_1' \| s_2}$	(2) $\dfrac{s_2 \xrightarrow{\mu}_2 s_2'}{s_1 \| s_2 \xrightarrow{\mu} s_1 \| s_2'}$
(3) $\dfrac{s_1 \xrightarrow{a}_1 s_1' \qquad s_2 \xrightarrow{\overline{a}}_2 s_2'}{s_1 \| s_2 \xrightarrow{\tau} s_1' \| s_2'}$	
(4) $\dfrac{s_1 \xrightarrow{\epsilon(d)}_1 s_1' \qquad s_2 \xrightarrow{\epsilon(d)}_2 s_2'}{s_1 \| s_2 \xrightarrow{\epsilon(d)} s_1' \| s_2'}$	$d = 0$ ou $\forall a \in \Sigma_u.$ $\neg(s_1 \xrightarrow{a}_1 \ \wedge s_2 \xrightarrow{\overline{a}}_2)$

où s_i, s_i' sont des état de $\mathcal{T}_i$ $(i \in \{1, 2\})$,
$\mu \in \mathsf{Act}_\tau$, $a, \overline{a} \in \mathsf{Act}$ et $d \in \mathbb{R}^+$.

Tableau 9.2: Règles définissant la relation de transitions $\longrightarrow$ dans $\mathcal{T}_1 \parallel \mathcal{T}_2$

La définition de l'opérateur $\|$ force les systèmes de transitions que nous composons à se synchroniser sur les attentes, avec la particularité qu'il n'est possible d'attendre que si aucune synchronisation entre actions urgentes n'est possible (voir la règle (4) du tableau 9.2). La composition parallèle que nous venons de définir correspond exactement à l'opérateur de composition parallèle utilisé dans UPPAAL et est très fortement similaire à la composition parallèle de TCCS [Yi90, Yi91] — la seule différence résidant dans le fait que dans TCCS, **toutes** les actions sont urgentes.

Le résultat suivant est classique.

Proposition 9.4 *La classe des systèmes de transitions temporisés avec urgence est close par les opérations de composition parallèle et de restriction.*

9.1.2 Automates temporisés *à la* UPPAAL

La notion d'automates temporisés que nous utilisons est une variamte du modèle original proposé par Alur et Dill et présenté au chapitre 2. Les automates que nous considérons n'ont ni états finals, ni états répétés car, dans ce chapitre, nous serons plus intéressés par les comportements que par les langages reconnus.

Définition 9.5 *Un* automate temporisé *à la* UPPAAL *est un 5-uplet $\mathcal{A} = (Q, X, \mathsf{Act}_\tau, q_0, T)$ où Q est un ensemble fini d'états, q_0 est l'état initial, X est un ensemble fini d'horloges et $T \subseteq Q \times \mathcal{C}(X) \times \mathsf{Act}_\tau \times 2^X \times Q$ est un ensemble de transitions. En outre, nous imposons la contrainte suivante :*

– URGENCE : *si $(q, g, \mu, r, q') \in T$ et $\mu \in \Sigma_u$, alors g est une tautologie, i.e. g représente la contrainte* Vrai.

***Exemple* 9.6** L'automate temporisé *à la* UPPAAL dessiné sur la figure 9.3 a cinq états étiquetés de q_0 à q_4, il a une horloge x, des actions $a \in \Sigma_u$ et $b \in \Sigma$, et quatre transitions. La transition de l'état q_1 à l'état q_2, par exemple, est contrainte par $x \geq 0$, étiquetée par l'action urgente a et remet à zéro l'horloge x. Remarquons que les contraintes des transitions étiquetées par une action urgente sont bien des tautologies.

Comme c'est le cas pour les automates temporisés classiques (voir la partie 3.4.1 à la page 70), la sémantique d'un automate temporisé *à la* UPPAAL $\mathcal{A}$ est donnée par un système de transitions temporisé $\mathcal{T}_\mathcal{A} = (S, \Gamma, s_0, \longrightarrow)$, où $S = \{(q, v) \mid q \in Q \text{ et } v \in \mathbb{T}^X\}$ est l'ensemble des états de $\mathcal{A}$, $s_0 = (q_0, \mathbf{0})$ est l'état initial ($\mathbf{0}$ est la valuation qui associe à chaque horloge zéro) et $\longrightarrow$ est la relation de transitions définie comme suit :

$$(q,v) \xrightarrow{\mu} (q',v') \quad \text{ssi} \quad \exists e = (q,g,\mu,r,q') \in T.\ g(v) \wedge v' = [r \leftarrow 0]v$$
$$(q,v) \xrightarrow{\epsilon(d)} (q',v') \quad \text{ssi} \quad q = q' \text{ et } v' = v + d$$

où $\mu \in \mathsf{Act}_\tau$ et $d \in \mathbb{T}$.

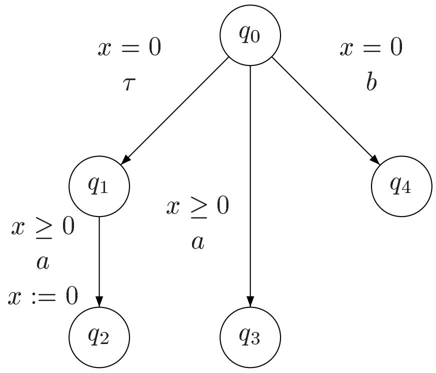


Figure 9.3: Automate temporisé $\mathcal{A}$ ($a \in \Sigma_u$ et $b \in \Sigma$)

***Exemple* 9.7** Ce qui suit est une suite valide de transitions pour l'automate temporisé de la figure 9.3, où le nombre entre accolades correspond à la valeur de l'horloge x :

$$(q_0, \{0\}) \xrightarrow{\tau} (q_1, \{0\}) \xrightarrow{\epsilon(3.14)} (q_1, \{3.14\}) \xrightarrow{a} (q_2, \{0\}).$$

Proposition 9.8 *Soit $\mathcal{A}$ un automate temporisé à la* UPPAAL. *Alors $\mathcal{T}_\mathcal{A}$ est un système de transitions temporisé avec urgence.*

Cette proposition est très facile à montrer, nous ne détaillons pas la preuve.

9.2 Les automates de test

Comme nous l'avons déjà dit, nous souhaitons donner une caractérisation complète de la classe de propriétés dont la vérification peut se réduire à de l'analyse d'accessibilité. Dans cette partie, nous commençons donc par définir de quelle manière la vérification de ces propriétés peut être transformée en de l'analyse d'accessibilité. De manière informelle, nous mettons en parallèle un automate (ou un réseau d'automates) avec un *automate de test*. Nous testons alors l'accessibilité d'un ou plusieurs états de cette mise en parallèle et nous disons que l'automate initial (ou le réseau d'automates initial) *rate le test* de l'automate de test si un état cadre *rejetant* peut être atteint dans la mise en parallèle. Dans la cas contraire, nous dirons que l'automate *passe le test*.

9.2.1 Que signifie « Tester » ?

Nous commençons par donner la définition d'un *automate de test* puis décrire comment un tel automate interagit avec les automates ou les réseaux d'automates que nous souhaitons vérifier. Nous pourrons alors définir ce que signifie « passer le test » ou bien « rater le test ».

Définition 9.9 *Un* automate de test *est un 7-uplet* $T = (N, X, X_0, \mathsf{Act}_\tau, n_0, N_T, E)$[3] *où le 5-uplet* $(N, X, \mathsf{Act}_\tau, n_0, E)$ *représente un automate temporisé à la* UPPAAL. *L'ensemble d'états* N_T *est appelé l'ensemble des* états rejetants. *L'ensemble* $X_0 \subseteq X$ *est l'ensemble des horloges qui sont initialisées à zéro lors de toute exécution impliquant* T.

La sémantique d'un automate de test est définie de la même façon que celle d'un automate temporisé à la UPPAAL, c'est-à-dire *via* le système de transitions associé.

Intuitivement, un automate de test interagit avec un automate temporisé ou un réseau d'automates temporisés ou bien plus généralement avec un système de transitions temporisé en communiquant avec lui. En général, l'état initial de l'automate de test qui convient pour tester une propriété donnée dépendra de cette propriété. Par exemple, il est raisonnable d'attendre que l'automate de test pour une propriété du type « la valeur de l'horloge x est égale à la valeur de l'horloge y » va avoir x et y parmi ses horloges et que leurs valeurs courantes vont être données, sans réinitialisation, par l'environnement que nous cherchons à tester.

C'est la raison pour laquelle nous allons autoriser plusieurs états initiaux pour les automates de test et nous associons à chaque automate de test un ensemble d'*horloges privées*, l'ensemble X_0. Ces horloges appartiennent à l'automate de test et sont remises à zéro à chaque entrée dans l'automate. Une *valuation initiale* pour T est une valuation de l'ensemble d'horloges X qui associe à toute horloge de X_0 la valeur 0. Un *état initial* de T est donc un couple (n_0, v_0) où v_0 est une valuation initiale de T.

Étant donné un état initial pour un automate de test, la dynamique de l'interaction entre l'automate de test et le système testé est décrite par la composition parallèle, restreinte aux actions de Act, des systèmes de transitions temporisés associés à l'automate de test et du système temporisé que nous testons.

Nous définissons maintenant précisément la notion de succès et d'échec pour un test.

Définition 9.10 *Soient* $\mathcal{T}$ *un système de transitions temporisé et* T *un automate de test.*

– *Un état* n *de* T *est dit* accessible *à partir d'un état* $(s_1 \parallel s_2)$ *de* $(\mathcal{T} \parallel \mathcal{T}_T)$ *si et seulement s'il existe un état* s'_1 *de* $\mathcal{T}$ *et une valuation* v *des horloges de* T *tels que l'état* $(s'_1 \parallel (n, v))$ *soit accessible à partir de* $(s_1 \parallel s_2)$ *dans* $(\mathcal{T} \parallel \mathcal{T}_T)$.

– *Un état* s *de* $\mathcal{T}$ ne passe pas le test T *à partir de l'état initial* (n_0, u_0) *si et seulement si un état rejetant de* T *est accessible dans* $(\mathcal{T} \parallel \mathcal{T}_T)$ *à partir de l'état* $(s \parallel (n_0, u_0))$.

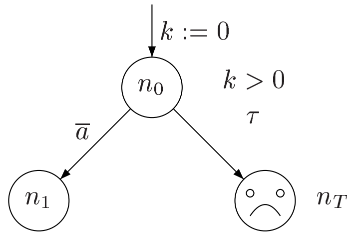
  Sinon, nous disons que s passe le test T *de l'état initial* (n_0, u_0).

Dans le reste de ce chapitre, les systèmes de transitions temporisés que nous considérerons seront souvent des systèmes de transitions provenant des automates temporisés. Dans ce cas, nous nous autoriserons à noter $(\mathcal{A} \parallel T)$ au lieu de $(\mathcal{T}_\mathcal{A} \parallel \mathcal{T}_T)$. L'état initial de T dont nous partirons sera toujours indiqué clairement.

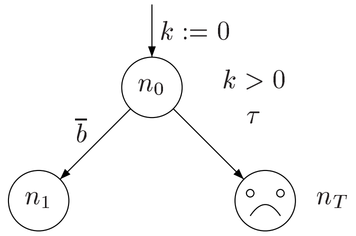
***Exemple* 9.11** Considérons l'automate temporisé $\mathcal{A}$ de l'exemple 9.6 (figure 9.3) et l'automate de test T_b ($b \notin \Sigma_u$) de la figure 9.4(b), où nous étiquetons la flèche arrivant dans l'état initial n_0 de T_b par la remise à zéro $k := 0$ pour dénoter le fait que l'horloge k est contenue dans X_0 (cette convention sera utilisée tout au long du chapitre). L'état rejetant n_T de l'automate de test est accessible de l'état initial de $(A \parallel T_b)$. En effet, comme b n'est pas une action urgente, il est possible de laisser s'écouler le temps alors que les deux automates sont dans leurs états initiaux. Ceci permet alors à l'automate T_b de faire l'action interne τ à partir de l'état n_0 de T_b : l'état n_T est bien accessible. Dans ce cas, nous disons que $\mathcal{A}$ ne passe pas le test T_b.

[3] Nous notons l'ensemble des états de l'automate de test N au lieu de Q pour bien distinguer les automates de test des autres automates représentant les systèmes que nous étudions.

Si nous testons $\mathcal{A}$ en utilisant l'automate T_a ($a \in \Sigma_u$) au lieu de T_b (figure 9.4(a)), alors, $\mathcal{A}$ et T_a doivent se synchroniser sur a et, comme a est une action urgente, il n'est pas possible d'attendre avant la synchronisation. Nous en déduisons alors que l'état rejetant n_T de T_a n'est pas accessible et donc que $\mathcal{A}$ passe le test représenté par T_a.



(a) Automate de test T_a ($a \in \Sigma_u$)



(b) Automate de test T_b ($b \notin \Sigma_u$)

Figure 9.4: Les automates de test T_a et T_b

9.2.2 Comment tester des propriétés ?

Nous venons de voir comment il est possible de réaliser des tests sur les systèmes de transitions temporisés. Nous aimerions maintenant utiliser ces automates de test pour déterminer si, étant donné un ensemble de propriétés $\mathbb{L}$, un état d'un système de transitions satisfait une formule de $\mathbb{L}$ ou pas. Comme il a déjà été remarqué, cette approche du model-checking n'est pas juste une curiosité théorique, mais c'est le moyen de savoir quelles sont les propriétés qu'il est possible de tester en utilisant seulement de l'accessibilité (comme il est d'usage dans UPPAAL).

Définition 9.12 (Propriétés que nous pouvons tester) *Soient $\mathbb{L}$ un ensemble de propriétés, φ une formule de $\mathbb{L}$, et $T = (N, X, X_0, \mathsf{Act}_\tau, n_0, N_T, E)$ un automate de test.*

- *Pour chaque état s d'un système de transitions temporisé $\mathcal{T}$ et pour chaque valuation u d'un ensemble d'horloges adéquats*[4] *contenant X, nous disons que (s, u)* passe le test *T si et seulement si aucun état rejetant de T n'est accessible à partir de l'état $(s \parallel (n_0, [X_0 \leftarrow 0](u \upharpoonright X)))$*[5] *dans $(\mathcal{T} \parallel \mathcal{T}_T)$.*
- *Nous disons que l'automate de test T* teste *la formule φ (et que la formule φ est* testable*) si pour chaque système de transitions $\mathcal{T}$, pour chaque état s de $\mathcal{T}$ et pour chaque valuation u d'au moins les horloges de T,*

$$(s, u) \models \varphi \text{ si et seulement si } (s, u) \text{ passe le test } T \tag{9.1}$$

Si la condition (9.1) n'est vérifiée que pour les systèmes de transitions provenant d'automates temporisés, nous disons que l'automate de test T teste *la formule φ (et que φ est* testable*) pour les automates temporisés.*
Un ensemble de propriétés est testable *si toute propriété de cet ensemble est testable.*

Nous obtenons immédiatement la propriété suivante :

[4]Nous disons que l'ensemble d'horloges est adéquat car il va dépendre du langage $\mathbb{L}$ que nous allons considérer, comme nous le verrons plus loin.

[5]La notation $u \upharpoonright X$ a été définie à la page 29 et représente la restriction de la valuation u à l'ensemble d'horloges X.

Proposition 9.13 *Soit $\mathbb{L}$ un ensemble de propriétés. Si $\mathbb{L}$ est testable alors pour tout système de transitions temporisé $\mathcal{T}$, pour tout état s de $\mathcal{T}$, pour toute valuation u, et pour toute propriété $\varphi \in \mathbb{L}$,*

$$(s,u) \models \varphi \text{ si et seulement si } \forall s'.\ s \xrightarrow{\tau}^{*} s' \implies (s',u) \models \varphi$$

PREUVE : Il nous suffit de montrer l'implication « seulement si » de l'équivalence. Supposons que $(s,u) \models \varphi$ mais qu'il existe un état s' tel que $s \xrightarrow{\tau}^{*} s'$ et $(s',u) \not\models \varphi$. Comme $\mathbb{L}$ est testable, un état rejetant de T_φ (T_φ est l'automate de test pour la formule φ) est accessible à partir de l'état $(s' \parallel (n_0, [X_0 \to 0](u \upharpoonright X)))$ (n_0 est l'état initial de T_φ) dans $(\mathcal{T} \parallel \mathcal{T}_{T_\varphi})$. Ainsi, un état rejetant est aussi accessible à partir de l'état $(s \parallel (n_0, [X_0 \to 0](u \upharpoonright X)))$. Comme $\mathbb{L}$ est testable, cela contredit notre hypothèse que $(s,u) \models \varphi$. □

9.3 Les langages de propriétés

Dans ce qui suit, nous considérons un *langage de propriétés* (c'est-à-dire un ensemble de propriétés) adapté à la spécification de propriétés de sûreté et de vivacité bornée pour les systèmes de transitions temporisés. Ce langage de propriétés est un fragment de la logique L_ν présentée dans [LLW95] (*cf* la partie 7.1.2) qui prend en outre en compte les actions urgentes et non urgentes.

Dans un souci de clarté, nous commençons par présenter un premier candidat naturel pour notre langage de propriétés, appelé SBLL, et nous montrons que cet ensemble est testable (*cf* les parties 9.3.1 et 9.3.2). Nous montrons ensuite que le formalisme des automates de test est strictement plus expressif que SBLL (partie 9.3.3). Le langage de propriétés $\mathcal{L}_{\forall S}$, dont nous montrerons la complétude dans la partie 9.4, est alors introduit dans la partie 9.3.4. Son expressivité est étudiée dans les parties 9.3.5 et 9.3.6. Enfin, la testabilité de $\mathcal{L}_{\forall S}$ est prouvée dans la partie 9.3.7.

Dans toute cette partie, nous dirons langage pour langage de propriétés.

9.3.1 Le langage SBLL

Convention. Dans ce qui suit, nous supposerons que les horloges utilisées dans les automates de test proviennent d'un ensemble fixé et dénombrable X_T alors que les horloges des autres automates proviennent d'un autre ensemble fixé et dénombrable $X_{\mathcal{A}}$, disjoint de X_T.

Définition 9.14 (Le langage SBLL) *Soit K un ensemble infini dénombrable d'horloges, disjoint de $X_{\mathcal{A}}$ et incluant X_T. Nous notons* `fail` *une action non contenue dans* Act_τ. *Le langage SBLL sur l'ensemble K est engendré par la grammaire suivante :*

$$\begin{array}{lll} \varphi & ::= & \mathtt{ff} \mid \varphi_1 \wedge \varphi_2 \mid g \vee \varphi \mid \forall\!\!\forall\varphi \mid [a]\varphi \mid \\ & & \langle a\rangle \mathtt{tt}\ \ (a \in \Sigma_u) \mid x\ \underline{\mathsf{in}}\ \varphi \mid Z \mid \mathsf{max}(Z,\varphi) \\ g & ::= & x \sim p \mid x - y \sim p \end{array}$$

où $a \in \mathsf{Act} \cup \{\mathtt{fail}\}$, $x, y \in K$, $p \in \mathbb{N}$, $\sim \in \{<,>,=\}$, Z est une variable et $\mathsf{max}(Z,\varphi)$ représente la solution maximale de l'équation $Z = \varphi$.

Une *formule récursive close* du langage SBLL est une formule dans laquelle chaque occurrence d'une variable Z apparaît sous un opérateur $\mathsf{max}(Z,\varphi)$. Étant donnée une formule non close, une variable qui n'est pas sous un opérateur max est dite *libre*. Nous utilisons la notation SBLL$^-$ pour représenter l'ensemble des formules récursives closes de SBLL qui ne contiennent pas d'occurrences du type $\langle a \rangle \mathtt{tt}$. Nous notons aussi $\mathsf{clocks}(\varphi)$ l'ensemble des horloges apparaissant dans la formule φ.

Reprenant ce qui a été fait dans [LLW95], les formules récursives closes de SBLL sont interprétées sur les couples de la forme (s, u), où s est un état d'un système de transitions temporisé et u est une valuation des horloges de K.

Nous définissons alors la relation de satisfaction pour SBLL comme étant la plus grande relation, notée $\models$, qui satisfait les implications du tableau 9.5.

$$
\begin{array}{rcl}
(s,u) \models \mathtt{ff} & \Longrightarrow & \mathsf{Faux} \\
(s,u) \models \varphi_1 \wedge \varphi_2 & \Longrightarrow & \forall s'.\ s \xrightarrow{\tau}^{*} s' \text{ implique } (s',u) \models \varphi_1 \text{ et } (s',u) \models \varphi_2 \\
(s,u) \models g \vee \varphi & \Longrightarrow & g(u) \text{ ou } \forall s'.\ s \xrightarrow{\tau}^{*} s' \text{ implique } (s',u) \models \varphi \\
(s,u) \models [a]\varphi & \Longrightarrow & \forall s'.\ s \xRightarrow{a} s' \text{ implique } (s',u) \models \varphi \\
(s,u) \models \langle a \rangle \mathtt{tt}\ (a \in \Sigma_u) & \Longrightarrow & \forall s'.\ s \xrightarrow{\tau}^{*} s' \text{ implique } s' \xrightarrow{a} s'' \text{ pour un } s'' \\
(s,u) \models \forall\!\!\forall \varphi & \Longrightarrow & \forall d \in \mathbb{R}^{+}\ \forall s'.\ s \xRightarrow{\epsilon(d)} s' \text{ implique } (s',u+d) \models \varphi \\
(s,u) \models x \ \underline{\text{in}}\ \varphi & \Longrightarrow & \forall s'.\ s \xrightarrow{\tau}^{*} s' \text{ implique } (s',[x \leftarrow 0]u) \models \varphi \\
(s,u) \models \mathsf{max}(Z,\varphi) & \Longrightarrow & \forall s'.\ s \xrightarrow{\tau}^{*} s' \text{ implique } (s',u) \models \varphi\{\mathsf{max}(Z,\varphi)/Z\}
\end{array}
$$

Tableau 9.5: Relation de satisfaction pour SBLL

Notons que cette interprétation est légèrement différente de celle de SBLL définie dans [LLW95] ne correspond pas à ce que nous voulons faire (grossièrement, l'interprétation de nos formules doit être « close par les actions internes »).

9.3.2 Tester SBLL

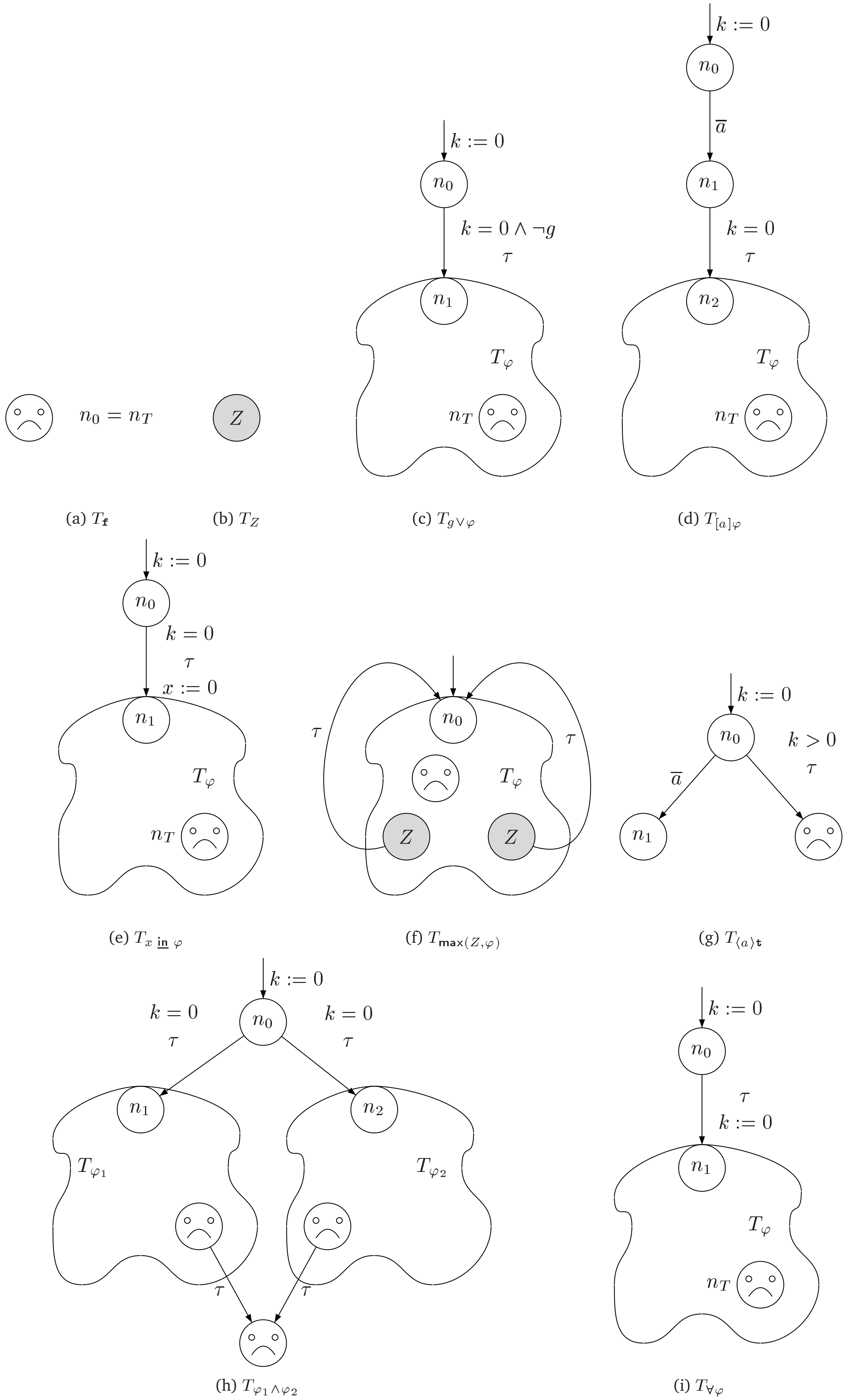
Notre but est d'utiliser le langage que nous venons juste de définir comme langage de spécification et de réduire son model-checking à un test d'accessibilité. Dans ce but, nous définissons une procédure de « compilation » qui va permettre de calculer un automate de test à partir des formules que nous voulons tester. Par le biais de cette procédure de compilation, nous automatisons la génération d'automates de test à partir de nos spécifications logiques, phase qui pourrait être difficile à réaliser sans cette procédure.

L'un de nos résultats importants est que le langage SBLL est testable (voir la définition 9.12). Plus précisément, nous avons le théorème suivant :

Théorème 9.15 *Pour chaque formule close φ de SBLL^{-}, il existe un automate de test T_φ, sur l'ensemble d'horloges $\{k\} \cup \mathsf{clocks}(\varphi)$, où k est une horloge n'apparaissant pas dans $\mathsf{clocks}(\varphi)$, qui teste la formule φ.*

*Pour chaque formule close φ de SBLL, il existe un automate de test T_φ, sur l'ensemble d'horloges $\{k\} \cup \mathsf{clocks}(\varphi)$, où k est une horloge n'apparaissant pas dans $\mathsf{clocks}(\varphi)$, qui teste la formule φ **pour les automates temporisés**.*

La construction de l'automate T_φ se fait par induction sur la structure de la formule φ. Les différents cas sont représentés de manière schématique sur la figure 9.6. La preuve complète de



(a) $T_{\mathtt{f}}$ (b) T_Z (c) $T_{g\vee\varphi}$ (d) $T_{[a]\varphi}$

(e) $T_{x\ \underline{\mathtt{in}}\ \varphi}$ (f) $T_{\mathsf{max}(Z,\varphi)}$ (g) $T_{\langle a\rangle\mathtt{t}}$

(h) $T_{\varphi_1\wedge\varphi_2}$ (i) $T_{\forall\varphi}$

Figure 9.6: Automates de test pour les sous-formules de SBLL correction de ces constructions, longue et un peu fastidieuse, est proposée dans l'annexe A de ce chapitre, à la page 207.

Il est maintenant naturel de se demander si chaque propriété φ qui est testable de cette manière peut être exprimée par une formule du langage SBLL. Cela nous amène à nous demander si chaque automate de test T peut s'exprimer dans le langage SBLL, dans le sens où il existe une formule ψ_T dans SBLL telle que chaque automate temporisé $\mathcal{A}$ passe le test T si et seulement si $\mathcal{A}$ satisfait ψ_T. En effet, si nous avions la réponse à cette question, alors nous obtiendrions une caractérisation complète de l'ensemble des propriétés que nous pouvons vérifier grâce aux automates de test.

9.3.3 SBLL **n'est pas suffisant !**

Nous commençons par montrer que les automates de test ont un pouvoir d'expression plus important que le langage de spécification SBLL. Par exemple, considérons l'automate de test T dessiné sur la figure 9.7 (a est supposée être une action urgente).

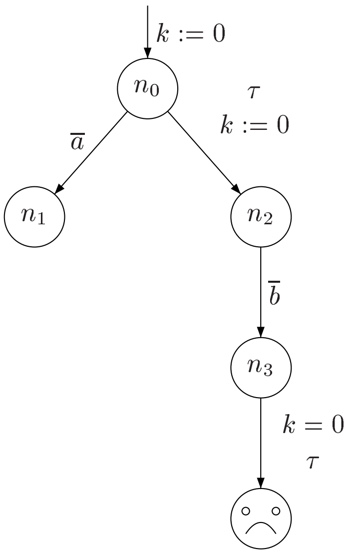


Figure 9.7: Un automate de test, T, qui ne peut pas être exprimé dans SBLL ($a \in \Sigma_u$)

Nous verrons bientôt (*cf* la preuve du théorème 9.26) qu'il y a deux automates temporisés qui vérifient les mêmes propriétés de SBLL, mais qui se comportent différemment en présence de l'automate de test T. En particulier, l'un de ces automates passe le test T, alors que l'autre ne le passe pas. Ceci montre que la propriété testée par l'automate de la figure 9.7 ne peut pas s'exprimer dans le langage SBLL. Intuitivement, la propriété qui est testée par l'automate T exprime le fait que, en attendant sans pouvoir faire une action a, un état ne peut permettre que d'aller dans un état où l'action b ne peut pas être effectuée.

Proposition 9.16 *Soit s un état d'un système de transitions temporisé $\mathcal{T}$ Alors s passe le test de la figure 9.7 si et seulement si, pour tout état s' de $\mathcal{T}$ et pour tout réel $d \in \mathbb{R}^+$, $s \overset{\epsilon(d)}{\Longrightarrow}_{\{a\}} s'$ implique $s' \overset{b}{\not\Longrightarrow}$.*

Preuve : Nous montrons les deux implications séparément.

– IMPLICATION « SEULEMENT SI » : Nous montrons la contraposée. À cette fin, nous supposons qu'il existe un état s' et une attente $d \in \mathbb{R}^+$ tels que $s \overset{\epsilon(d)}{\Longrightarrow}_{\{a\}} s'$ et $s' \overset{b}{\Longrightarrow}$. Nous montrons que s' ne passe pas le test T.
Tout d'abord, notons que dans $(\mathcal{T} \parallel \mathcal{T}_T)$, nous avons

$$(s \| (n_0, [k \leftarrow 0])) \overset{\epsilon(d)}{\Longrightarrow} (s' \| (n_0, [k \leftarrow d]))$$

car $s \overset{\epsilon(d)}{\Longrightarrow}_{\{a\}} s'$ et $(n_0, [k \leftarrow 0]) \xrightarrow{\epsilon(d)} (n_0, [k \leftarrow d])$. En utilisant l'hypothèse que $s' \overset{b}{\Longrightarrow} s''$ pour un état s'', nous pouvons maintenant compléter l'exécution précédente de la manière suivante :

$$\begin{array}{rcl}(s' \parallel (n_0, [k \leftarrow d])) & \xrightarrow{\tau} & (s' \parallel (n_2, [k \leftarrow 0])) \\ & \overset{\tau}{\Longrightarrow} & (s'' \parallel (n_3, [k \leftarrow 0])) \\ & \xrightarrow{\tau} & (s'' \parallel (n_T, [k \leftarrow 0])).\end{array}$$

Ainsi, l'état rejetant de T est accessible, et s ne passe pas le test T.

– IMPLICATION « SI » : De nouveau, nous montrons la contraposée. À cette fin, supposons que s ne passe pas le test T. Nous montrons qu'il existe un état s' et une attente $d \in \mathbb{R}^+$ tels que $s \overset{\epsilon(d)}{\Longrightarrow}_{\{a\}} s'$ et $s' \overset{b}{\Longrightarrow}$.
Comme s ne passe pas le test T, il existe dans $(\mathcal{T} \parallel \mathcal{T}_T)$ une exécution de la forme

$$(s \parallel (n_0, [k \leftarrow 0])) \quad \rightarrow^* \quad (s'' \parallel (n_T, [k \rightarrow d'])) \tag{9.2}$$

pour un état s'' et une attente $d' \in \mathbb{R}^+$. Considérons un tel chemin de longueur minimale. Il n'est pas difficile de voir que la dernière transition d'une telle exécution doit être de la forme

$$(s_2 \| (n_3, [k \leftarrow 0])) \xrightarrow{\tau} (s_2 \| (n_T, [k \leftarrow 0]))$$

pour un état s_2. Il s'ensuit que $d' = 0$. Ainsi, (9.2) a la forme

$$\begin{array}{rcl}(s \| (n_0, [k \leftarrow 0])) & \rightarrow^* & (s_1 \| (n_0, [k \leftarrow d])) \\ & \xrightarrow{\tau} & (s_1 \| (n_2, [k \leftarrow 0])) \\ & \rightarrow^* & (s_2 \| (n_3, [k \leftarrow 0])) \\ & \xrightarrow{\tau} & (s_2 \| (n_T, [k \leftarrow 0]))\end{array}$$

pour un $d \in \mathbb{R}^+$ et un état s_1. Le fragment initial de cette exécution

$$(s \| (n_0, [k \leftarrow 0])) \rightarrow^* (s_1 \| (n_0, [k \leftarrow d]))$$

peut juste être constitué de délais synchronisés de somme d, et de τ-transitions provenant du membre de gauche de la composition parallèle. Comme a est urgente et $(n_0, [k \rightarrow d'']) \xrightarrow{\bar{a}}$, pour un $d'' \in \mathbb{R}^+$, aucune action a n'est autorisée à partir de s durant cette exécution d'attente. Il s'ensuit que $s \overset{\epsilon(d)}{\Longrightarrow}_{\{a\}} s_1$. Comme la transition étiquetée par b à partir de l'état n_2 ne remet pas à zéro l'horloge k, l'exécution

$$(s_1 \| (n_2, [k \leftarrow 0])) \rightarrow^* (s_2 \| (n_3, [k \leftarrow 0]))$$

provient d'une synchronisation de $s_1 \overset{b}{\Longrightarrow} s_2$ avec $(n_2, [k \leftarrow 0]) \xrightarrow{\bar{b}} (n_3, [k \leftarrow 0])$. Nous pouvons alors en déduire que $s \overset{\epsilon(d)}{\Longrightarrow}_{\{a\}} s_1 \overset{b}{\Longrightarrow} s_2$, ce que nous voulions montrer.

La preuve est maintenant complète. □

Le type d'automate de test dessiné à la figure 9.7 suggère un enrichissement du langage SBLL, dans lequel l'opérateur $\forall\!\!\!\forall$ serait paramétré par un ensemble d'actions urgentes. Les éléments de cet ensemble d'actions urgentes ne peuvent pas être autorisés au niveau d'un état d'attente. Comme nous allons le voir dans la suite (théorème 9.38), cette simple extension de SBLL va permettre d'obtenir un langage complet pour le test *via* les automates de test.

9.3.4 Un enrichissement de SBLL, le langage $\mathcal{L}_{\forall S}$

Le langage que nous allons étudier est une extension de SBLL. Il est très proche de la logique modale L_ν présentée dans [LLW95] et utilisée ensuite dans [LL95]. La complexité du model-checking de ces langages a été étudiée dans [AL99, AL02] et L_ν est utilisé comme langage de spécification dans l'outil CMC [LL98] (nous avons rappelé tous ces faits dans le chapitre 7).

Rappelons que X_T représente un ensemble d'horloges d'où proviennent toutes les horloges des automates de test alors que $X_{\mathcal{A}}$ (disjoint de X_T) représente un ensemble d'horloges d'où proviennent toutes les horloges des automates temporisés qui vont être testés.

Définition 9.17 *Soit K un ensemble infini dénombrable d'horloges, disjoint de $X_{\mathcal{A}}$ et incluant X_T. Nous utilisons* `fail` *pour représenter une action non contenue dans* Act_τ*. L'ensemble $\mathcal{L}_{\forall S}$ des formules sur K est engendré par la grammaire suivante :*

$$\begin{array}{lcl}\varphi & ::= & \mathtt{ff} \;\mid\; \varphi_1 \wedge \varphi_2 \;\mid\; g \vee \varphi \;\mid\; \forall\!\!\forall_S \varphi \;\mid\; [a]\varphi \;\mid \\ & & \langle a\rangle \mathtt{tt}\;\; (a \in \Sigma_u) \;\mid\; x \;\underline{\mathsf{in}}\; \varphi \;\mid\; Z \;\mid\; \mathsf{max}(Z,\varphi) \\ g & ::= & x \sim p \;\mid\; x - y \sim p\end{array}$$

où $S \subseteq \Sigma_u$, $a \in \mathsf{Act} \cup \{\mathtt{fail}\}$, $x, y \in K$, $p \in \mathbb{N}$, $\sim \in \{<, >, =\}$, Z est une variable et $\mathsf{max}(Z,\varphi)$ représente la solution maximale de l'équation $Z = \varphi$. Nous utilisons $\mathcal{L}^-_{\forall S}$ pour représenter l'ensemble des formules de $\mathcal{L}_{\forall S}$ qui ne contiennent pas d'occurrence de $\langle a\rangle\mathtt{tt}$.

Comme pour SBLL, nous définissons une *formule récursive close* de $\mathcal{L}_{\forall S}$ comme étant une formule dans laquelle toute variable Z apparaît sous un opérateur $\mathsf{max}(Z,\varphi)$. Dans le reste de ce chapitre, chaque formule sera close, à moins que le contraire soit précisé. Comme pour SBLL, nous notons $\mathsf{clocks}(\varphi)$ l'ensemble des horloges apparaissant dans la formule φ. Une horloge x est libre dans la formule ϕ s'il y a une occurrence de x dans ϕ qui n'est pas sous un opérateur $x \;\underline{\mathsf{in}}\; \varphi$. Une formule est *close pour les horloges* si elle n'a pas d'horloges libres.

Étant donné un système de transitions temporisé $\mathcal{T} = (S, \Gamma, s_0, \longrightarrow)$, nous interprétons les formules closes de $\mathcal{L}_{\forall S}$ sur des couples de la forme (s,u) où s est un état de $\mathcal{T}$ et u est une valuation des horloges de K. La relation de satisfaction $\models$ est la plus grande relation satisfaisant les implications de la table 9.8. Une relation satisfaisant ces implications est appelée une *relation de satisfaisabilité*.

$$\begin{array}{rcl}(s,u) \models \mathtt{ff} & \Longrightarrow & \text{Faux} \\ (s,u) \models \varphi_1 \wedge \varphi_2 & \Longrightarrow & \forall s'.\ s \xrightarrow{\tau}{}^{*} s' \text{ implique } (s',u) \models \varphi_1 \text{ et } (s',u) \models \varphi_2 \\ (s,u) \models g \vee \varphi & \Longrightarrow & g(u) \text{ ou } \forall s'.\ s \xrightarrow{\tau}{}^{*} s' \text{ implique } (s',u) \models \varphi \\ (s,u) \models [a]\varphi & \Longrightarrow & \forall s'.\ s \xRightarrow{a} s' \text{ implique } (s',u) \models \varphi \\ (s,u) \models \langle a\rangle\mathtt{tt}\ (a \in \Sigma_u) & \Longrightarrow & \forall s'.\ s \xrightarrow{\tau}{}^{*} s' \text{ implique } s' \xrightarrow{a} s'' \text{ pour un } s'' \\ (s,u) \models \forall\!\!\forall_S \varphi & \Longrightarrow & \forall d \in \mathbb{R}^+\ \forall s'.\ s \xRightarrow{\epsilon(d)}_S s' \text{ implique } (s', u+d) \models \varphi \\ (s,u) \models x \;\underline{\mathsf{in}}\; \varphi & \Longrightarrow & \forall s'.\ s \xrightarrow{\tau}{}^{*} s' \text{ implique } (s', [x \leftarrow 0]u) \models \varphi \\ (s,u) \models \mathsf{max}(Z,\varphi) & \Longrightarrow & \forall s'.\ s \xrightarrow{\tau}{}^{*} s' \text{ implique } (s',u) \models \varphi\{\mathsf{max}(Z,\varphi)/Z\}\end{array}$$

Tableau 9.8: Relation de satisfaction pour $\mathcal{L}_{\forall S}$

De la théorie générale des points fixes [Tar55, Ace94], il vient que $\models$ est l'union de toutes les relations de satisfaisabilité et que, pour $\models$, les implications ($\Longrightarrow$) sont en fait des équivalences ($\Longleftrightarrow$).

Nous disons que $\mathcal{T}$ satisfait φ, ce que nous écrivons $\mathcal{T} \models \varphi$, si $(s_0, K \leftarrow 0) \models \varphi$. Dans la suite, pour un automate temporisé $\mathcal{A}$, nous écrirons $\mathcal{A} \models \varphi$ à la place de $\mathcal{T}_{\mathcal{A}} \models \varphi$.

Remarque : Comme `fail` n'appartient pas à Act, tout état d'un système de transitions temporisé satisfait les formules de la forme $[\texttt{fail}]\phi$. Le rôle joué par ces formules deviendra plus clair dans la partie 9.4.

Le lemme suivant établit une propriété de base de la relation de satisfaction.

Lemme 9.18 *Si u et v sont deux valuations qui assignent aux horloges libres d'une formule φ la même valeur, alors pour tout état s d'un système de transitions temporisé,*

$$(s, u) \models \varphi \ \textit{implique} \ (s, v) \models \varphi$$

PREUVE : Soit u une valuation. La relation $\mathcal{R}_u$ définie par :

$$\mathcal{R}_u \stackrel{\text{def}}{=} \{((s,v),\varphi) \mid (s,u) \models \varphi \text{ et } u, v \text{ assignent la même valeur aux horloges libres de } \varphi\}$$

est une relation de satisfaisabilité. La vérification facile de ce fait n'est pas détaillée ici. □

Si la formule ϕ est close pour les horloges, alors la véracité de $(s, u) \models \phi$ est indépendante de la valuation u (lemme 9.18). Dans ce cas, nous écrirons souvent simplement $s \models \phi$ à la place de $(s, u) \models \phi$. Il est aussi évident que la véracité de $(s, u) \models \phi$ dépend uniquement de la restriction de u à l'ensemble des horloges apparaissant dans φ. Ainsi, dans les preuves, nous écrirons souvent $(s, u \restriction \mathsf{clocks}(\varphi)) \models \varphi$ à la place de $(s, u) \models \varphi$.

La relation de satisfaction est close par la relation de transition $\xrightarrow{\tau}^*$, dans le sens de la proposition suivante.

Proposition 9.19 *Soit $\mathcal{T} = (\mathcal{S}, \Gamma, s_0, \longrightarrow)$ un système de transitions temporisé. Alors, pour tout $s \in \mathcal{S}$, pour tout $\varphi \in \mathcal{L}_{\forall S}$ et pour toute valuation u des horloges de K, $(s, u) \models \varphi$ si et seulement si pour tout s' tel que $s \xrightarrow{\tau}^* s'$, $(s', u) \models \varphi$.*

PREUVE : La seule chose à vérifier est que, si $(s, u) \models \varphi$ et $s \xrightarrow{\tau}^* s'$, alors $(s', u) \models \varphi$. Dans ce but, il est suffisant de montrer que la relation $\mathcal{R}$ définie par :

$$\mathcal{R} \stackrel{\text{def}}{=} \{((s,u),\varphi) \mid \exists t.\ (t,u) \models \varphi \text{ et } t \xrightarrow{\tau}^* s\} \cup \models$$

est une relation de satisfaisabilité. La vérification assez aisée n'est pas détaillée. □

Cette proposition recoupe ce qui était demandé dans la proposition 9.13. Notre définition de la relation de satisfaction est quelque peu différente de celle communément présentée dans la littérature. Par exemple, nous pourrions nous attendre à ce que la formule $\langle a \rangle \mathtt{tt}$ soit interprétée de la manière suivante :

$$(s, u) \models \langle a \rangle \mathtt{tt} \quad \text{implique} \quad s \stackrel{a}{\Longrightarrow} s' \text{ pour un certain } s'. \tag{9.3}$$

Rappelons cependant que le but de ce travail est de proposer un langage de spécification pour les automates temporisés pour lequel le model-checking peut être réduit à un test d'accessibilité. Ayant ce but en tête, une proposition raisonnable pour un automate de test pour la formule $\langle a \rangle \mathtt{tt}$, interprétée de manière standard comme en (9.3), est l'automate dessiné sur la figure 9.4(a). Cependant, il n'est pas difficile de voir qu'un tel automate pourrait être emmené jusqu'à son état rejetant n_T par une interaction avec l'automate temporisé de la figure 9.9 (représentant le processus TCCS $a + \tau$). Ceci est dû au fait que, à cause de la définition de la composition parallèle, un automate de test ne peut pas empêcher le système testé d'effectuer des actions internes menant à un état à partir duquel aucune action a n'est possible.

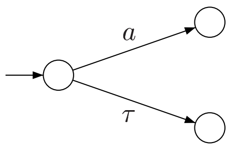


Figure 9.9: Automate temporisé associé au processus TCCS $a + \tau$

9.3.5 Propriétés de base de $\mathcal{L}_{\forall S}$

Définition 9.20 *Soit $\mathcal{E}$ un ensemble de systèmes de transitions temporisés.*

– *Deux formules ϕ et ψ de $\mathcal{L}_{\forall S}$ sont dites* équivalentes *vis-à-vis de $\mathcal{E}$ si et seulement si pour tout état s d'un système de transitions temporisé de $\mathcal{E}$ et pour toute valuation w des horloges des formules,*

$$(s, w) \models \phi \iff (s, w) \models \psi.$$

– *Soient $\mathbb{L}$ et $\mathbb{L}'$ deux sous-ensembles de $\mathcal{L}_{\forall S}$. Nous disons que $\mathbb{L}$ est* au moins aussi expressif *que $\mathbb{L}'$ vis-à-vis de $\mathcal{E}$ si et seulement si pour toute formule ϕ de $\mathbb{L}'$, il existe une formule ψ de $\mathbb{L}$ qui est équivalente à ϕ vis-à-vis de $\mathcal{E}$.*
– *Si $\mathbb{L}$ est au moins aussi expressif que $\mathbb{L}'$ vis-à-vis de $\mathcal{E}$, et s'il y a au moins une formule ϕ dans $\mathbb{L}$ qui n'est équivalente à aucune formule de $\mathbb{L}'$ vis-à-vis de $\mathcal{E}$, alors nous disons que $\mathbb{L}$ est* strictement plus expressif *que $\mathbb{L}'$ vis-à-vis de $\mathcal{E}$.*
– *Les langages $\mathbb{L}$ et $\mathbb{L}'$ sont dits* expressivement équivalents *vis-à-vis de $\mathcal{E}$ si et seulement si $\mathbb{L}$ est au moins aussi expressif que $\mathbb{L}$ vis-à-vis de $\mathcal{E}$, et vice versa.*

Dans la suite, nous distinguons en particulier deux ensembles de systèmes de transitions temporisés :
– l'ensemble de tous les systèmes de transitions temporisés, que nous noterons STT et
– l'ensemble des systèmes de transitions temporisés provenant des automates temporisés, que nous noterons $\mathsf{STT}_{\text{aut}}$.

Comme, pour tout automate temporisé $\mathcal{A}$, la structure $\mathcal{T}_{\mathcal{A}}$ est un système de transitions temporisé (proposition 9.8), il vient que si deux formules sont équivalentes vis-à-vis de STT, alors alors elles sont aussi équivalentes vis-à-vis de $\mathsf{STT}_{\text{aut}}$. Nous verrons un peu plus tard que la réciproque n'est pas vraie.

Le lemme suivant, dont nous ne développons pas la preuve, rassemble tous les résultats de base sur l'équivalence entre formules, certains de ces résultats seront utiles pour les preuves de la partie 9.4.

Lemme 9.21

1. *Pour chaque formule φ et pour toutes les horloges x, y, la formule x $\underline{\text{in}}$ $(y$ $\underline{\text{in}}$ $\varphi)$ est équivalente à y $\underline{\text{in}}$ $(x$ $\underline{\text{in}}$ $\varphi)$ vis-à-vis de* STT.

2. *Pour chaque formule φ et pour chaque ensemble d'actions urgentes S, la formule $\forall\!\!\!\forall_S\varphi$ est équivalente à $\forall\!\!\!\forall_S\forall\!\!\!\forall_S\varphi$ vis-à-vis de* STT.
3. *Pour toutes les formules φ, ϕ et pour toute action a, la formule $[a](\varphi \wedge \phi)$ est équivalente à $([a]\varphi) \wedge ([a]\phi)$ vis-à-vis de* STT.

Notation. La première équivalence de ce lemme nous permet d'écrire pour chaque ensemble d'horloges $\{y_1, \ldots, y_n\}$ et pour chaque formule φ, $\{y_1, \ldots, y_n\}\ \underline{\text{in}}\ \varphi$ comme « raccourci » pour $y_1\ \underline{\text{in}}\ (y_2\ \underline{\text{in}}\ \cdots (y_n\ \underline{\text{in}}\ \varphi) \cdots)$. Si $n = 0$, alors, par convention, $\varnothing\ \underline{\text{in}}\ \varphi$ est équivalent à φ. Nous écrivons aussi g à la place de $g \vee \mathtt{ff}$, et $\mathtt{tt}$ pour $x \geq 0$. Quand ϕ est une contrainte d'horloges sur K et $\psi \in \mathcal{L}_{\forall S}$, nous utilisons $\phi \implies \psi$ pour $\neg\phi \vee \psi$, où $\neg\phi$ remplace une contrainte représentant la négation de ϕ.

***Exemple* 9.22** Considérons l'automate temporisé de la figure 9.10, où a est une action urgente.

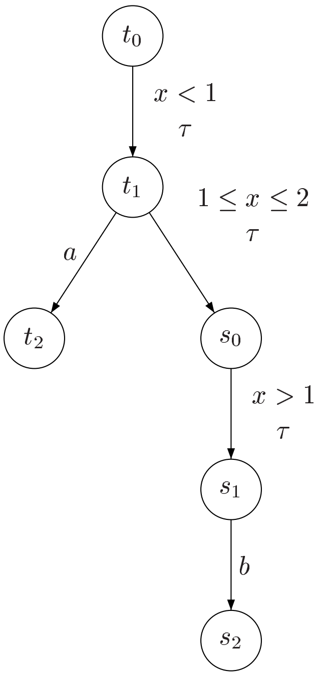


Figure 9.10: Un patron d'automate temporisé ($a \in \Sigma_u$)

Nous allons prouver que :

1. $(t_0, [x \leftarrow 0]) \models \forall\!\!\!\forall_{\{a\}}[b]\mathtt{ff}$, mais
2. $(s_0, [x \leftarrow 0]) \not\models \forall\!\!\!\forall_{\{a\}}[b]\mathtt{ff}$.

Nous commençons par montrer que $(t_0, [x \leftarrow 0])$ satisfait la formule $\forall\!\!\!\forall_{\{a\}}[b]\mathtt{ff}$. À cette fin, supposons que $(t_0, [x \leftarrow 0]) \stackrel{\epsilon(d)}{\Longrightarrow}_{\{a\}} t$ pour un état t donné. Nous allons voir que $t \stackrel{b}{\not\Longrightarrow}$ en examinant les formes possibles que peut prendre t. Nous distinguons deux cas, dépendant de si d est strictement plus petit que 1, ou pas.

– Supposons que $d < 1$. Alors soit $t = (t_0, [x \to d])$, soit $t = (t_1, [x \to d])$. Dans les deux cas, comme $d < 1$, la transition étiquetée par τ qui part de t_1 ne peut pas être prise. Nous en déduisons que $t \stackrel{b}{\not\Longrightarrow}$.
– Supposons que $d \geq 1$. Nous allons montrer que t peut seulement être $(t_0, [x \to d])$ — ce qui impliquera le résultat voulu car, comme $d \geq 1$, aucune action ne peut être faite à partir d'un tel état. Pour voir que c'est bien le cas, remarquons tout d'abord que $(t_0, [x \leftarrow 0]) \stackrel{\epsilon(d)}{\Longrightarrow}_{\{a\}} (t_0, [x \to d])$ car la seule transition sortante de t_0 est étiquetée par τ. En outre, comme $d \geq 1$, toute transition de la forme $(t_0, [x \leftarrow 0]) \stackrel{\epsilon(d)}{\Longrightarrow}_{\{a\}} t$ qui

quitte l'état t_0 doit avoir le préfixe suivant pour un $0 \leq d' < 1$:

$$(t_0, [x \leftarrow 0]) \xrightarrow{\epsilon(d')}_{\{a\}} (t_0, [x \rightarrow d']) \xrightarrow{\tau} (t_1, [x \rightarrow d'])$$

Comme $(t_1, [x \rightarrow d']) \xrightarrow{a}$ et $d' < 1$, le délai nécessaire pour pouvoir prendre la transition étiquetée par τ qui sort de t_1 ne peut pas être attendu sans pouvoir effectuer une action a. Nous en concluons donc que, comme annoncé, aucune transition de la forme $(t_0, [x \leftarrow 0]) \overset{\epsilon(d)}{\Longrightarrow}_{\{a\}} t$ avec $d \geq 1$ ne peut quitter l'état t_0.
D'un autre côté, $(s_0, [x \leftarrow 0]) \not\models \forall\!\!\forall_{\{a\}}[b]\mathtt{ff}$. En fait, par exemple,

$$(s_0, [x \leftarrow 0]) \overset{\epsilon(1.1)}{\Longrightarrow}_{\{a\}} (s_1, [x \rightarrow 1.1]) \xrightarrow{b}$$

L'exemple ci-dessus est d'un intérêt particulier car, comme nous le verrons bientôt, chaque propriété que nous pourrons exprimer dans SBLL qui est satisfaite par l'état $(t_0, [x \leftarrow 0])$, l'est aussi par l'état $(s_0, [x \leftarrow 0])$.

9.3.6 Résultats d'expressivité pour $\mathcal{L}_{\forall S}$ et $\mathcal{L}^-_{\forall S}$

Dans cette partie, nous étudions le pouvoir d'expression relatif des langages introduits précédemment. Nous commençons par voir que les langages $\mathcal{L}_{\forall S}$ et $\mathcal{L}^-_{\forall S}$ sont expressivement équivalents vis-à-vis de $\mathsf{STT}_{\mathrm{aut}}$, mais que $\mathcal{L}_{\forall S}$ est strictement plus expressif que $\mathcal{L}^-_{\forall S}$ vis-à-vis de STT. Nous verrons par la suite que la propriété $\langle a \rangle \mathtt{tt}$ ne peut pas être vérifiée par les automates de test.

Théorème 9.23 *Les langages $\mathcal{L}_{\forall S}$ et $\mathcal{L}^-_{\forall S}$ sont expressivement équivalents vis-à-vis de STT_{aut}, mais $\mathcal{L}_{\forall S}$ est strictement plus expressif que $\mathcal{L}^-_{\forall S}$ vis-à-vis de* STT.

Comme $\mathcal{L}^-_{\forall S}$ est un sous-langage de $\mathcal{L}_{\forall S}$, il est clair que chaque propriété qui peut être exprimée dans $\mathcal{L}^-_{\forall S}$ peut aussi l'être dans $\mathcal{L}_{\forall S}$. Nous montrons maintenant que l'inverse est vraie si nous nous restreignons aux $\mathsf{STT}_{\mathrm{aut}}$. Comme la seule différence réside dans la formule $\langle a \rangle \mathtt{tt}$ $(a \in \Sigma_u)$, il est suffisant de montrer que cette formule peut être codée dans $\mathcal{L}^-_{\forall S}$ (si nous nous restreignons aux $\mathsf{STT}_{\mathrm{aut}}$). C'est ce qu'exprime le résultat suivant :

Lemme 9.24

1. *Soient s un état d'un système de transitions temporisé, et w une valuation quelconque. Supposons que $(s, w) \models \langle a \rangle \mathtt{tt}$. Alors $(s, w) \models x \ \underline{\mathrm{in}}\ \forall\!\!\forall_{\{a\}}(x = 0)$.*
2. *Soient (q, u) un état d'un automate temporisé, et w une valuation quelconque. Supposons que $((q, u), w) \models x \ \underline{\mathrm{in}}\ \forall\!\!\forall_{\{a\}}(x = 0)$. Alors $((q, u), w) \models \langle a \rangle \mathtt{tt}$.*

Ainsi, la formule $\langle a \rangle \mathtt{tt}$ $(a \in \Sigma_u)$ est équivalente à $x \ \underline{\mathrm{in}}\ \forall\!\!\forall_{\{a\}}(x = 0)$ vis-à-vis de STT.

PREUVE : L'équivalence des formules $\langle a \rangle \mathtt{tt}$ et $x \ \underline{\mathrm{in}}\ \forall\!\!\forall_{\{a\}}(x = 0)$ vis-à-vis de $\mathsf{STT}_{\mathrm{aut}}$ est une conséquence immédiate des points 1 et 2 du lemme. Nous les montrons maintenant séparément.

1. Soient s un état d'un système de transitions temporisé, et w une valuation pour les horloges de formule. Supposons que $(s, w) \models \langle a \rangle \mathtt{tt}$. Nous montrons que $(s, w) \models x \ \underline{\mathrm{in}}\ \forall\!\!\forall_{\{a\}}(x = 0)$. À cette fin, supposons que
$$s \xrightarrow{\tau}{}^{*} s' \overset{\epsilon(d)}{\Longrightarrow}_{\{a\}} s''.$$
Nous souhaitons montrer que $(s'', [x \rightarrow d](w + d)) \models x = 0$, *i.e.*, que $d = 0$.
Tout d'abord, remarquons que $(s', w) \models \langle a \rangle \mathtt{tt}$ car $(s, w) \models \langle a \rangle \mathtt{tt}$ et $s \xrightarrow{\tau}{}^{*} s'$ (proposition 9.19). Une induction simple sur la longueur de la dérivation $s' \overset{\epsilon(d)}{\Longrightarrow}_{\{a\}} s''$ prouve que $d = 0$, ce que nous voulions montrer.

2. Nous montrons la propriété contraposée. À cette fin, supposons que, pour un état d'un automate temporisé (q, u) et une valuation w des horloges de formules, $((q, u), w) \not\models \langle a \rangle \mathtt{tt}$. Nous montrons que $((q, u), w) \not\models x \ \underline{\mathrm{in}}\ \mathbb{W}_{\{a\}}(x = 0)$.

   Comme $((q, u), w) \not\models \langle a \rangle \mathtt{tt}$, il existe un état (q', u') tel que $(q, u) \xrightarrow{\tau}{}^{*} (q', u') \overset{a}{\not\longrightarrow}$. Nous montrons que $((q', u'), [x \to 0]w) \not\models \mathbb{W}_{\{a\}}(x = 0)$. À cette fin, il est suffisant de trouver un réel positif d tel que $(q', u') \xrightarrow{\epsilon(d)}_{\{a\}} (q', u' + d)$. Comme $(q', u') \overset{a}{\not\longrightarrow}$ et comme a est une action urgente, l'hypothèse de persistance en arrière des actions urgentes (voir la définition du début de la partie 9.1.1) entraîne que $(q', u' + d) \overset{a}{\not\longrightarrow}$ pour chaque $d \geq 0$. Ainsi, par exemple, nous avons que $(q', u') \xrightarrow{\epsilon(1)}_{\{a\}} (q', u' + 1)$. Comme $((q', u' + 1), [x \to 1]w) \not\models x = 0$, nous avons montré ce que nous voulions.

La preuve du lemme est maintenant complète. $\square$

En utilisant le lemme 9.24, il n'est maintenant pas difficile de montrer que le langage $\mathcal{L}^{-}_{\forall S}$ est au moins aussi expressif que $\mathcal{L}_{\forall S}$ vis-à-vis de $\mathsf{STT}_{\mathrm{aut}}$. En fait, chaque formule de $\mathcal{L}_{\forall S}$ peut être transformée en une formule équivalente de $\mathcal{L}^{-}_{\forall S}$ simplement en remplaçant chaque occurrence d'une formule de la forme $\langle a \rangle \mathtt{tt}$ par $x \ \underline{\mathrm{in}}\ \mathbb{W}_{\{a\}}(x = 0)$. Nous en déduisons alors que $\mathcal{L}_{\forall S}$ et $\mathcal{L}^{-}_{\forall S}$ sont équivalents vis-à-vis de $\mathsf{STT}_{\mathrm{aut}}$.

Notons que le lemme 9.24(2) n'est plus valable pour des systèmes de transitions temporisés arbitraires. Considérons par exemple le système de transitions avec un seul état, s, et une seule transition d'attente $s \xrightarrow{\epsilon(0)} s$. Alors pour chaque valuation w des horloges, $(s, w) \models x \ \underline{\mathrm{in}}\ \mathbb{W}_{\{a\}}(x = 0)$. Par ailleurs, s n'autorise pas de transition étiquetée par a, et donc $(s, w) \not\models \langle a \rangle \mathtt{tt}$. Il est facile de vérifier que l'état s ne peut pas correspondre à un état d'un automate temporisé. Ainsi, le codage de la propriété $\langle a \rangle \mathtt{tt}$ dans $\mathcal{L}^{-}_{\forall S}$ proposée juste avant n'est plus valable ici. Nous allons maintenant voir qu'il n'y a aucune formule de $\mathcal{L}^{-}_{\forall S}$ équivalente à $\langle a \rangle \mathtt{tt}$ vis-à-vis de STT et donc que le langage $\mathcal{L}_{\forall S}$ est strictement plus expressif que $\mathcal{L}^{-}_{\forall S}$ vis-à-vis de STT. Dans ce but, nous exhibons deux états s et t d'un système de transitions temporisé avec la propriété suivante :
- pour chaque valuation w des horloges, (t, w) satisfait $\langle a \rangle \mathtt{tt}$, mais (s, w) ne la satisfait pas,
- pour chaque valuation w des horloges, l'ensemble des formules de $\mathcal{L}^{-}_{\forall S}$ qui sont satisfaites par (t, w) est inclus dans celui des formules satisfaites par (s, w).

Ces deux propriétés assurent qu'il n'y a pas de formule dans $\mathcal{L}^{-}_{\forall S}$ équivalente à $\langle a \rangle \mathtt{tt}$ vis-à-vis de STT.

Considérons les états s et t d'un système de transitions temporisé défini par :

$$s \xrightarrow{\epsilon(0)} s$$
$$t \xrightarrow{\epsilon(0)} t$$
$$t \xrightarrow{a} t$$

Évidemment, indépendamment de la valuation w pour les horloges, les couples (t, w) satisfont $\langle a \rangle \mathtt{tt}$, mais, comme nous l'avons remarqué ci-dessus, (s, w) ne satisfait pas cette formule. D'un autre côté, chaque formule de $\mathcal{L}^{-}_{\forall S}$ qui est satisfaite par (t, w) l'est aussi par (s, w).

Lemme 9.25 *Pour chaque formule $\varphi \in \mathcal{L}^{-}_{\forall S}$ et pour chaque valuation w des horloges, si $(t, w) \models \varphi$, alors $(s, w) \models \varphi$.*

PREUVE : Il est suffisant de montrer que la relation

$$\mathcal{R} \overset{\mathrm{def}}{=} \{((s, w), \varphi) \mid \varphi \in \mathcal{L}^{-}_{\forall S} \text{ et } (t, w) \models \varphi\}$$

est une relation de satisfaisabilité. Nous ne détaillons pas ce résultat et nous présentons uniquement le cas $\varphi \equiv \forall\!\!\forall_S \phi$.

Supposons que $((s,w), \forall\!\!\forall_S \phi) \in \mathcal{R}$, et que $s \stackrel{\epsilon(d)}{\Longrightarrow}_S s'$ pour un réel positif d et un état s'. Nous souhaitons montrer que $((s', w+d), \phi) \in \mathcal{R}$. À cette fin, remarquons tout d'abord que $d = 0$ et que $s' = s$. De plus, t autorise la transition $t \xrightarrow{\epsilon(0)}_S t$ que a soit dans S ou pas. Comme $(t,w) \models \forall\!\!\forall_S \phi$, nous montrons que $(t,w) \models \phi$. Finalement, par la définition de $\mathcal{R}$, il vient que $((s,w),\phi) \in \mathcal{R}$, ce que nous voulions montrer. □

À la lumière du lemme 9.25 et de toutes les considérations précédentes, nous obtenons finalement que $\mathcal{L}_{\forall S}$ est strictement plus expressif que $\mathcal{L}^-_{\forall S}$ vis-à-vis de STT, et ceci complète la preuve du théorème 9.23.

Nous allons maintenant comparer l'expressivité des langages SBLL et $\mathcal{L}^-_{\forall S}$. Nous allons montrer que SBLL est strictement moins expressif que $\mathcal{L}^-_{\forall S}$ vis-à-vis de STT, et *a fortiori* vis-à-vis de $\mathsf{STT}_{\mathrm{aut}}$.

La différence entre le langage présenté dans la définition 9.17 et celui présenté dans la définition 9.14 réside dans le fait que, contrairement à SBLL, $\mathcal{L}_{\forall S}$ a un opérateur de délai universel paramétré par un ensemble d'actions urgentes S. En effet, le langage SBLL est juste le sous-langage de $\mathcal{L}_{\forall S}$ dans lequel les seules occurrences de $\forall\!\!\forall_S$ autorisées sont celles pour lesquelles $S = \varnothing$. Nous allons maintenant montrer que l'utilisation de $\forall\!\!\forall_S$ pour des ensembles arbitraires d'actions urgentes S accroît l'expressivité. C'est ce qu'exprime le résultat suivant :

Théorème 9.26 *Le langage $\mathcal{L}^-_{\forall S}$ est strictement plus expressif que* SBLL *vis-à-vis de* $\mathsf{STT}_{\mathit{aut}}$.

Pour établir ce résultat, nous montrons que

1. chaque formule de SBLL est équivalente à une formule de $\mathcal{L}^-_{\forall S}$ (vis-à-vis de $\mathsf{STT}_{\mathrm{aut}}$) et que
2. il y a une formule dans $\mathcal{L}^-_{\forall S}$ qui n'est équivalente à aucune formule de SBLL vis-à-vis de $\mathsf{STT}_{\mathrm{aut}}$.

La première des conditions peut être obtenue directement car il y a une inclusion syntaxique de SBLL dans $\mathcal{L}_{\forall S}$ qui envoie chaque occurrence de $\forall\!\!\forall$ sur $\forall\!\!\forall_\varnothing$, et chaque occurrence de $\langle a \rangle \mathtt{tt}$ sur la formule $x \ \underline{\text{in}}\ \forall\!\!\forall_{\{a\}}(x = 0)$. À la lumière du lemme 9.24, tout ceci préserve clairement la sémantique des formules car les transitions $\stackrel{\epsilon(d)}{\Longrightarrow}$ et $\stackrel{\epsilon(d)}{\Longrightarrow}_\varnothing$ coïncident.

Pour établir la deuxième condition, il est suffisant d'exhiber deux automates temporisés $\mathcal{A}$ et $\mathcal{B}$, ainsi qu'une formule ϕ de $\mathcal{L}^-_{\forall S}$ tels que

- chaque formule de SBLL qui est satisfaite par $\mathcal{A}$ est aussi satisfaite par $\mathcal{B}$, et
- $\mathcal{A}$ satisfait ϕ, mais $\mathcal{B}$ ne satisfait pas ϕ.

Soient $\mathcal{A}$ et $\mathcal{B}$ des automates temporisés dont les états initiaux sont $(t_0, [x \leftarrow 0])$ et $(s_0, [x \leftarrow 0])$ de la figure 9.10, respectivement. Prenons $\phi \equiv \forall\!\!\forall_{\{a\}}[b]\mathtt{ff}$. Nous avons déjà vu que $\mathcal{A}$ satisfait ϕ, mais que $\mathcal{B}$ ne satisfait pas ϕ (voir l'exemple 9.22 page 194), et donc que le comportement de ces deux automates peut être différencié en utilisant une propriété de $\mathcal{L}^-_{\forall S}$. Nous allons maintenant montrer que chaque formule de SBLL qui est satisfaite par $\mathcal{A}$ est aussi satisfaite par $\mathcal{B}$. C'est ce qu'exprime le résultat suivant :

Lemme 9.27 *Soient t_0 et s_0 comme à la figure 9.10. Alors, pour toute formule $\varphi \in$ SBLL et pour toute valuation v pour les horloges de K,*

$$((t_0, [x \to 0]), v) \models \varphi \textit{ implique } ((s_0, [x \to 0]), v) \models \varphi.$$

PREUVE : Nous allons montrer que la relation $\mathcal{R}$ définie par :

$$\mathcal{R} \stackrel{\text{def}}{=} \{(((s_0,[x \to d]),v),\varphi) \mid ((t_0,[x \to d]),v) \models \varphi,\ \varphi \in \text{SBLL et } d < 1\} \cup \models$$

est une relation de satisfaisabilité. En effet, la seule chose intéressante qu'il faut tester est que les implications définissant $\models$ sont vérifiées par les paires $(((s_0,[x \to d]),v),\varphi)$ lorsque $d < 1$ et $((t_0,[x \to d]),v) \models \varphi$. Nous le faisons par une analyse de cas sur la forme de φ et nous allons uniquement présenter les détails de la preuve pour le cas où $\varphi \equiv \forall\!\!\!\forall\phi$. Pour les autres cas, nous remarquons juste que :

– le cas $\varphi \equiv [c]\phi$ est trivial car, comme $d < 1$, il n'y a pas de transition sortant de l'état $(s_0,[x \to d])$,
– le cas $\varphi \equiv \langle c\rangle\mathtt{tt}$ est évident aussi car $(t_0,[x \to d]) \not\models \langle c\rangle\mathtt{tt}$,

Comme annoncé, nous présentons maintenant les détails du cas que nous avons sélectionné.

Cas $\varphi \equiv \forall\!\!\!\forall\phi$**.** Supposons que $(s_0,[x \to d]) \stackrel{\epsilon(d')}{\Longrightarrow} (s,[x \to d''])$ pour $d',d'' \in \mathbb{R}^+$ et un état s. Nous montrons que $(((s,[x \to d'']),v+d'),\phi) \in \mathcal{R}$ en distinguant trois cas dépendant du fait que $d+d' < 1$, ou bien $d+d' = 1$, ou bien $d+d' > 1$.

➜ **Cas** $d+d' < 1$. Dans ce cas, nous montrons que $s_0 = s$ et $d'' = d+d'$. Comme $(t_0,[x \to d]) \xrightarrow{\epsilon(d')} (t_0,[x \to d+d'])$ et $((t_0,[x \to d]),v) \models \forall\!\!\!\forall\phi$, il s'ensuit que

$$((t_0,[x \to d+d']),v+d') \models \phi.$$

Utilisant la définition de la relation $\mathcal{R}$, nous obtenons maintenant immédiatement ce que nous voulions, à savoir que $s_0 = s$ et $d'' = d+d'$.

➜ **Cas** $d+d' = 1$**.** Dans ce cas, nous montrons que $s_0 = s$ et $d'' = d+d'$. Comme $d < 1$ et $d+d' = 1$, nous obtenons que

$$(t_0,[x \to d]) \xrightarrow{\tau} (t_1,[x \to d]) \xrightarrow{\epsilon(d')} (t_1,[x \to d+d']) \xrightarrow{\tau} (s_0,[x \to d+d']).$$

Comme $((t_0,[x \to d]),v) \models \forall\!\!\!\forall\phi$, il vient que $((s_0,[x \to d+d']),v+d') \models \phi$, et nous avons fini car $\models$ est incluse dans $\mathcal{R}$.

➜ **Cas** $d+d' > 1$**.** Dans ce cas, $(s,[x \to d''])$ peut prendre l'une des deux formes suivantes :

1. $(s,[x \to d'']) = (s_0,[x \to d+d'])$,
2. $(s,[x \to d'']) = (s_1,[x \to d+d'])$.

Dans le premier cas, en raisonnant comme ci-dessus, nous pouvons montrer que $(((s_0,[x \to d+d']),v+d'),\phi)$ est contenu dans $\mathcal{R}$. Dans le second cas, comme $d < 1$ et $d+d' > 1$, nous pouvons montrer que

$$(t_0,[x \to d]) \stackrel{\epsilon(d')}{\Longrightarrow} (s_1,[x \to d+d']).$$

Comme $((t_0,[x \to d]),v) \models \forall\!\!\!\forall\phi$, il s'ensuit que $((s_1,[x \to d+d']),v+d')$ satisfait ϕ. L'inclusion de $\models$ dans $\mathcal{R}$ termine la preuve.

Les cas restants de la preuve sont identiques et ne sont pas détaillés. $\square$

L'analyse ci-dessus montre que la propriété $\forall\!\!\!\forall_{\{a\}}[b]\mathtt{ff}$ ne peut pas être exprimée dans SBLL (lorsque nous nous restreignons aux $\mathsf{STT}_{\text{aut}}$). Supposons que, en fait, il y ait une formule ϕ dans SBLL équivalente à $\forall\!\!\!\forall_{\{a\}}[b]\mathtt{ff}$ vis-à-vis de $\mathsf{STT}_{\text{aut}}$. Alors $\mathcal{A}$ satisfait ϕ, mais $\mathcal{B}$ ne satisfait pas ϕ (exemple 9.22). Cependant, cela contredit le lemme 9.27. Ainsi, aucune formule de SBLL n'est équivalente à $\forall\!\!\!\forall_{\{a\}}[b]\mathtt{ff}$ vis-à-vis de $\mathsf{STT}_{\text{aut}}$. Ceci termine la preuve du théorème 9.26.

9.3.7 Tester $\mathcal{L}^{-}_{\forall S}$

Dans la partie 9.3.2, nous avons vu comment nous pouvions tester le langage SBLL. Nous souhaitons maintenant étendre notre résultat au langage $\mathcal{L}^{-}_{\forall S}$.

Théorème 9.28 *Pour chaque formule close φ de $\mathcal{L}^{-}_{\forall S}$, il existe un automate de test T_φ, sur l'ensemble d'horloges $\{k\} \cup \mathsf{clocks}(\varphi)$, où k n'apparaît pas dans $\mathsf{clocks}(\varphi)$, qui teste φ.*

PREUVE : Il nous suffit de compléter la preuve du théorème 9.15 en considérant le nouvel opérateur $\forall\!\!\!\forall_S \varphi$. La preuve pour SBLL était basée sur une induction structurelle sur les formules (qui ne sont pas nécessairement closes). Nous supposons donc que nous pouvons construire un automate de test pour la formule φ et nous voulons construire un automate de test pour la formule $\forall\!\!\!\forall_S \varphi$. Considérons l'automate de test $T_{\forall_S \varphi}$ dessiné sur la figure 9.11.

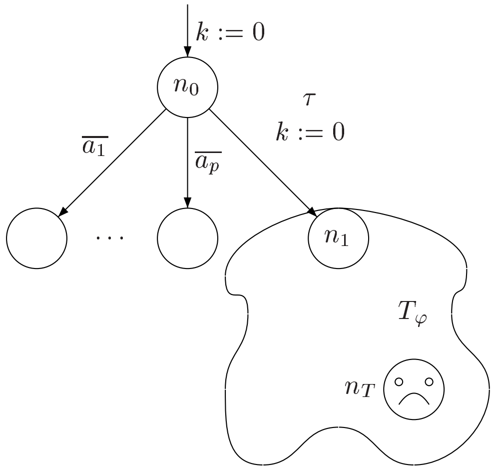


Figure 9.11: $T_{\forall_S \varphi}$ si $S = \{a_1, \dots, a_p\}$

Supposons qu'un état rejetant de $T_{\forall_S \varphi}$ soit accessible dans la composition parallèle d'un système de transitions $\mathcal{T}$ avec $T_{\forall_S \varphi}$. Alors cela signifie que l'exécution suivante est permise dans $(\mathcal{T} \parallel T_{\forall_S \varphi})$ ($d \geq 0$ est un réel quelconque) :

$$\begin{array}{rcl} (s_0 \parallel (n_0, [k \leftarrow 0]w_0)) & \stackrel{\epsilon(d)}{\Longrightarrow} & (s_1 \parallel (n_0, ([k \leftarrow 0]w_0) + d)) \\ & \stackrel{\tau}{\longrightarrow} & (s_1 \parallel (n_1, [k \leftarrow 0](w_0 + d))) \\ & \rightsquigarrow^* & (s_2 \parallel (n_T, w)) \end{array}$$

En outre, la première transition $\stackrel{\epsilon(d)}{\Longrightarrow}$ est en fait une transition $\stackrel{\epsilon(d)}{\Longrightarrow}_S$ car il n'y a pas de synchronisation autorisée entre une action a_i et une action $\overline{a_i}$. Il s'ensuit que $(s_1, ([k \leftarrow 0]w_0) + d) \not\models \varphi$, et donc que $(s_0, [k \leftarrow 0]w_0) \not\models \forall\!\!\!\forall_S \varphi$.
Réciproquement, supposons que $(s_0, w_0) \not\models \forall\!\!\!\forall_S \varphi$. Cela signifie qu'il existe un réel $d \geq 0$ et un état s_1 de $\mathcal{T}$ tels que $s_0 \stackrel{\epsilon(d)}{\Longrightarrow}_S s_1$ et $(s_1, w_0 + d) \not\models \varphi$. Ainsi, dans la composition parallèle de $\mathcal{T}$ et $T_{\forall_S \varphi}$, l'exécution suivante est autorisée :

$$\begin{array}{rcl} (s_0 \parallel (n_0, [k \leftarrow 0]w_0)) & \stackrel{\epsilon(d)}{\Longrightarrow}_S & (s_1 \parallel (n_0, ([k \leftarrow 0]w_0) + d)) \\ & \stackrel{\tau}{\longrightarrow} & (s_1 \parallel (n_1, [k \leftarrow 0](w_0 + d))) \\ & \rightsquigarrow^* & (s \parallel (n_T, w_T)) \end{array}$$

Ainsi, un état rejetant de $T_{\forall_S \varphi}$ est accessible dans $(\mathcal{T} \parallel T_{\forall_S \varphi})$. □

Comme conséquence du théorème ci-dessus et du lemme 9.24, nous obtenons le résultat plus fort pour les automates temporisés.

Corollaire 9.29 *Soit φ une formule close de $\mathcal{L}_{\forall S}$. Alors il existe un automate de test T_φ, sur l'ensemble d'horloges $\{k\} \cup \mathsf{clocks}(\varphi)$, qui teste φ pour les automates temporisés.*

Comme nous l'avons remarqué dans la partie 9.3, le langage $\mathcal{L}^-_{\forall S}$ permet uniquement une utilisation restreinte de l'opérateur « ou ». De plus, même si les formules de la forme $\langle a\rangle\mathtt{tt}$ peuvent être codées dans $\mathcal{L}^-_{\forall S}$ si a est une action urgente (lemme 9.24), il n'est pas possible d'exprimer $\langle a\rangle\mathtt{tt}$ dans $\mathcal{L}^-_{\forall S}$ si a n'est pas urgente. Ces restrictions sont justifiées par les résultats négatifs suivants, où nous considérons des extensions du langage $\mathcal{L}^-_{\forall S}$ avec ce que nous venons de mentionner.

Proposition 9.30

1. *La formule $\langle a\rangle\mathtt{tt}$ n'est pas testable, que a soit urgente ou pas.*
2. *La formule $\langle a\rangle\mathtt{tt}$ (a non urgente) n'est pas testable pour les automates temporisés.*
3. *La formule $[a]\mathtt{ff} \vee [b]\mathtt{ff}$ (a, b non urgentes) n'est pas testable pour les automates temporisés.*

PREUVE : Nous montrons les trois propriétés séparément.

1. Supposons, dans le but de trouver une contradiction, que l'automate de test T teste la formule $\langle a\rangle\mathtt{tt}$. Considérons le système de transitions temporisé $\mathcal{T}$ constitué d'un seul état s et de la transition $s \xrightarrow{\epsilon(0)} s$. Soit u une valuation arbitraire pour les horloges des formules. Comme (s, u) ne satisfait pas $\langle a\rangle\mathtt{tt}$, le couple (s, u) ne doit pas passer le test T — c'est-à-dire qu'un état rejetant de T est accessible de l'état $(s \parallel (n_0, [X_0 \leftarrow 0](u \restriction X)))$ dans $(\mathcal{T} \parallel \mathcal{T}_T)$, où n_0 est l'état initial de T. Comme s ne permet que de faire une transition d'attente de 0, il est facile de se rendre compte que cela signifie qu'il y a une suite de τ-transitions allant de $(n_0, [X_0 \leftarrow 0](u \restriction X))$ jusqu'à un état rejetant de T. Considérons maintenant le système de transitions temporisé ayant un unique état t et dont les transitions sont

   $$t \xrightarrow{\epsilon(0)} t \quad \text{et} \quad t \xrightarrow{a} t.$$

   De manière évidente, $(t, u) \models \langle a\rangle\mathtt{tt}$. Cependant, (t, u) ne passe pas le test T car T peut, de manière indépendante, aller jusqu'à un état rejetant en faisant une suite de τ-transitions (comme pour le système de transitions précédent). Ceci contredit notre hypothèse que T teste la formule $\langle a\rangle\mathtt{tt}$.

2. (SKETCH.) Supposons, dans le but de trouver une contradiction, qu'un automate de test T teste la formule $\langle a\rangle\mathtt{tt}$ (a n'est pas une action urgente) pour les automates temporisés. Alors l'automate temporisé $\mathcal{A}_{a+\tau}$[6] dessiné sur la figure 9.12(a) ne doit pas passer le test T. En utilisant l'hypothèse que a n'est pas urgente, nous pouvons maintenant montrer, en analysant une exécution arbitraire menant à un état rejetant de T dans $(\mathcal{A}_{a+\tau} \parallel T)$, qu'un état rejetant de T peut aussi être atteint dans $(\mathcal{A}_a \parallel T)$, où $\mathcal{A}_a$ est l'automate dessiné sur la figure 9.12(b)). Comme $\mathcal{A}_a$ satisfait bien évidemment la formule $\langle a\rangle\mathtt{tt}$, ceci contredit l'hypothèse que T teste la formule $\langle a\rangle\mathtt{tt}$.

3. (SKETCH.) Supposons, dans le but de trouver une contradiction, qu'un automate de test T teste la formule $[a]\mathtt{ff} \vee [b]\mathtt{ff}$. L'automate temporisé $\mathcal{A}_{a+b}$ dessiné sur la figure 9.12(c) ne doit pas pouvoir passer le test T. En utilisant l'hypothèse que T teste la formule $[a]\mathtt{ff} \vee [b]\mathtt{ff}$, par une analyse d'une exécution arbitraire menant à un état rejetant de T dans $(\mathcal{A}_{a+b} \parallel T)$, nous obtenons que soit

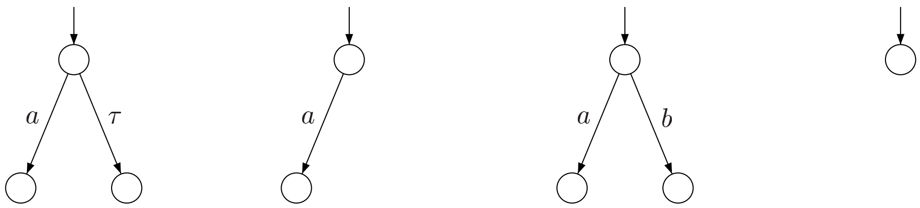
[6] La notation $a + \tau$ vient du fait que cet automate représente le processus TCCS $a + b$ [Yi90, Yi91].

(A) une telle exécution contient uniquement des transitions d'attente et des τ-transitions de l'automate T, soit

(B) elle doit contenir une synchronisation avec une action a ou une action b, potentiellement précédée ou suivie par des τ-transitions et des transitions d'attente de l'automate T.

Dans le cas (A), l'automate temporisé $\mathcal{A}_{\mathsf{nil}}$ dessiné sur la figure 9.12(d) ne passera pas non plus le test T. Comme un tel automate temporisé satisfait bien évidemment la propriété $[a]\mathtt{ff} \vee [b]\mathtt{ff}$, ceci contredit notre hypothèse que T teste la formule $[a]\mathtt{ff} \vee [b]\mathtt{ff}$.

Dans le cas (B), soit l'automate temporisé associé avec le processus a, soit celui associé avec le processus b ne passera pas le test T. De nouveau, comme ces deux automates temporisés satisfont de manière évidente la formule $[a]\mathtt{ff} \vee [b]\mathtt{ff}$, ceci contredit notre hypothèse que T teste la formule $[a]\mathtt{ff} \vee [b]\mathtt{ff}$. □



(a) $\mathcal{A}_{a+\tau}$ (b) $\mathcal{A}_a$ (c) $\mathcal{A}_{a+b}$ (d) $\mathcal{A}_{\mathsf{nil}}$

Figure 9.12: Les automates que nous testons

Le premier point de la proposition ci-dessus justifie notre remarque précédente.

Dans la partie qui suit, nous étudierons le problème de la complétude de $\mathcal{L}^-_{\forall S}$ vis-à-vis de l'accessibilité. En particulier, nous montrerons notre principal résultat, c'est-à-dire que les propriétés qui peuvent s'exprimer dans $\mathcal{L}^-_{\forall S}$ sont précisément celles qui peuvent être testées en utilisant les automates de test.

9.4 Compositionnalité et complétude

Nous avons montré que chaque propriété φ qui peut s'exprimer dans le langage $\mathcal{L}_{\forall S}$ (et, *a fortiori*, dans SBLL) est testable pour les automates temporisés, et que $\mathcal{L}^-_{\forall S}$ est testable (pour les systèmes de transitions temporisés quelconques). Nous nous intéressons maintenant au problème de la complétude de ces langages. Plus précisément, nous allons étudier si toutes les propriétés qui sont testables pour les systèmes de transitions temporisés peuvent être exprimées dans les langages SBLL et $\mathcal{L}^-_{\forall S}$ — dans le sens où, pour chaque automate de test T, il existe une formule ψ_T telle que chaque état d'un système de transitions temporisé passe le test T si et seulement s'il satisfait ψ_T.

Nous avons déjà suffisamment d'informations pour affirmer que le langage SBLL est strictement moins expressif que le formalisme des automates de test. En effet, l'automate dessiné à la figure 9.7 n'est rien d'autre que l'automate de test pour la formule $\mathbb{W}_{\{a\}}[b]\mathtt{ff}$, ce qui ne peut pas être exprimé dans SBLL (voir la preuve du théorème 9.26).

Notre but, dans cette partie, est de montrer que, contrairement à SBLL, le langage $\mathcal{L}^-_{\forall S}$ est complet. Dans la preuve de ce résultat de complétude, nous allons suivre une approche indirecte en nous focalisant sur la compositionnalité du langage $\mathbb{L}$. Comme nous le verrons dans la proposition 9.33, si le langage $\mathbb{L}$ est compositionnel (voir la définition 9.32), alors il sera complet (voir la définition 9.31). Nous montrerons ensuite que SBLL n'est pas compositionnelle, mais que $\mathcal{L}^-_{\forall S}$ l'est.

Nous commençons par quelques définitions préliminaires, introduisant les concepts-clé de compositionnalité et de complétude.

Définition 9.31 (Complétude) *Soit $\mathbb{L}$ un langage sur l'ensemble d'horloges K. Nous disons que $\mathbb{L}$ est* complet *(par rapport aux automates de test) si pour tout automate de test T, il existe une formule $\varphi_T \in \mathbb{L}$ telle que,*

« pour tout état s d'un système de transitions temporisé, pour toute valuation u,
$(s,u) \models \varphi_T$ si et seulement si (s,u) passe le test T »

L'idée intuitive de la compositionnalité a été donnée dans la partie 7.2.4. Dans notre cadre, elle peut être définie formellement de la façon suivante :

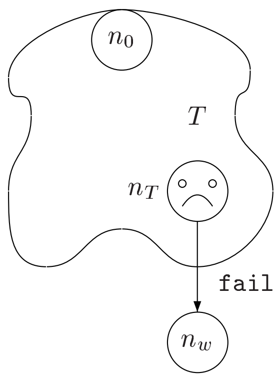
Définition 9.32 (Compositionnalité) *Soit $\mathbb{L}$ un langage sur un ensemble d'horloges K, nous disons que $\mathbb{L}$ est* compositionnel *(par rapport aux automates de test et de $\|$) si, pour tout $\varphi \in \mathbb{L}$ et pour tout automate de test $T = (N, Z, X_0, \mathsf{Act}_\tau, n_0, N_T, E)$ (avec X disjoint de $\mathsf{clocks}(\varphi)$), il existe une formule $\varphi/T \in \mathbb{L}$ sur l'ensemble d'horloges $\mathsf{clocks}(\varphi) \cup X$ telle que, pour chaque état s d'un système de transitions temporisé $\mathcal{T}$ et pour chaque valuation u de K,*

$$\Big(\Big(s \parallel (n_0, [X_0 \leftarrow 0](u \upharpoonright X))\Big), u\Big) \models \varphi \iff (s, [X_0 \leftarrow 0]u) \models \varphi/T^7$$

Notre intérêt dans la compositionnalité provient du résultat suivant, qui fait le lien entre les notions de compositionnalité et de complétude. Dans ce qui suit, nous utilisons $\mathbb{L}_{\text{bad}}$ pour représenter le langage, uniquement constitué de la formule $\mathbb{V}_\varnothing[\mathtt{fail}]\mathtt{ff}$, où `fail` est une action non contenue dans Act_τ.

Proposition 9.33 *Soit $\mathbb{L}$ un langage (sur un ensemble d'horloges K) qui contient $\mathbb{L}_{bad}$. Supposons que $\mathbb{L}$ soit compositionnel. Alors $\mathbb{L}$ est complet.*

Preuve : Supposons que $\mathbb{V}_\varnothing[\mathtt{fail}]\mathtt{ff} \in \mathbb{L}$ et que $\mathbb{L}$ soit compositionnel. Considérons alors un automate de test $T = (N, X, X_0, \mathsf{Act}_\tau, n_0, N_T, E)$. Nous voulons montrer qu'il existe une formule $\varphi_T \in \mathbb{L}$ qui vérifie les propriétés décrites dans la définition 9.31. Dans ce but, nous commençons par ajouter à T un nouvel état n_w et des transitions $e = n_T \xrightarrow{\mathtt{tt},\mathtt{fail},\varnothing} n_w$, où n_T est un état rejetant de T, et `fail` est une action qui n'est pas dans Act_τ, comme décrit ci-dessous :



[7] u est une valuation d'un ensemble d'horloges contenant celles de l'automate de test T et adéquat pour la logique $\mathbb{L}$. Ainsi, $(n_0, [X_0 \leftarrow 0](u \upharpoonright X))$ est un état initial de T donc $\Big(s \parallel (n_0, [X_0 \leftarrow 0](u \upharpoonright X))\Big)$ est un état de $(\mathcal{T} \parallel \mathcal{T}_T)$.

Appelons l'automate de test résultant $T_{\texttt{fail}}$. Clairement, tout état s d'un système de transitions temporisé passe le test T si et seulement si il passe le test $T_{\texttt{fail}}$. Comme $\mathbb{L}$ est compositionnel, nous pouvons définir la formule φ_T donnée par la définition 9.32, *i.e.* $X_0 \ \underline{\text{in}} \ ((\forall\!\!\!\forall_{\varnothing}[\texttt{fail}]\mathtt{ff})/T_{\texttt{fail}})$. Nous allons maintenant montrer qu'une telle formule φ_T vérifie en fait les propriétés décrites dans la définition 9.31.

Soient $\mathcal{T}$ un système de transitions temporisé, s un état de $\mathcal{T}$ et v une valuation d'au moins les horloges de T. Nous raisonnons de la manière suivante :

(s, v) passe le test T ssi $(s \parallel (n_0, [X_0 \leftarrow 0](v \restriction X)))$ ne permet pas d'atteindre un état rejetant de T dans $(\mathcal{T} \parallel \mathcal{T}_T)$
(Par définition de « passer un test »)

ssi $(s \parallel (n_0, [X_0 \leftarrow 0](v \restriction X)))$ ne permet pas d'atteindre un état rejetant de $T_{\texttt{fail}}$ dans $(\mathcal{T} \parallel \mathcal{T}_{T_{\texttt{fail}}})\backslash\mathsf{Act}$
(Comme `fail` n'est pas dans Act et par construction de $T_{\texttt{fail}}$)

ssi $(s, [X_0 \leftarrow 0]v) \models (\forall\!\!\!\forall_{\varnothing}\texttt{fail}]\mathtt{ff})/T_{\texttt{fail}}$
(Comme $\mathbb{L}$ est compositionnel et $\mathsf{clocks}((\forall\!\!\!\forall_{\varnothing}[\texttt{fail}]\mathtt{ff})/T_{\texttt{fail}}) \subseteq X$)

ssi $(s, v) \models X_0 \ \underline{\text{in}} \ ((\forall\!\!\!\forall_{\varnothing}[\texttt{fail}]\mathtt{ff})/T_{\texttt{fail}})$
(Par définition de la relation $\models$)

ssi $(s, v) \models \varphi_T$.

Comme T était un automate de test arbitraire, nous pouvons en conclure que le langage de propriétés $\mathbb{L}$ est complet. □

Comme SBLL et $\mathcal{L}^-_{\forall S}$ sont des extensions de $\mathbb{L}_{\text{bad}}$, à la lumière de la proposition ci-dessus, une façon de montrer qu'ils sont complets est de vérifier s'ils sont compositionnels. Malheureusement, le premier résultat est négatif.

Proposition 9.34 *Le langage* SBLL *n'est pas compositionnel.*

PREUVE : Supposons, dans le but d'obtenir une contradiction, que SBLL soit compositionnel. Comme SBLL contient $\mathbb{L}_{\text{bad}}$, la proposition 9.33 implique que SBLL est complet. Ainsi, en particulier, il existe une formule $\varphi \in$ SBLL telle que pour chaque système de transitions temporisés $\mathcal{T}$, pour chaque état s de $\mathcal{T}$ et pour chaque valuation u de K,

$$(s, u) \text{ passe le test } T_\phi \text{ si et seulement si } (s, u) \models \varphi$$

où T_ϕ est l'automate de test pour la formule $\phi \equiv \forall\!\!\!\forall_{\{a\}}[b]\mathtt{ff}$ dessiné sur la figure 9.7. Soient t_0 et s_0 comme sur la figure 9.10. À la lumière du théorème 9.28 et de l'exemple 9.22, pour chaque valuation w des horloges de K, $((t_0, [x \to 0]), w)$ satisfait φ, mais pas $((s_0, [x \to 0]), w)$. Cependant, comme φ est contenue dans SBLL, ceci contredit le lemme 9.27. □

D'un autre côté, comme nous le montrerons bientôt, $\mathcal{L}^-_{\forall S}$ est compositionnel, et est donc complet. Nous commençons par définir le quotient des formules de $\mathcal{L}_{\forall S}$, dans l'esprit des travaux présentés dans [And95, LL95].

Notations. Si $\mathcal{A}$ est un automate temporisé, pour chaque état n et pour chaque action μ dans $\mathcal{A}$, nous notons $E(n, \mu)$ l'ensemble de toutes les transitions partant de q étiquetées par l'action μ. Pour chaque état n, nous définissons $\mathsf{Act}(n)$ l'ensemble de toutes les actions de Act étiquetant une transition partant de n, *i.e.* $\mathsf{Act}(n) = \{\mu \in \mathsf{Act} \mid E(n, \mu) \neq \varnothing\}$ et nous utilisons $\Sigma_u(n)$ l'ensemble de toutes les actions urgentes de $\mathsf{Act}(n)$. Si e est une transition, nous la noterons souvent $n_e \xrightarrow{g_e, \mu_e, r_e} n'_e$.

Définition 9.35 (Quotient pour $\mathcal{L}_{\forall S}$) *Soient $\mathcal{A}$ un automate temporisé, et n l'un de ses états. Soit φ une formule de $\mathcal{L}_{\forall S}^{-}$. Nous définissons la formule φ/n (dire « φ quotientée par n ») selon le tableau 9.13.*

Remarquons que, reprenant les notations de la définition 9.32, si T est un automate φ/T correspond au quotient de φ par l'état initial de T.

La définition de la formule quotient φ/n présentée ci-dessus correspond en fait à une liste finie d'équations récursives, sur les variables ϕ/m pour chaque formule ϕ et chaque état m d'un automate temporisé. La formule quotient φ/n elle-même est la composante associée à φ/n de la solution la plus grande du système d'équations ayant φ/n comme membre de gauche. Par exemple, si φ est la formule $[a]\mathtt{ff}$ et si n est un état d'un automate temporisé qui a pour unique transition $n \xrightarrow{\mathtt{t},\bar{b},\varnothing} n$, alors comme il est facile de vérifier, φ/n est la solution la plus grande de l'équation récursive :

$$\varphi/n \quad \stackrel{\text{def}}{=} \quad [a]\mathtt{ff} \wedge [b](\varphi/n)$$

ce qui correspond à la formule $\mathsf{max}(Z, [a]\mathtt{ff} \wedge [b]Z)$ dans le langage de propriétés $\mathcal{L}_{\forall S}^{-}$. Cette formule exprime le fait assez intuitif que l'état $(s \parallel n)$ ne peut pas faire une transition $\stackrel{a}{\Longrightarrow}$ si et seulement si s ne peut pas non plus faire de telle transition après une séquence quelconque de synchronisations sur l'action b avec n.

Remarquons que, comme nous venons de le constater, le quotient d'une formule sans opérateur de récursion, peut tout de même être une formule comportant de la récursion. Nous pouvons montrer que ceci est inévitable, car le fragment sans récursion de $\mathcal{L}_{\forall S}^{-}$ n'est pas compositionnel.

Nous pouvons maintenant établir le résultat-clé suivant.

Théorème 9.36 *Soit φ une formule close de $\mathcal{L}_{\forall S}^{-}$. Supposons que s soit un état d'un système de transitions et que n' soit un état d'un automate temporisé sur l'ensemble d'horloges X. Supposons en outre que u et v' soient des valuations pour les ensembles disjoints d'horloges* $\mathsf{clocks}(\varphi)$ *et X respectivement. Alors*

$$(s \parallel (n', v'), u) \models \varphi \iff (s, v' : u) \models \varphi/n'.$$

Tout comme les formules ϕ/n étaient construites par induction sur la structure de ϕ (tableau 9.13), la preuve des deux implications du théorème 9.36 va être faite par induction sur ϕ. Celle-ci, longue et un peu fastidieuse, est proposée dans l'annexe B de ce chapitre, à la page 212.

Une application immédiate de ce théorème est la propriété suivante :

Corollaire 9.37 *Le langage de propriétés $\mathcal{L}_{\forall S}^{-}$ est compositionnel.*

Utilisant alors ce corollaire et la proposition 9.33, nous obtenons alors le théorème suivant, qui dit que le langage $\mathcal{L}_{\forall S}^{-}$ permet d'exprimer toutes les propriétés qui peuvent être testées *via* les automates de test.

Théorème 9.38 *Le langage de propriétés $\mathcal{L}_{\forall S}^{-}$ est complet.*

Théorème 9.39 *Le langage $\mathcal{L}_{\forall S}^{-}$ est l'extension compositionnelle de $\mathbb{L}_{bad}$ la moins expressive vis-à-vis de* STT.

$$
\begin{aligned}
\mathit{ff}/n' &\stackrel{\text{def}}{=} \mathit{ff} \\
(\phi_1 \wedge \phi_2)/n' &\stackrel{\text{def}}{=} \phi_1/n' \wedge \phi_2/n' \\
&\quad \wedge \bigwedge_{e \in E(n',\tau)} (g_e \Rightarrow r_e \ \underline{\mathsf{in}}\ (\phi_1 \wedge \phi_2)/n'_e) \\
&\quad \wedge \bigwedge_{b \in \mathsf{Act}} \ \bigwedge_{e \in E(n',b)} (g_e \Rightarrow [\overline{b}]\,(r_e \ \underline{\mathsf{in}}\ (\phi_1 \wedge \phi_2)/n'_e)) \\
(g \vee \phi)/n' &\stackrel{\text{def}}{=} (g \vee (\phi/n')) \\
&\quad \wedge \bigwedge_{e \in E(n',\tau)} (g_e \Rightarrow r_e \ \underline{\mathsf{in}}\ (g \vee \phi)/n'_e) \\
&\quad \wedge \bigwedge_{b \in \mathsf{Act}} \ \bigwedge_{e \in E(n',b)} (g_e \Rightarrow [\overline{b}]\,(r_e \ \underline{\mathsf{in}}\ (g \vee \phi)/n'_e)) \\
([a]\phi)/n' &\stackrel{\text{def}}{=} [a](\phi/n') \\
&\quad \wedge \bigwedge_{e \in E(n',a)} (g_e \Rightarrow r_e \ \underline{\mathsf{in}}\ \phi/n'_e) \\
&\quad \wedge \bigwedge_{e \in E(n',\tau)} (g_e \Rightarrow r_e \ \underline{\mathsf{in}}\ ([a]\phi)/n'_e) \\
&\quad \wedge \bigwedge_{b \in \mathsf{Act}} \ \bigwedge_{e \in E(n',b)} (g_e \Rightarrow [\overline{b}]\,(r_e \ \underline{\mathsf{in}}\ ([a]\phi)/n'_e)) \\
(x \ \underline{\mathsf{in}}\ \phi)/n' &\stackrel{\text{def}}{=} x \ \underline{\mathsf{in}}\ (\phi/n') \\
&\quad \wedge \bigwedge_{e \in E(n',\tau)} (g_e \Rightarrow r_e \ \underline{\mathsf{in}}\ (x \ \underline{\mathsf{in}}\ \phi)/n'_e) \\
&\quad \wedge \bigwedge_{b \in \mathsf{Act}} \ \bigwedge_{e \in E(n',b)} (g_e \Rightarrow [\overline{b}]\,(r_e \ \underline{\mathsf{in}}\ (x \ \underline{\mathsf{in}}\ \phi)/n'_e)) \\
(\forall\!\!\!\forall_S \phi)/n' &\stackrel{\text{def}}{=} \forall\!\!\!\forall_{S \cup \overline{\Sigma_u(n')}}\ (\phi/n' \\
&\quad \wedge \bigwedge_{e \in E(n',\tau)} (g_e \Rightarrow r_e \ \underline{\mathsf{in}}\ (\forall\!\!\!\forall_S \phi)/n'_e) \\
&\quad \wedge \bigwedge_{b \in \mathsf{Act}} \ \bigwedge_{e \in E(n',b)} (g_e \Rightarrow [\overline{b}]\,(r_e \ \underline{\mathsf{in}}\ (\forall\!\!\!\forall_S \phi)/n'_e))) \qquad \text{si } S \cap \Sigma_u(n') = \varnothing \\
(\forall\!\!\!\forall_S \phi)/n' &\stackrel{\text{def}}{=} (\phi/n' \\
&\quad \wedge \bigwedge_{e \in E(n',\tau)} (g_e \Rightarrow r_e \ \underline{\mathsf{in}}\ (\forall\!\!\!\forall_S \phi)/n'_e) \\
&\quad \wedge \bigwedge_{b \in \mathsf{Act}} \ \bigwedge_{e \in E(n',b)} (g_e \Rightarrow [\overline{b}]\,(r_e \ \underline{\mathsf{in}}\ (\forall\!\!\!\forall_S \phi)/n'_e))) \qquad \text{si } S \cap \Sigma_u(n') \neq \varnothing \\
Z/n' &\stackrel{\text{def}}{=} Z \\
\mathsf{max}(Z,\phi)/n' &\stackrel{\text{def}}{=} (\phi\{\mathsf{max}(Z,\phi)/Z\})/n' \\
&\quad \wedge \bigwedge_{e \in E(n',\tau)} (g_e \Rightarrow r_e \ \underline{\mathsf{in}}\ \mathsf{max}(Z,\phi)/n'_e) \\
&\quad \wedge \bigwedge_{b \in \mathsf{Act}} \ \bigwedge_{e \in E(n',b)} (g_e \Rightarrow [\overline{b}]\,(r_e \ \underline{\mathsf{in}}\ (\mathsf{max}(Z,\phi))/n'_e))
\end{aligned}
$$

Tableau 9.13: Construction du quotient pour $\mathcal{L}^{-}_{\forall\!\!\!\forall S}$

PREUVE : Soit $\mathbb{L}$ un langage de propriétés compositionnel qui contient $\mathbb{L}_{\text{bad}}$. Nous montrons que chaque propriété de $\mathcal{L}^{-}_{\forall S}$ est équivalente à une formule de $\mathbb{L}$ vis-à-vis de STT, *i.e.*, que $\mathbb{L}$ est au moins aussi expressif que $\mathcal{L}^{-}_{\forall S}$. À cette fin, soit φ une propriété de $\mathcal{L}^{-}_{\forall S}$. Par le théorème 9.28, il existe un automate de test T_φ tel que pour chaque état s de $\mathcal{T}$ et pour chaque valuation u des horloges de K,

$$(s,u) \models \varphi \text{ si et seulement si } (s,u) \text{ passe le test } T_\varphi.$$

Comme $\mathbb{L}$ est une extension compositionnelle de $\mathbb{L}_{\text{bad}}$, la proposition 9.33 implique que $\mathbb{L}$ est complet. Ainsi, il existe une formule $\phi \in \mathbb{L}$ telle que pour chaque état s, pour chaque valuation u,

$$(s,u) \models \phi \text{ si et seulement si } (s,u) \text{ passe le test } T_\varphi.$$

Il s'ensuit que ϕ et φ sont satisfaites par les mêmes couples (s,u) où s est un état d'un système de transitions temporisé et u est une valuation et sont donc équivalentes vis-à-vis de STT. □

9.5 Conclusion

Comme Pierre Wolper l'a argumenté dans [Wol97], des algorithmes efficaces pour décider l'accessibilité peuvent être utilisés pour résoudre beaucoup de problèmes en vérification. Dans l'étude que nous avons menée dans ce chapitre, nous avons montré comment il est possible de réduire le model-checking de propriétés de sûreté et de vivacité bornée exprimables dans le langage $\mathcal{L}_{\forall S}$ à l'analyse de l'accessibilité d'états rejetants d'un automate de test. Cette approche nous permet de tirer entièrement profit de l'outil de vérification UPPAAL [BLL+95, LPY97, BLL+98, ABB+01] qui permet essentiellement de vérifier de manière efficace des propriétés d'accessibilité pour les automates temporisés.

L'approche du model-checking par les automates de test que nous venons de décrire a été testée sur un protocole CSMA/CD [ABL98]. Les études de cas réalisées avec UPPAAL et décrites dans [KP95, BGK+96, JLS96] utilisaient déjà des automates de test pour tester certaines des propriétés, ce qui montrent que l'approche des automates de test peut être appliquée à des études de cas réalistes.

Notons pour finir que la génération automatique d'automates de test à partir de formules de $\mathcal{L}_{\forall S}$ a été implémentée dans sc Uppaal (voir [ABL98]).

Appendice

Nous allons présenter les preuves des théorèmes 9.15 et 9.36, que nous n'avons pas développées dans le corps de ce chapitre pour en faciliter la lecture.

A. Preuve du théorème 9.15

Théorème 9.15 *Pour chaque formule close φ de SBLL^{-}, il existe un automate de test T_φ, sur l'ensemble d'horloges $\{k\} \cup \mathsf{clocks}(\varphi)$, où k est une horloge n'apparaissant pas dans $\mathsf{clocks}(\varphi)$, qui teste la formule φ.*

*Pour chaque formule close φ de SBLL, il existe un automate de test T_φ, sur l'ensemble d'horloges $\{k\} \cup \mathsf{clocks}(\varphi)$, où k est une horloge n'apparaissant pas dans $\mathsf{clocks}(\varphi)$, qui teste la formule φ **pour les automates temporisés**.*

Avant de démontrer ce théorème, nous généralisons notre notion de test à des formules non closes, car cela nous sera utile dans la preuve (qui va être faite par induction sur la structure des formules). Un *automate de test non clos* est un 9-uplet $T = (N, X, X_0, \mathsf{Act}_\tau, n_0, N_T, E, \mathcal{X}, \lambda)$ où $(N, X, X_0, \mathsf{Act}_\tau, n_0, N_T, E)$ est un automate de test classique, $\mathcal{X}$ est un ensemble de variables et $\lambda : \mathcal{X} \longrightarrow 2^N$ est une fonction d'étiquetage qui associe à chaque variable un ensemble d'états (dans lesquels cette variable sera « libre »). Bien sûr, un automate de test simple peut être vu comme un automate de test non clos où l'ensemble des variables $\mathcal{X}$ est vide.

Soit T un automate de test (supposé non clos) et T' un automate de test (qui peut aussi être non clos). Nous définissons l'automate de test $T\{T'/Z\}$ comme étant l'automate obtenu à partir de T en remplaçant tous les états étiquetés par Z par l'automate T'. La construction est décrite sur la figure 9.14.

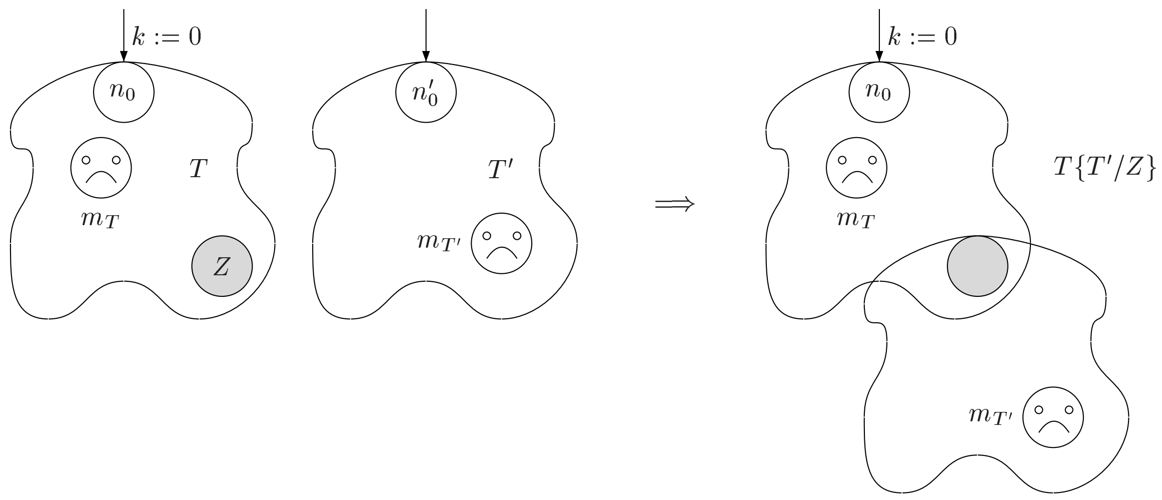


Figure 9.14: Construction de l'automate de test $T\{T'/Z\}$

Nous étendons alors les définitions de test aux formules non closes : nous disons qu'un automate de test (non clos) T teste une formule non close φ si pour chaque propriété testable $\varphi_1, \ldots, \varphi_k$, l'automate de test (clos) $T\{T_{\varphi_1}/Z_1, \ldots, T_{\varphi_k}/Z_k\}$ teste la formule close $\varphi\{\varphi_1/Z_1, \ldots, \varphi_k/Z_k\}$ (nous supposons que T_{φ_i} teste la formule φ_i et que $\{Z_1, \ldots, Z_k\}$ est l'ensemble des variables libres de la formule φ).

Nous allons maintenant procéder à une énumération de tous les cas.

PREUVE : Nous considérons les automates T_ψ dessinés sur la figure 9.15 (semblable à la figure 9.6). Nous allons montrer que chaque T_ψ teste la formule ψ. Ceci va être fait par induction sur la structure de ψ. Nous considérons dans toute la preuve un système de transitions temporisé $\mathcal{T}$.

➜ **Cas $\psi \equiv \mathtt{ff}$.** L'état rejetant de $T_{\mathtt{ff}}$ est accessible dans toute composition parallèle de $T_{\mathtt{ff}}$ avec un système de transitions temporisé.

➜ **Cas $\psi = g \vee \varphi$.** Nous supposons d'abord qu'un état rejetant de $T_{g\vee\varphi}$ est accessible à partir de l'état initial $(s_0 \parallel (n_0, [k_0 \leftarrow 0]w_0))$, où w_0 est une valuation initiale donnée, dans $(\mathcal{T} \parallel \mathcal{T}_{T_{g\vee\varphi}})$. Cela signifie que l'exécution suivante est autorisée dans la composition parallèle :

$$\begin{array}{rcl}(s_0 \parallel (n_0, [k \leftarrow 0]w_0)) & \xrightarrow{\tau}{}^{*} & (s \parallel (n_0, [k \leftarrow 0]w_0)) \\ & \xrightarrow{\tau} & (s \parallel (n_1, [k \leftarrow 0]w_0)) \\ & \rightsquigarrow^{*} & (s' \parallel (n_T, w_T))\end{array}$$

où, en particulier, $(k = 0 \wedge \neg g)([k \leftarrow 0]w_0)$ est vraie et donc, $g(w_0)$ est fausse. En appliquant l'hypothèse d'induction à φ, nous obtenons aussi que $(s, w_0) \not\models \varphi$. Ainsi, nous en déduisons que $(s_0, w_0) \not\models g \vee \varphi$.

Réciproquement, supposons que $(s_0, w_0) \not\models g \vee \varphi$, que $g(w_0)$ est fausse et qu'un état rejetant est accessible dans la composition parallèle $(\mathcal{T} \parallel T_\varphi)$. L'exécution suivante est alors autorisée (k est une nouvelle horloge n'apparaissant pas dans T_φ) :

$$\begin{array}{rcl}(s_0 \parallel (n_0, [k \leftarrow 0]w_0)) & \xrightarrow{\tau} & (s_0 \parallel (n_1, [k \leftarrow 0]w_0)) \\ & \rightsquigarrow^{*} & (s' \parallel (n_T, w_T)).\end{array}$$

Ainsi, l'état rejetant de $T_{g\vee\varphi}$ est accessible à partir de s_0 dans la composition parallèle $(\mathcal{T} \parallel T_{g\vee\varphi})$.

➜ **Cas $\psi = [a]\varphi$.** Nous commençons par supposer qu'un état rejetant de $T_{[a]\varphi}$ est accessible à partir de $(s_0 \parallel (n_0, [k \leftarrow 0]w_0))$ dans $(\mathcal{T} \parallel \mathcal{T}_{T_{[a]\varphi}})$. Ainsi, l'exécution suivante est autorisée :

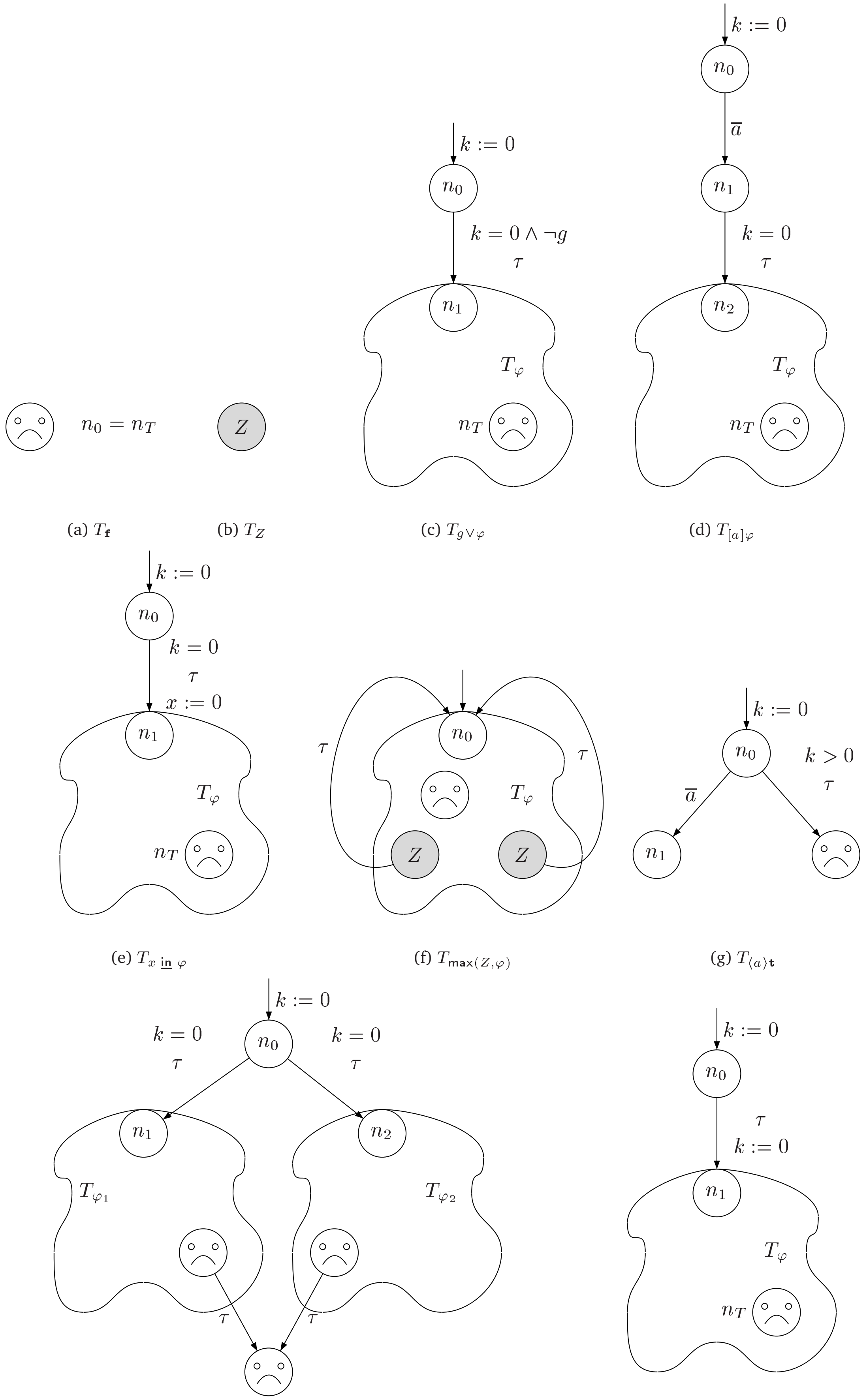
$$\begin{array}{rcl}(s_0 \parallel (n_0, [k \leftarrow 0]w_0)) & \xrightarrow{\tau}{}^{*} & (s \parallel (n_0, [k \leftarrow 0]w_0)) \\ & \xrightarrow{\tau = a/\overline{a}} & (s' \parallel (n_1, [k \leftarrow 0]w_0)) \\ & \xrightarrow{\tau}{}^{*} & (s'' \parallel (n_1, [k \leftarrow 0]w_0)) \\ & \xrightarrow{\tau} & (s'' \parallel (n_2, [k \leftarrow 0]w_0)) \\ & \rightsquigarrow^{*} & (s''' \parallel (n_T, w_T).\end{array}$$

Cela signifie précisément que $s_0 \overset{a}{\Longrightarrow} s''$ et que $(s'', [k \leftarrow 0]w_0) \not\models \varphi$, donc $(s_0, [k \leftarrow 0]w_0) \not\models [a]\varphi$.

Réciproquement, supposons que $(s_0, [k \leftarrow 0]w_0]) \not\models [a]\varphi$: il existe un état s_1 tel que $s_0 \overset{a}{\Longrightarrow} s_1$ et $(s_1, [k \leftarrow 0]w_0) \not\models \varphi$. Ainsi, dans la composition parallèle $(\mathcal{T} \parallel T_{[a]\varphi})$,

$$\begin{array}{rcl}(s_0 \parallel (n_0, [k \leftarrow 0]w_0)) & \xrightarrow{\tau}{}^{*} \xrightarrow{\tau = a/\overline{a}} & (s_1 \parallel (n_1, [k \leftarrow 0]w_0)) \\ & \xrightarrow{\tau} & (s_1 \parallel (n_2, [k \leftarrow 0]w_0)) \\ & \rightsquigarrow^{*} & (s_2 \parallel (n_T, w_T))\end{array}$$

ce qui signifie qu'un état rejetant est accessible à partir de $(s_0 \parallel (n_0, [k \leftarrow 0]w_0))$ dans $(\mathcal{T} \parallel T_{[a]\varphi})$.



(a) $T_{\mathtt{f}}$ (b) T_Z (c) $T_{g \vee \varphi}$ (d) $T_{[a]\varphi}$

(e) $T_{x \text{ \underline{in} } \varphi}$ (f) $T_{\mathsf{max}(Z,\varphi)}$ (g) $T_{\langle a \rangle \mathtt{t}}$

(h) $T_{\varphi_1 \wedge \varphi_2}$ (i) $T_{\forall \varphi}$

Figure 9.15: Automates de test pour les sous-formules de SBLL

➜ **Cas $\psi = x \ \underline{\text{in}}\ \varphi$.** Nous supposons tout d'abord qu'un état rejetant de $T_{x \ \underline{\text{in}}\ \varphi}$ est accessible à partir de $(s_0 \parallel (n_0, [k \leftarrow 0]w_0))$ dans $(\mathcal{T} \parallel \mathcal{T}_{\mathcal{T}_{\S \ \underline{\text{in}}\ \varphi}})$. L'exécution suivante est autorisée :

$$\begin{array}{rcl}(s_0 \parallel (n_0, [k \leftarrow 0]w_0)) & \xrightarrow{\tau}^* & (s \parallel (n_0, [k \leftarrow 0]w_0)) \\ & \xrightarrow{\tau} & (s \parallel (n_1, [k, x \leftarrow 0]w_0)) \\ & \rightsquigarrow^* & (s' \parallel (n_T, w_T)).\end{array}$$

Ceci signifie précisément que $(s, [x, k \leftarrow 0]w_0) \not\models \varphi$, ainsi $(s_0, [k \leftarrow 0]w_0) \not\models x \ \underline{\text{in}}\ \varphi$.

Réciproquement, supposons que $(s_0, [k \leftarrow 0]w_0) \not\models x \ \underline{\text{in}}\ \varphi$: $(s_0, [k, x \leftarrow 0]w_0) \not\models \varphi$. Ainsi, dans la composition parallèle $(\mathcal{T} \parallel T_{x \ \underline{\text{in}}\ \varphi})$,

$$\begin{array}{rcl}(s_0 \parallel (n_0, [k \leftarrow 0]w_0)) & \xrightarrow{\tau} & (s_0 \parallel (n_1, [k, x \leftarrow 0]w_0)) \\ & \rightsquigarrow^* & (s \parallel (n_T, w_T))\end{array}$$

ce qui signifie qu'un état rejetant est accessible à partir de $(s_0 \parallel (n_0, [k \leftarrow 0]w_0))$ dans $(\mathcal{T} \parallel T_{x \ \underline{\text{in}}\ \varphi})$.

➜ **Cas $\psi = Z$.** Il est immédiat que l'automate de test non clos dessiné sur la figure 9.15(b) teste la formule Z (dans le sens défini juste avant la preuve).

➜ **Cas $\psi = \max(Z, \varphi)$.** Il est difficile de montrer la propriété directement sur la formule récursive, nous définissons donc inductivement les formules suivantes :

$$\begin{array}{rcl}\psi^{(0)} & = & \varphi\{\mathtt{tt}/Z\} \\ \psi^{(l+1)} & = & \varphi\{\psi^{(l)}/Z\}\end{array}$$

Soit u une valuation. Comme propriété facile des points fixes maximaux [Tar55, Ace94], nous obtenons que

$$(s_0, u) \models \max(Z, \varphi) \iff \forall l \geq 0.\ (s_0, u) \models \psi^{(l)}.$$

L'automate de test pour $\psi^{(l)}$ est obtenu par une juxtaposition séquentielle de l copies de T_φ, plus précisément $T_{\psi^{(0)}}$ correspond à $T_\varphi\{T_{\mathtt{tt}}/Z\}$ et $T_{\psi^{(l+1)}}$ correspond à $T_\varphi\{T_{\psi^{(l)}}/Z\}$ où $T_{\mathtt{tt}}$ est l'automate de test avec un unique état (initial) et sans état rejetant.
Ainsi, nous avons juste à montrer qu'un état rejetant de $T_{\max(Z,\varphi)}$ est accessible dans la composition parallèle $(\mathcal{T} \parallel T_{\max(Z,\varphi)})$ si et seulement si, pour un entier l, un état rejetant de $T_{\psi^{(l)}}$ est accessible dans la composition parallèle $(\mathcal{T} \parallel T_{\psi^{(l)}})$.
Dans ce but, nous définissons pour chaque entier l une projection π_l de $T_{\psi^{(l)}}$ sur $T_{\max(Z,\varphi)}$ telle que, si m est un état de $T_{\psi^{(l)}}$, $\pi_l(m)$ est le « même » état dans $T_{\max(Z,\varphi)}$ (rappelons que $T_{\psi^{(l)}}$ est une juxtaposition de copies de T_φ et que $T_{\mathsf{max}(Z,\varphi)}$ a le même ensemble d'états que T_φ) et $\pi_l(m \xrightarrow{\alpha} m')$ est $\pi_l(m) \xrightarrow{\alpha} \pi_l(m')$.
Nous avons les deux propriétés suivantes :

$$\begin{array}{ccc}(m, u) \xrightarrow{\alpha} (m', u') \text{ dans } T_{\psi^{(l)}} & \Longrightarrow & (\pi_l(m), u) \xrightarrow{\alpha} (\pi_l(m'), u') \text{ dans } T_{\mathsf{max}(Z,\varphi)} \\ & \text{et} & \\ m \text{ état rejetant de } T_{\psi^{(l)}} & \Longleftrightarrow & \pi_l(m) \text{ état rejetant de } T_{\mathsf{max}(Z,\varphi)}.\end{array}$$

Si

$$(n_{i_0}, u_0) \xrightarrow{\alpha_1} (n_{i_1}, u_1) \xrightarrow{\alpha_2} \dots \xrightarrow{\alpha_p} (n_{i_p}, u_p)$$

est une exécution dans $T_{\psi^{(l)}}$, alors

$$(\pi_l(n_{i_0}), u_0) \xrightarrow{\alpha_1} (\pi_l(n_{i_1}), u_1) \xrightarrow{\alpha_2} \dots \xrightarrow{\alpha_p} (\pi_l(n_{i_p}), u_p)$$

est une exécution dans $T_{\max(Z,\varphi)}$. Ainsi, si un état rejetant est accessible dans la composition parallèle $(\mathcal{T} \parallel T_{\psi^{(l)}})$, alors un état rejetant est accessible dans la composition parallèle $(\mathcal{T} \parallel T_{\max(Z,\varphi)})$.

Réciproquement, considérons une exécution dans $T_{\mathsf{max}(Z,\varphi)}$

$$(n_{i_0}, u_0) \xrightarrow{\alpha_1} (n_{i_1}, u_1) \ldots \xrightarrow{\alpha_p} (n_{i_p}, u_p) \tag{9.4}$$

et définissons J comme étant l'ensemble suivant d'indices

$$J = \{j \in \{1, \ldots, p\} \mid \text{la transition } \xrightarrow{\alpha_j} \text{ est nouvelle}\}$$

où une transition est dite *nouvelle* si elle est dans l'automate de test $T_{\mathsf{max}(Z,\varphi)}$, mais pas dans T_φ (*i.e.* c'est une transition $m \xrightarrow{\tau} n_0$ où m est étiqueté par Z alors que n_0 est initial dans T_φ). Définissons l par $l = \#J + 1$ où $\#J$ représente le cardinal de J. Nous allons montrer qu'il existe un chemin dans $T_{\psi^{(l)}}$

$$(m'_{i_0}, u_0) \xrightarrow{\alpha_1} (m'_{i_1}, u_1) \ldots \xrightarrow{\alpha_p} (m'_{i_p}, u_p)$$

dont l'image par π_l est le chemin (9.4).

Supposons que $J = \{j_1, \ldots j_{l-1}\}$ avec $j_i < j_{i+1}$ pour chaque $1 \leq i < l-1$. Pour chaque $1 \leq i < l-1$, pour chaque $j_i \leq h < j_{i+1}$, l'état m'_{i_h} est défini comme étant l'unique état de la $i^{\text{ème}}$ copie de T_φ dans $T_{\mathsf{max}(Z,\varphi)}$ tel que $\pi_l(m'_{i_h}) = n_{i_h}$. Le chemin

$$(m'_{i_0}, u_0) \xrightarrow{\alpha_1} (m'_{i_1}, u_1) \ldots \xrightarrow{\alpha_p} (m'_{i_p}, u_p)$$

est un chemin de $T_{\psi^{(l)}}$ dont l'image par π_l est (9.4).

Nous avons donc montré que si un état rejetant est accessible dans la composition parallèle $(\mathcal{T} \parallel T_{\max(Z,\varphi)})$, alors un état rejetant est accessible dans la composition parallèle $(\mathcal{T} \parallel T_{\psi^{(l)}})$ pour un entier l donné.

➜ **Cas** $\psi = \varphi_1 \wedge \varphi_2$**.** Ce cas est très facile et la preuve est omise.

➜ **Cas** $\psi = \forall\!\!\!\forall\varphi$**.** Supposons qu'un état rejetant est accessible à partir de

$$(s_0 \parallel (n_0, [k \leftarrow 0]w_0))$$

dans la composition parallèle $(\mathcal{T} \parallel T_{\forall\varphi})$. L'exécution suivante est autorisée (k est une horloge n'apparaissant pas dans T_φ) : il existe un réel $d \geq 0$ tel que

$$\begin{array}{rcl} (s_0 \parallel (n_0, [k \leftarrow 0]w_0)) & \xrightarrow{\epsilon(d)} & (s_1 \parallel (n_0, ([k \leftarrow 0]w_0) + d)) \\ & \xrightarrow{\tau} & (s_1 \parallel (n_1, ([k \leftarrow 0](w_0 + d))) \\ & \rightsquigarrow^* & (s_2 \parallel (n_T, w_T)). \end{array} \tag{9.5}$$

En particulier, $(s_1, w_0 + d) \not\models \varphi$ (k n'est pas utile pour φ). Ainsi, $(s_0, [k \leftarrow 0]w_0) \not\models \forall\!\!\!\forall\varphi$.

Réciproquement, supposons que $(s_0, w_0) \not\models \forall\!\!\!\forall\varphi$. Alors une exécution comme (9.5) est autorisée dans la composition parallèle $(\mathcal{T} \parallel T_{\forall\varphi})$: un état rejetant est donc accessible.

Nous avons terminé la preuve pour SBLL^-, nous avons montré que toutes les formules de SBLL^- sont testables. Il nous reste à montrer que la logique SBLL est testable **pour les automates temporisés**, c'est-à-dire que nous allons montrer que si nous nous restreignons aux systèmes de transitions temporisés provenant d'automates temporisés, il est possible de construire un automate de test pour chaque formule de SBLL qui permet de la tester. Tout ce que nous venons de faire est encore valable dans ce cadre, il nous suffit donc de traiter le cas où ψ est de la forme $\langle a\rangle\mathtt{tt}$ pour conclure la preuve.

➜ **Cas** $\psi = \langle a \rangle \mathtt{tt}$**.** Le système de transitions temporisé $\mathcal{T}$ est supposé provenir d'un automate temporisé $\mathcal{A}$. Nous commençons par supposer qu'un état rejetant de $T_{\langle a \rangle \mathtt{tt}}$ est accessible à partir de l'état initial (q_0, u_0) de $\mathcal{A}$ si nous mettons en parallèle $\mathcal{A}$ et $T_{\langle a \rangle \mathtt{tt}}$ avec w_0 une valuation initiale pour $T_{\langle a \rangle \mathtt{tt}}$. Cela signifie que l'exécution suivante est autorisée dans la composition parallèle $(\mathcal{A} \parallel T_{\langle a \rangle \mathtt{tt}})$: il existe des réels $d, d' > 0$ tels que

$$\begin{aligned}
((q_0, u_0) \parallel (n_0, [k \leftarrow 0]w_0)) &\xrightarrow{\tau}{}^{*} ((q_1, u_1) \parallel (n_0, [k \leftarrow 0]w_0)) \\
&\xrightarrow{\epsilon(d')} ((q_2, u_2) \parallel (n_0, ([k \leftarrow 0]w_0) + d')) \\
&\overset{\epsilon(d-d')}{\Longrightarrow} ((q_3, u_3) \parallel (n_0, ([k \leftarrow 0]w_0) + d)) \\
&\xrightarrow{\tau} ((q_3, u_3) \parallel (n_T, ([k \leftarrow 0]w_0) + d)).
\end{aligned}$$

De plus, comme a est urgente (et donc $\overline{a}$ est aussi urgente), cela signifie que $(q_1, u_1) \overset{a}{\not\longrightarrow}$ et donc que $((q_0, u_0), [k \leftarrow 0]w_0) \not\models \langle a \rangle \mathtt{tt}$.

Réciproquement, supposons que $((q_0, u_0), w_0) \not\models \langle a \rangle \mathtt{tt}$. Cela signifie qu'il existe un état de $\mathcal{A}$, (q_1, u_1) tel que $(q_0, u_0) \xrightarrow{\tau}{}^{*} (q_1, u_1)$ et $(q_1, u_1) \overset{a}{\not\longrightarrow}$. Ainsi, l'exécution suivante est autorisée dans la composition parallèle :

$$\begin{aligned}
((q_0, u_0) \parallel (n_0, [k \leftarrow 0]w_0)) &\xrightarrow{\tau}{}^{*} ((q_1, u_1) \parallel (n_0, [k \leftarrow 0]w_0)) \\
&\xrightarrow{\epsilon(1)} ((q_1, u_1 + 1) \parallel (n_0, ([k \leftarrow 0]w_0) + 1)) \\
&\xrightarrow{\tau} ((q_1, u_1 + 1) \parallel (n_T, ([k \leftarrow 0]w_0) + 1)).
\end{aligned}$$

Un état rejetant est donc accessible dans $(\mathcal{A} \parallel T_{\langle a \rangle \mathtt{tt}})$.

Ceci conclut la preuve du théorème 9.15. □

B. Preuve du théorème 9.36

Théorème 9.36 *Soit φ une formule close de $\mathcal{L}_{\forall S}^{-}$. Supposons que s soit un état d'un système de transitions et que n' soit un état d'un automate temporisé sur l'ensemble d'horloges X. Supposons en outre que u et v' soient des valuations pour les ensembles disjoints d'horloges* clocks(φ) *et X respectivement. Alors*

$$(s \parallel (n', v'), u) \models \varphi \iff (s, v' : u) \models \varphi / n'.$$

Nous rappelons la définition du quotient dans le tableau 9.16 (c'est le même que le tableau 9.13).

PREUVE : Nous allons montrer les deux implications séparément.

Implication « Si »

Nous avons pour but de montrer que $(s, v' : u) \models \varphi / n'$ implique $(s \parallel (n', v'), u) \models \varphi$. À cette fin, il est suffisant de montrer que la relation $\mathcal{R}$ définie ci-dessous satisfait les implications de la relation $\models$:

$$\mathcal{R} \stackrel{\text{def}}{=} \{((s \parallel (n', v'), u), \varphi) \mid (s, v' : u) \models \varphi / n'\}$$

Supposons que $((s \parallel (n', v'), u), \varphi) \in \mathcal{R}$. Nous montrons que $\mathcal{R}$ est une relation de satisfaisabilité par une disjonction de cas sur la forme de φ.

➜ **Cas** $\varphi \equiv \mathtt{ff}$**.** Ce cas est évident.

$$
\begin{array}{rcl}
\mathrm{ff}/n' & \stackrel{\mathrm{def}}{=} & \mathrm{ff} \\
(\phi_1 \wedge \phi_2)/n' & \stackrel{\mathrm{def}}{=} & \phi_1/n' \wedge \phi_2/n' \\
& & \wedge \bigwedge_{e \in E(n',\tau)} (g_e \Rightarrow r_e \ \underline{\mathsf{in}}\ (\phi_1 \wedge \phi_2)/n'_e) \\
& & \wedge \bigwedge_{b \in \mathsf{Act}} \bigwedge_{e \in E(n',b)} (g_e \Rightarrow [\overline{b}]\,(r_e \ \underline{\mathsf{in}}\ (\phi_1 \wedge \phi_2)/n'_e)) \\
(g \vee \phi)/n' & \stackrel{\mathrm{def}}{=} & (g \vee (\phi/n')) \\
& & \wedge \bigwedge_{e \in E(n',\tau)} (g_e \Rightarrow r_e \ \underline{\mathsf{in}}\ (g \vee \phi)/n'_e) \\
& & \wedge \bigwedge_{b \in \mathsf{Act}} \bigwedge_{e \in E(n',b)} (g_e \Rightarrow [\overline{b}]\,(r_e \ \underline{\mathsf{in}}\ (g \vee \phi)/n'_e)) \\
([a]\phi)/n' & \stackrel{\mathrm{def}}{=} & [a](\phi/n') \\
& & \wedge \bigwedge_{e \in E(n',a)} (g_e \Rightarrow r_e \ \underline{\mathsf{in}}\ \phi/n'_e) \\
& & \wedge \bigwedge_{e \in E(n',\tau)} (g_e \Rightarrow r_e \ \underline{\mathsf{in}}\ ([a]\phi)/n'_e) \\
& & \wedge \bigwedge_{b \in \mathsf{Act}} \bigwedge_{e \in E(n',b)} (g_e \Rightarrow [\overline{b}]\,(r_e \ \underline{\mathsf{in}}\ ([a]\phi)/n'_e)) \\
(x \ \underline{\mathsf{in}}\ \phi)/n' & \stackrel{\mathrm{def}}{=} & x \ \underline{\mathsf{in}}\ (\phi/n') \\
& & \wedge \bigwedge_{e \in E(n',\tau)} (g_e \Rightarrow r_e \ \underline{\mathsf{in}}\ (x \ \underline{\mathsf{in}}\ \phi)/n'_e) \\
& & \wedge \bigwedge_{b \in \mathsf{Act}} \bigwedge_{e \in E(n',b)} (g_e \Rightarrow [\overline{b}]\,(r_e \ \underline{\mathsf{in}}\ (x \ \underline{\mathsf{in}}\ \phi)/n'_e)) \\
(\forall\!\!\!\forall_S \phi)/n' & \stackrel{\mathrm{def}}{=} & \forall\!\!\!\forall_{S \cup \overline{\Sigma_u(n')}}\ (\phi/n' \\
& & \wedge \bigwedge_{e \in E(n',\tau)} (g_e \Rightarrow r_e \ \underline{\mathsf{in}}\ (\forall\!\!\!\forall_S \phi)/n'_e) \\
& & \wedge \bigwedge_{b \in \mathsf{Act}} \bigwedge_{e \in E(n',b)} (g_e \Rightarrow [\overline{b}]\,(r_e \ \underline{\mathsf{in}}\ (\forall\!\!\!\forall_S \phi)/n'_e))) \qquad \text{si } S \cap \Sigma_u(n') = \varnothing \\
(\forall\!\!\!\forall_S \phi)/n' & \stackrel{\mathrm{def}}{=} & (\phi/n' \\
& & \wedge \bigwedge_{e \in E(n',\tau)} (g_e \Rightarrow r_e \ \underline{\mathsf{in}}\ (\forall\!\!\!\forall_S \phi)/n'_e) \\
& & \wedge \bigwedge_{b \in \mathsf{Act}} \bigwedge_{e \in E(n',b)} (g_e \Rightarrow [\overline{b}]\,(r_e \ \underline{\mathsf{in}}\ (\forall\!\!\!\forall_S \phi)/n'_e))) \qquad \text{si } S \cap \Sigma_u(n') \neq \varnothing \\
Z/n' & \stackrel{\mathrm{def}}{=} & Z \\
\mathsf{max}(Z,\phi)/n' & \stackrel{\mathrm{def}}{=} & (\phi\{\mathsf{max}(Z,\phi)/Z\})/n' \\
& & \wedge \bigwedge_{e \in E(n',\tau)} (g_e \Rightarrow r_e \ \underline{\mathsf{in}}\ \mathsf{max}(Z,\phi)/n'_e) \\
& & \wedge \bigwedge_{b \in \mathsf{Act}} \bigwedge_{e \in E(n',b)} (g_e \Rightarrow [\overline{b}]\,(r_e \ \underline{\mathsf{in}}\ (\mathsf{max}(Z,\phi))/n'_e))
\end{array}
$$

Tableau 9.16: Construction du quotient pour $\mathcal{L}^-_{\forall\!\!\!\forall S}$

➜ **Cas $\varphi \equiv \phi_1 \wedge \phi_2$.** Supposons que

$$s \parallel (n', v') \xrightarrow{\tau}{}^{*} s_1 \parallel (n'_1, v'_1) \tag{9.6}$$

Soit k le nombre d'étapes auxquelles le second composant de la composition parallèle participe durant l'exécution (9.6). Par induction sur k, nous allons montrer que, pour $j = 1, 2$, $((s_1 \parallel (n'_1, v'_1), u), \phi_j) \in \mathcal{R}$.

- <u>CAS DE BASE</u>. Si $k = 0$, alors $n' = n'_1$, $v' = v'_1$ et $s \xrightarrow{\tau}{}^{*} s_1$. Comme $(s, v' : u) \models \varphi/n'$, de par la définition de la formule quotient φ/n', nous obtenons que $(s_1, v' : u) \models \phi_1/n'$ et $(s_1, v' : u) \models \phi_2/n'$. Par définition de $\mathcal{R}$, il vient que $((s_1 \parallel (n'_1, v'_1), u), \phi_j) \in \mathcal{R}$ pour $j = 1, 2$, ce que nous voulions justement montrer.
- <u>ÉTAPE D'INDUCTION</u>. Supposons que le résultat a été prouvé pour k_0 et que $k = k_0 + 1$. L'exécution (9.6) est de la forme suivante :

$$\begin{array}{rcl} s \parallel (n', v') & \xrightarrow{\tau}{}^{*} & s_3 \parallel (n', v') \\ & \xrightarrow{\tau} & s_2 \parallel (n'_2, v'_2) \\ & \xrightarrow{\tau}{}^{*} & s_1 \parallel (n'_1, v'_1) \end{array}$$

  Nous analysons les deux formes que peut prendre la transition

$$s_3 \parallel (n', v') \xrightarrow{\tau} s_2 \parallel (n'_2, v'_2).$$

  – **Cas** $(n', v') \xrightarrow{\tau} (n'_2, v'_2)$ **et** $s_3 = s_2$. Dans ce cas, il y a une transition $e \in E(n', \tau)$ telle que $n'_e = n'_e$, $v'_2 = [r_e \leftarrow 0]v'$ et $g_e(v')$ est Vrai. Comme $(s, v' : u) \models \varphi/n'$ et $s \xrightarrow{\tau}{}^{*} s_3$, par la définition de la formule quotient φ/n' nous pouvons dire que $(s_3, [r_e \leftarrow 0]v' : u) \models \varphi/n'_2$. Par définition de la relation $\mathcal{R}$, il vient que

$$(s_2 \parallel (n'_2, v'_2), u), \varphi) \in \mathcal{R}$$

  Nous pouvons maintenant appliquer l'hypothèse d'induction à l'exécution

$$s_2 \parallel (n'_2, v'_2) \xrightarrow{\tau}{}^{*} s_1 \parallel (n'_1, v'_1)$$

  pour obtenir que $((s_1 \parallel (n'_1, v'_1), u), \phi_j) \in \mathcal{R}$ pour $j = 1, 2$, ce que nous voulions montrer.

  – **Cas** $(n', v') \xrightarrow{b} (n'_2, v'_2)$ **et** $s_3 \xrightarrow{\bar{b}} s_2$ **pour un** $b \in \mathsf{Act}$. La preuve est identique à celle du cas précédent et n'est donc pas détaillée.

  Ceci termine l'étape d'induction et finit donc la preuve pour le cas $\varphi \equiv \phi_1 \wedge \phi_2$.

➜ **Cas $\varphi \equiv g \vee \phi$.** Supposons que

$$s \parallel (n', v') \xrightarrow{\tau}{}^{*} s_1 \parallel (n'_1, v'_1) \tag{9.7}$$

Soit k le nombre d'étapes auxquelles la deuxièmes composante participe durant l'exécution (9.7). Par induction sur k, nous montrons que soit $g(v'_1 : u)$ est vrai, soit $((s_1 \parallel (n'_1, v'_1), u), \phi) \in \mathcal{R}$.

- <u>CAS DE BASE</u>. Si $k = 0$, alors $n' = n'_1$, $v' = v'_1$ et $s \xrightarrow{\tau}{}^{*} s_1$. Comme $(s, v' : u) \models \varphi/n'$, par définition de la formule quotient φ/n', nous obtenons que soit $g(v'_1 : u)$ est vrai, soit $(s_1, v' : u) \models \phi/n'$. Par définition de $\mathcal{R}$, il vient que soit $g(v'_1 : u)$ est vrai, soit $((s_1 \parallel (n'_1, v'_1), u), \phi) \in \mathcal{R}$, ce que nous voulions montrer.
- <u>ÉTAPE D'INDUCTION</u>. Supposons que le résultat soit prouvé pour k_0 et que $k = k_0 + 1$. L'exécution (9.7) a la forme suivante :

$$\begin{array}{rcl} s \parallel (n', v') & \xrightarrow{\tau}{}^{*} & s_3 \parallel (n', v') \\ & \xrightarrow{\tau} & s_2 \parallel (n'_2, v'_2) \\ & \xrightarrow{\tau}{}^{*} & s_1 \parallel (n'_1, v'_1) \end{array}$$

Nous considérons les deux formes que peut prendre la transition

$$s_3 \parallel (n', v') \xrightarrow{\tau} s_2 \parallel (n'_2, v'_2).$$

– **Cas** $(n', v') \xrightarrow{\tau} (n'_2, v'_2)$ **et** $s_3 = s_2$. Dans ce cas, il y a une transition $e \in E(n', \tau)$, telle que $n'_2 = n'_2$, $v'_2 = [r_e \leftarrow 0]v'$ et $g_e(v')$ est vrai. Comme $(s, v' : u) \models \varphi/n'$ et $s \xrightarrow{\tau}^* s_3$, par définition de la formule quotient φ/n', nous obtenons que $(s_3, [r \leftarrow 0]v' : u) \models \varphi/n'$. Par définition de la relation $\mathcal{R}$, il vient que

$$(s_2 \parallel (n'_2, v'_2), u), \varphi) \in \mathcal{R}$$

Nous appliquons maintenant l'hypothèse d'induction à l'exécution

$$s_2 \parallel (n'_2, v'_2) \xrightarrow{\tau}^* s_1 \parallel (n'_1, v'_1)$$

pour obtenir que soit $g(v'_1 : u)$ est vrai, soit $((s_1 \parallel (n'_1, v'_1), u), \phi) \in \mathcal{R}$, ce que nous voulions montrer.

– **Cas** $(n', v') \xrightarrow{b} (n'_2, v'_2))$ **et** $s_3 \xrightarrow{\bar{b}} s_2$ **pour un** $b \in \mathsf{Act}$. La preuve est identique à celle du cas précédent et n'est donc pas détaillée.

Ceci complète l'étape d'induction et termine la preuve pour le cas $\varphi \equiv q \vee \phi$.

➜ **Cas** $\varphi \equiv [a]\phi$**.** Supposons que

$$s \parallel (n', v') \xRightarrow{a} s_1 \parallel (n'_1, v'_1) \tag{9.8}$$

Nous montrons que $((s_1 \parallel (n'_1, v'_1), u), \phi) \in \mathcal{R}$ par induction sur le nombre d'étapes k dans lesquelles la seconde composante participe durant l'exécution (9.8).

- <u>Cas de base</u>. Si $k = 0$, alors $n' = n'_1$, $v' = v'_1$ et $s \xRightarrow{a} s_1$. Comme $(s, v' : u) \models \varphi/n'$, par définition de la formule quotient φ/n', nous obtenons que $(s_1, v' : u) \models \phi/n'$. Par définition de $\mathcal{R}$, il vient que $((s_1 \parallel (n'_1, v'_1), u), \phi) \in \mathcal{R}$, ce que nous voulions montrer.
- <u>Étape d'induction</u>. Supposons que le résultat soit prouvé pour k_0 et que $k = k_0 + 1$. Nous considérons les deux formes possibles que peut prendre l'exécution (9.8).

  – Supposons que l'exécution (9.8) prenne la forme

$$\begin{array}{lll} s \parallel (n', v') & \xrightarrow{\tau}^* & s_1 \parallel (n', v') \\ & \xrightarrow{a} & s_1 \parallel (n'_1, v'_1) \end{array}$$

  car $s \xrightarrow{\tau}^* s_1$ et $(n', v') \xrightarrow{a} (n'_1, v'_1)$. Comme $(n', v') \xrightarrow{a} (n'_1, v'_1)$, il y a une transition $e \in E(n', a)$ telle que $n'_2 = n'_1$, $g_e(v')$ est vrai et $v'_1 = [r_e \leftarrow 0]v'$. Comme $(s, v' : u) \models \varphi/n'$ et $s \xrightarrow{\tau}^* s_1$, la définition de la formule quotient φ/n' donne maintenant que $(s_1, v'_1 : u) \models \phi/n'_1$. Par définition de $\mathcal{R}$, il vient que $((s_1 \parallel (n'_1, v'_1), u), \phi) \in \mathcal{R}$, ce que nous voulions montrer.

  – Supposons que l'exécution (9.8) prenne la forme

$$\begin{array}{lll} s \parallel (n', v') & \xrightarrow{\tau}^* & s_2 \parallel (n', v') \\ & \xrightarrow{\tau} & s_2 \parallel (n'_2, v'_2) \\ & \xRightarrow{a} & s_1 \parallel (n'_1, v'_1) \end{array}$$

  car $s \xrightarrow{\tau}^* s_2$ et $(n', v') \xrightarrow{\tau} (n'_2, v'_2)$. Comme $(n', v') \xrightarrow{\tau} (n'_2, v'_2)$, il y a une transition $e \in E(n', \tau)$ telle que $n'_2 = n'_2$, $g_e(v')$ est vrai et $v'_2 = [r_e \leftarrow 0]v'$. Comme $(s, v' : u) \models \varphi/n'$ et $s \xrightarrow{\tau}^* s_2$, la définition de la formule quotient φ/n' entraîne maintenant que $(s_2, v'_2 : u) \models \varphi/n'_2$. Par définition de $\mathcal{R}$, il vient que $((s_2 \parallel (n'_2, v'_2), u), \varphi) \in \mathcal{R}$. Nous pouvons donc appliquer l'hypothèse d'induction à l'exécution $s_2 \parallel (n'_2, v'_2) \xRightarrow{a} s_1 \parallel (n'_1, v'_1)$ pour obtenir que $((s_1 \parallel (n'_1, v'_1), u), \phi) \in \mathcal{R}$, ce que nous voulions montrer.

– Supposons que l'exécution (9.8) prenne la forme

$$\begin{aligned} s \parallel (n', v') &\xrightarrow{\tau}{}^{*} s_3 \parallel (n', v') \\ &\xrightarrow{\tau} s_2 \parallel (n'_2, v'_2) \\ &\overset{a}{\Longrightarrow} s_1 \parallel (n'_1, v'_1) \end{aligned}$$

car $(n', v') \xrightarrow{b} (n'_2, v'_2)$ et $s \xrightarrow{\tau}{}^{*} s_3 \xrightarrow{\bar{b}} s_2$ pour une action b de Act. (Remarquons que b peut être a.) Comme $(n', v') \xrightarrow{b} (n'_2, v'_2)$, il y a une transition $e \in E(n', b)$ telle que $n'_e = n'_2$, $g_e(v')$ est vrai et $v'_2 = [r_e \leftarrow 0]v'$. Comme $(s, v' : u) \models \varphi/n'$ et $s \xrightarrow{\tau}{}^{*} s_3$, la définition de la formule quotient φ/n' entraîne maintenant que $(s_2, v'_2 : u) \models \varphi/n'_2$. Par définition de $\mathcal{R}$, il vient que $((s_2 \parallel (n'_2, v'_2), u), \varphi) \in \mathcal{R}$. Nous pouvons appliquer l'hypothèse d'induction à l'exécution $s_2 \parallel (n'_2, v'_2) \overset{a}{\Longrightarrow} s_1 \parallel (n'_1, v'_1)$ pour obtenir que $((s_1 \parallel (n'_1, v'_1), u), \phi) \in \mathcal{R}$, ce que nous voulions montrer.

Ceci complète l'étape d'induction et termine la preuve pour le cas $\varphi \equiv [a]\phi$.

➜ **Cas** $\varphi = x$ <u>in</u> ϕ**.** Supposons que $(s, u) \parallel ((n', v'), u) \xrightarrow{\tau}{}^{*} (s_1, u) \parallel ((n'_1, v'_1), u)$. Nous montrons que $((s_1, u) \parallel ((n'_1, v'_1), [x \leftarrow 0]u), \phi) \in \mathcal{R}$. Soit k le nombre d'étapes auxquelles la deuxième composante prend part durant l'exécution. La preuve va être faite par induction sur k. Si $k = 0$, alors la deuxième composante reste dans l'état n', et $(s, v' : u) \xrightarrow{\tau}{}^{*} (s_1, v'_1 : u)$. Par définition du quotient, $(s_1, v'_1 : [x \leftarrow 0]u) \models \phi/n'$, et donc

$$((s_1, [x \leftarrow 0]u) \parallel ((n', v'_1), [x \leftarrow 0]u), \phi) \in \mathcal{R}$$

Maintenant, supposons que $k = k_0 + 1$ et que nous avons déjà résolu le problème pour k_0. Alors l'exécution a la forme suivante :

$$\begin{aligned} (s, u) \parallel ((n', v'), u) &\xrightarrow{\tau}{}^{*} (s_3, u) \parallel ((n', v'), u) \\ &\xrightarrow{\tau} (s_2, u) \parallel ((n'_2, v'_2), u) \\ &\xrightarrow{\tau}{}^{*} (s_1, u) \parallel ((n'_1, v'_1), u) \end{aligned}$$

Comme dans le cas précédent, il y a deux choix possibles pour la première étape de la deuxième composante, *i.e.* pour $(s_3, u) \parallel ((n', v'), u) \xrightarrow{\tau} (s_2, u) \parallel ((n'_2, v'_2), u)$: la deuxième composante fait une action interne τ ou bien une action b et se synchronise sur b avec la première composante. Ainsi, la preuve est semblable aux autres, et le résultat est obtenu grâce à l'hypothèse d'induction.

➜ **Cas** $\varphi = \forall\!\!\forall_S \phi$**.** Supposons que

$$s \parallel (n', v') \overset{\epsilon(d)}{\Longrightarrow}_S s_1 \parallel (n'_1, v'_1) \tag{9.9}$$

pour un $d \geq 0$. Nous considérons les cas $S \cap \Sigma_u(n') = \varnothing$ et $S \cap \Sigma_u(n') \neq \varnothing$ séparément.

– **Cas** $S \cap \Sigma_u(n') = \varnothing$. Nous montrons que $((s_1 \parallel (n'_1, v'_1), u + d), \phi) \in \mathcal{R}$ par induction sur le nombre d'étapes k auxquelles la deuxième composante participe pendant l'exécution (9.9).
  - <u>Cas de base</u>. Si $k = 0$, alors nécessairement $d = 0$, $n' = n'_1$, $v' = v'_1$ et $s \overset{\epsilon(0)}{\Longrightarrow}_S s_1$. Comme la relation de transitions $\overset{\epsilon(0)}{\Longrightarrow}_S$ coïncide avec $\xrightarrow{\tau}{}^{*}$ pour chaque ensemble S, il vient que $s \overset{\epsilon(0)}{\Longrightarrow}_{S \cup \overline{\Sigma_u(n')}} s_1$. Comme $(s, v' : u) \models \varphi/n'$, par définition de la formule quotient φ/n', nous obtenons que $(s_1, v' : u) \models \phi/n'$. Par définition de $\mathcal{R}$, il vient que $((s_1 \parallel (n'_1, v'_1), u), \phi) \in \mathcal{R}$, ce que nous voulions montrer.
  - <u>Étape d'induction</u>. Supposons que le résultat est prouvé pour k_0 et que $k = k_0 + 1$. Nous allons considérer les trois formes possibles que peut prendre l'exécution (9.9).

1. Supposons que l'exécution (9.9) prenne la forme

$$\begin{array}{rcl} s \parallel (n', v') & \xrightarrow{\tau}{}^{*} & s_2 \parallel (n', v') \\ & \xrightarrow{\tau} & s_2 \parallel (n'_2, v'_2) \\ & \overset{\epsilon(d)}{\Longrightarrow}_S & s_1 \parallel (n'_1, v'_1) \end{array}$$

car $s \xrightarrow{\tau}{}^{*} s_2$ et $(n', v') \xrightarrow{\tau} (n'_2, v'_2)$. Comme $(n', v') \xrightarrow{\tau} (n'_2, v'_2)$, il y a une transition $e \in E(n', \tau)$ telle que $n'_2 = n'_e$, $g_e(v')$ est vrai et $v'_2 = [r_e \leftarrow 0]v'$. Comme $(s, v' : u) \models \varphi/n'$ et $s \xrightarrow{\tau}{}^{*} s_2$, la définition de la formule quotient φ/n' entraîne que $(s_2, v'_2 : u) \models \varphi/n'_2$. Par définition de $\mathcal{R}$, il vient que

$$((s_2 \parallel (n'_2, v'_2), u), \varphi) \in \mathcal{R}$$

Nous appliquons alors l'hypothèse d'induction à l'exécution

$$s_2 \parallel (n'_2, v'_2) \overset{\epsilon(d)}{\Longrightarrow}_S s_1 \parallel (n'_1, v'_1)$$

pour obtenir que $((s_1 \parallel (n'_1, v'_1), u + d), \phi) \in \mathcal{R}$, ce que nous voulions montrer.

2. Supposons que l'exécution (9.9) prenne la forme

$$\begin{array}{rcl} s \parallel (n', v') & \xrightarrow{\tau}{}^{*} & s_3 \parallel (n', v') \\ & \xrightarrow{\epsilon(d')}_S & s_2 \parallel (n', v' + d') \\ & \overset{\epsilon(d'')}{\Longrightarrow} & s_1 \parallel (n'_1, v'_1) \end{array}$$

car $s \xrightarrow{\tau}{}^{*} s_3 \xrightarrow{\epsilon(d')}_S s_2$, $(n', v') \xrightarrow{\epsilon(d')}_S (n', v' + d')$ et $d = d' + d''$. Comme

$$s_3 \parallel (n', v') \xrightarrow{\epsilon(d')}_S s_2 \parallel (n', v' + d')$$

nous avons soit $d' = 0$, soit $s_3 \overset{\bar{a}}{\not\longrightarrow}$, pour chaque $a \in \Sigma_u(n')$. Dans ces deux cas, nous pouvons voir que $s \overset{\epsilon(d')}{\Longrightarrow}_{S \cup \overline{\Sigma_u(n')}} s_2$. Comme la formule φ/n' est équivalente à $\forall\!\!\forall_{S \cup \overline{\Sigma_u(n')}}(\varphi/n')$ (lemme 9.21(2)) et $(s, v' : u) \models \varphi/n'$, nous obtenons que $(s_2, v' + d' : u + d') \models \varphi/n'$. Par définition de $\mathcal{R}$, il vient que $((s_2 \parallel (n', v' + d'), u + d'), \varphi) \in \mathcal{R}$. Nous appliquons alors l'hypothèse d'induction à l'exécution $s_2 \parallel (n', v' + d') \overset{\epsilon(d'')}{\Longrightarrow} s_1 \parallel (n'_1, v'_1)$ pour obtenir que $((s_1 \parallel (n'_1, v'_1), u + d' + d''), \phi) \in \mathcal{R}$, ce que nous voulions montrer.

3. Supposons que l'exécution (9.9) prenne la forme

$$\begin{array}{rcl} s \parallel (n', v') & \xrightarrow{\tau}{}^{*} & s_3 \parallel (n', v') \\ & \xrightarrow{\tau} & s_2 \parallel (n'_2, v'_2) \\ & \overset{\epsilon(d)}{\Longrightarrow}_S & s_1 \parallel (n'_1, v'_1) \end{array}$$

car $(n', v') \xrightarrow{b} (n'_2, v'_2)$ et $s \xrightarrow{\tau}{}^{*} s_3 \xrightarrow{\bar{b}} s_2$ pour une action b de Act. La preuve de ce sous-cas est identique à celle du sous-cas 1 et n'est donc pas détaillée.

Ceci complète l'étape d'induction et termine la preuve du cas $S \cap \Sigma_u(n') = \varnothing$.

– **Cas** $S \cap \Sigma_u(n') \neq \varnothing$. Nous procédons par induction sur le nombre d'étapes de l'exécution (9.9).

• <u>CAS DE BASE</u>. Supposons que $s \parallel (n', v') \xrightarrow{\epsilon(d)}_S s_1 \parallel (n'_1, v'_1)$. Ceci implique, entre autres, que $s \xrightarrow{\epsilon(d)}_S s_1$, $(n', v') \xrightarrow{\epsilon(d)}_S (n'_1, v'_1)$ et $(n'_1, v'_1) = (n', v' + d)$. Comme $S \cap \Sigma_u(n') \neq \varnothing$, l'état (n', v') autorise une action urgente dans l'ensemble S. Ceci entraîne que $d = 0$, et, ainsi, que $v'_1 = v'$ et $s = s_1$. Comme $(s, v' : u) \models \varphi/n'$, par définition de la formule quotient, nous obtenons que $(s, v' : u) \models \phi/n'$. La définition de $\mathcal{R}$ implique maintenant que

$$((s \parallel (n', v'), u), \phi) \in \mathcal{R}$$

ce que nous voulions montrer.

- ÉTAPE D'INDUCTION. Nous considérons les quatre formes que l'exécution (9.9) peut prendre.

  1. Supposons que l'exécution (9.9) prenne la forme

$$\begin{array}{lcl} s \parallel (n', v') & \xrightarrow{\tau} & s_2 \parallel (n', v') \\ & \stackrel{\epsilon(d)}{\Longrightarrow}_S & s_1 \parallel (n'_1, v'_1) \end{array}$$

car $s \xrightarrow{\tau} s_2$. Comme $(s, v' : u) \models \varphi/n'$ et $s \xrightarrow{\tau} s_2$, la proposition 9.19 implique que $(s_2, v'_2 : u) \models \varphi/n'_2$. Par définition de $\mathcal{R}$, il vient que

$$((s_2 \parallel (n', v'), u), \varphi) \in \mathcal{R}$$

Nous pouvons alors impliquer l'hypothèse d'induction à l'exécution

$$s_2 \parallel (n', v') \stackrel{\epsilon(d)}{\Longrightarrow}_S s_1 \parallel (n'_1, v'_1)$$

pour obtenir que $((s_1 \parallel (n'_1, v'_1), u + d), \phi) \in \mathcal{R}$, ce que nous voulions montrer.

  2. Supposons que l'exécution (9.9) prenne la forme

$$\begin{array}{lcl} s \parallel (n', v') & \xrightarrow{\tau} & s_1 \parallel (n'_2, v'_2) \\ & \stackrel{\epsilon(d)}{\Longrightarrow}_S & s_1 \parallel (n'_1, v'_1) \end{array}$$

car $(n', v') \xrightarrow{\tau} (n'_2, v'_2)$. Comme $(n', v') \xrightarrow{\tau} (n'_2, v'_2)$, il existe une transition $e \in E(n', \tau)$, telle que $n'_2 = n'_e$, $g_e(v')$ est vrai et $v'_2 = [r_e \leftarrow 0]v'$. Comme $(s, v' : u) \models \varphi/n'$, la définition de la formule quotient φ/n' donne que $(s, v'_2 : u) \models \varphi/n'_2$. Par définition de $\mathcal{R}$, il vient que

$$((s \parallel (n'_2, v'_2), u), \varphi) \in \mathcal{R}$$

Nous pouvons maintenant appliquer l'hypothèse d'induction à l'exécution

$$s \parallel (n'_2, v'_2) \stackrel{\epsilon(d)}{\Longrightarrow}_S s_1 \parallel (n'_1, v'_1)$$

pour obtenir que $((s_1 \parallel (n'_1, v'_1), u + d), \phi) \in \mathcal{R}$, ce que nous voulions montrer.

  3. Supposons que l'exécution (9.9) prenne la forme

$$\begin{array}{lcl} s \parallel (n', v') & \xrightarrow{\epsilon(d')}_S & s_2 \parallel (n', v' + d') \\ & \stackrel{\epsilon(d'')}{\Longrightarrow} & s_1 \parallel (n'_1, v'_1) \end{array}$$

car $s \xrightarrow{\epsilon(d')}_S s_2$, $(n', v') \xrightarrow{\epsilon(d')}_S (n', v' + d')$ et $d = d' + d''$. Comme $S \cap \Sigma_u(n') \neq \varnothing$, l'état (n', v') peut faire une action urgente de l'ensemble S. Ceci entraîne que $d' = 0$, et ainsi que $s = s_2$ et $d'' = d$. Nous pouvons donc appliquer l'hypothèse d'induction à l'exécution $s \parallel (n', v') \stackrel{\epsilon(d)}{\Longrightarrow} s_1 \parallel (n'_1, v'_1)$ pour montrer que $((s_1 \parallel (n'_1, v'_1), u + d' + d''), \phi) \in \mathcal{R}$, ce que nous voulions montrer.

4. Supposons que l'exécution (9.9) prenne la forme

$$\begin{array}{rcl} s \parallel (n', v') & \xrightarrow{\tau} & s_2 \parallel (n'_2, v'_2) \\ & \overset{\epsilon(d)}{\Longrightarrow}_S & s_1 \parallel (n'_1, v'_1) \end{array}$$

car $(n', v') \xrightarrow{b} (n'_2, v'_2)$ et $s \xrightarrow{\bar{b}} s_2$ pour une action b de Act. La preuve de ce sous-cas est identique à celle du sous-cas 1 et n'est donc pas détaillée.

Ceci complète l'étape d'induction et termine la preuve du cas $S \cap \Sigma_u(n') = \varnothing$.

La preuve pour le cas $\varphi \equiv \forall\!\!\!\forall_S \phi$ est maintenant complète.

➜ **Cas** $\varphi \equiv \mathsf{max}(Z, \phi)$**.** Supposons que

$$s \parallel (n', v') \xrightarrow{\tau}{}^* s_1 \parallel (n'_1, v'_1) \tag{9.10}$$

Nous allons montrer que $((s_1 \parallel (n'_1, v'_1), u), \phi\{\mathsf{max}(Z, \phi)/Z\}) \in \mathcal{R}$ par induction sur le nombre k d'étapes de l'exécution (9.10) auxquelles la deuxième composante participe.

- <u>Cas de base</u>. Si $k = 0$, alors nous avons que $s \xrightarrow{\tau}{}^* s_1$, $n'_1 = n'$ et $v'_1 = v'$. Comme $(s, v' : u) \models \varphi/n'$, nous obtenons que

$$(s_1, v' : u) \models (\phi\{\mathsf{max}(Z, \phi)/Z\})/n'$$

La définition de $\mathcal{R}$ entraîne alors que

$$((s_1 \parallel (n', v'), u), \phi\{\mathsf{max}(Z, \phi)/Z\}) \in \mathcal{R}$$

ce que nous voulions montrer.

- <u>Étape d'induction</u>. Supposons le résultat montré pour k_0 et que $k = k_0 + 1$. L'exécution (9.10) est de la forme suivante :

$$\begin{array}{rcl} s \parallel (n', v') & \xrightarrow{\tau}{}^* & s_3 \parallel (n', v') \\ & \xrightarrow{\tau} & s_2 \parallel (n'_2, v'_2) \\ & \xrightarrow{\tau}{}^* & s_1 \parallel (n'_1, v'_1) \end{array}$$

Nous allons distinguer deux formes possibles pour la transition

$$s_3 \parallel (n', v') \xrightarrow{\tau} s_2 \parallel (n'_2, v'_2)$$

  – **Cas** $(n', v') \xrightarrow{\tau} (n'_2, v'_2)$ **et** $s_3 = s_2$**.** Dans ce cas, il y a une transition $e \in E(n', \tau)$ telle que $n'_2 = n'_e$, $v'_2 = [r_e \leftarrow 0]v'$ et $g_e(v')$ est vrai. Comme $(s, v' : u) \models \varphi/n'$ et $s \xrightarrow{\tau}{}^* s_3$, par définition de la formule quotient φ/n', nous obtenons que $(s_3, [r \leftarrow 0]v' : u) \models \varphi/n'_2$. Par définition de la relation $\mathcal{R}$, il vient que

$$(s_2 \parallel (n'_2, v'_2), u), \varphi) \in \mathcal{R}$$

  Nous pouvons maintenant appliquer l'hypothèse d'induction à l'exécution

$$s_2 \parallel (n'_2, v'_2) \xrightarrow{\tau}{}^* s_1 \parallel (n'_1, v'_1)$$

  pour obtenir que

$$((s_1 \parallel (n'_1, v'_1), u), \phi\{\mathsf{max}(Z, \phi)/Z\}) \in \mathcal{R}$$

  ce que nous voulions montrer.

  – **Cas** $(n', v') \xrightarrow{b} (n'_2, v'_2))$ **et** $s_3 \xrightarrow{\bar{b}} s_2$ **où** $b \in \mathsf{Act}$**.** La preuve de ce sous-cas est identique à celle du cas précédent et n'est donc pas détaillée.

Ceci complète l'étape d'induction et termine la preuve pour le cas $\varphi \equiv \mathsf{max}(Z, \phi)$.

Ceci complète la preuve de l'implication « Si ».

Implication « Seulement si »

Nous voulons montrer que $(s \parallel (n', v'), u') \models \varphi$ implique $(s, v' : u) \models \varphi/n'$. Nous avons juste besoin de montrer que l'environnement ρ défini comme suit pour chaque formule φ et pour chaque état n'

$$[\![\varphi/n']\!]\rho \stackrel{\text{def}}{=} \{(s, v' : u) \mid (s \parallel (n', v'), u) \models \varphi\}$$

est un post-point fixe de la fonction monotone sur les environnements induite par la définition du quotient (voir la table 9.16) [Lar90, SI94].

Supposons que $(s, v' : u) \in [\![\varphi/n']\!]\rho$. Nous voulons montrer que l'état $(s, v' : u)$ est contenu dans l'interprétation de la partie droite de la formule définissant φ/n' (voir la table 9.16). Nous procédons par une analyse de cas sur la forme que peut prendre la formule φ.

➜ **Cas** $\varphi \equiv \mathtt{ff}$**.** Ce cas est évident.

➜ **Cas** $\varphi \equiv \phi_1 \wedge \phi_2$**.** Nous allons montrer que

$$(s, v' : u) \in \left[\!\!\left[\begin{array}{c} (\phi_1/n') \wedge (\phi_2/n') \wedge \bigwedge_{e \in E(n', \tau)} (g_e \Rightarrow r_e \ \underline{\mathsf{in}}\ (\varphi/n'_e)) \\ \wedge \\ \bigwedge_{b \in \mathsf{Act}} \ \bigwedge_{e \in E(n', b)} (g_e \ \underline{\mathsf{in}}\ [\overline{b}]\, (r_e \ \underline{\mathsf{in}}\ (\varphi/n'_e))) \end{array} \right]\!\!\right] \rho$$

où $[\![\psi]\!]\rho$ est l'ensemble des états satisfaisant ψ dans l'environnement ρ. À cette fin, supposons que $s \stackrel{\tau}{\longrightarrow}^* s_1$. Nous souhaitons montrer que $(s_1, v' : u) \in [\![\psi]\!]\rho$, pour chaque formule ψ dans l'ensemble

$$\{\phi_1/n',\ \phi_2/n'\} \cup \bigcup_{e \in E(n', \tau)} \{g_e \Rightarrow r_e \ \underline{\mathsf{in}}\ (\varphi/n'_e)\} \cup \bigcup_{b \in \mathsf{Act}} \ \bigcup_{e \in E(n', b)} \{g_e \Rightarrow [\overline{b}]\, (r_e \ \underline{\mathsf{in}}\ (\varphi/n'_e))\}$$

Par exemple, considérons la formule $\psi \equiv g_e \Rightarrow [\overline{b}]\, (r_e \ \underline{\mathsf{in}}\ (\varphi/n'_e))$, pour une action b et $e \in E(n', b)$. Supposons que $g_e(v')$ est vrai et que $s_1 \stackrel{\overline{b}}{\Longrightarrow}^* s_2$. Alors s et (n', v') peuvent se synchroniser sur l'action b entraînant l'exécution suivante à partir de $s \parallel (n', v')$:

$$s \parallel (n', v') \quad \stackrel{\tau}{\longrightarrow}^* \quad s_2 \parallel (n'_e, [r_e \leftarrow 0]v')$$

Comme $(s \parallel (n', v'), u) \models \phi_1 \wedge \phi_2$, il vient par la proposition 9.19 que

$$(s_2 \parallel (n'_e, [r_e \leftarrow 0]v'), u) \models \varphi$$

Par définition de l'environnement ρ, nous obtenons que $(s_2, [r_e \leftarrow 0]v' : u) \in [\![\phi/n'_e]\!]\rho$. Nous pouvons donc en conclure que

$$(s_1, v' : u) \in [\![g_e \Rightarrow [\overline{b}](r_e \ \underline{\mathsf{in}}\ (\varphi/n'_e))]\!]\rho$$

ce que nous voulions montrer. La preuve pour les autres cas pour ψ est semblable à celle que nous venons de faire, et n'est donc pas détaillée ici.

➜ **Cas** $\varphi \equiv g \vee \phi$**.** La preuve est semblable à celle que nous venons de faire pour $\varphi \equiv \phi_1 \wedge \phi_2$, et est par conséquence omise.

➜ **Cas** $\varphi \equiv [a]\phi$**.** Nous allons montrer que

$$(s, v' : u) \in \left[\!\left[\begin{array}{c} [a](\phi/n') \wedge \bigwedge_{e\in E(n',a)} (g_e \Rightarrow r_e \ \underline{\text{in}}\ \phi/n'_e) \\ \wedge \\ \bigwedge_{e\in E(n',\tau)} (g_e \Rightarrow r_e \ \underline{\text{in}}\ ([a]\phi)/n'_e) \\ \wedge \\ \bigwedge_{b\in \mathsf{Act}} \bigwedge_{e\in E(n',b)} (g_e \Rightarrow [\overline{b}]\,(r_e \ \underline{\text{in}}\ ([a]/\phi)/n'_e)) \end{array} \right]\!\right] \rho$$

À cette fin, supposons que $s \xrightarrow{\tau}{}^{*} s_1$. Nous voulons montrer que $(s_1, v' : u) \in [\![\psi]\!]\rho$, pour chaque formule ψ dans l'ensemble :

$$\begin{array}{c} \{[a](\phi/n')\} \cup \bigcup_{e\in E(n',a)} \{g_e \Rightarrow r_e \ \underline{\text{in}}\ \phi/n'_e\} \cup \bigcup_{e\in E(n',\tau)} \{g_e \Rightarrow r_e \ \underline{\text{in}}\ ([a]\phi)/n'_e\} \\ \cup \bigcup_{b\in \mathsf{Act}} \bigcup_{e\in E(n',b)} \{g_e \Rightarrow [\overline{b}]\,(r_e \ \underline{\text{in}}\ ([a]/\phi)/n'_e)\} \end{array}$$

Nous présentons les détails de la preuve pour le cas $\psi \equiv g_e \Rightarrow r_e \ \underline{\text{in}}\ \phi/n'_e$ avec $e \in E(n', a)$. Les autres cas pourraient être traités d'une façon similaire.

Supposons que $e \in E(n', a)$ et $g_e(v')$ est vrai. Alors nous avons que :

$$s \parallel (n', v') \overset{a}{\Longrightarrow} s_1 \parallel (n'_e, v'_1)$$

où $v'_1 = [r_e \leftarrow 0]v'$. Comme $(s \parallel (n', v'), u') \models \varphi$, nous obtenons que $(s_1 \parallel (n'_e, v'_1), u) \models \phi$. Par définition de l'environnement ρ, il vient que $(s_1, v'_1 : u) \in [\![\phi/n'_e]\!]\rho$. Nous en concluons que $(s_1, v' : u) \in [\![r_e \ \underline{\text{in}}\ (\phi/n'_e)]\!]\rho$, ce que nous voulions montrer.

➜ **Cas** $\varphi \equiv x \ \underline{\text{in}}\ \phi$**.** Nous voulons montrer que

$$(s, v' : u) \in \left[\!\left[\begin{array}{c} x \ \underline{\text{in}}\ (\phi/n') \\ \wedge \\ \bigwedge_{e\in E(n',\tau)} (g_e \Rightarrow r_e \ \underline{\text{in}}\ (x \ \underline{\text{in}}\ \phi)/n'_e) \\ \wedge \\ \bigwedge_{b\in \mathsf{Act}} \bigwedge_{e\in E(n',b)} (g_e \Rightarrow [\overline{b}]\,(r_e \ \underline{\text{in}}\ (x \ \underline{\text{in}}\ \phi)/n'_e)) \end{array} \right]\!\right] \rho$$

Le cas que nous allons étudier est le cas $x \ \underline{\text{in}}\ (\phi/n')$ car pour les autres, la preuve est identique aux précédentes. Supposons que $(s, v' : u) \xrightarrow{\tau}{}^{*} (s_1, v' : u)$. Alors, en se synchronisant avec le deuxième composant, nous avons que : $(s, u) \parallel ((n', v'), u) \xrightarrow{\tau}{}^{*} (s_1, u) \parallel ((n', v'), u)$. Par hypothèse, $(s_1, [x \leftarrow 0]u) \parallel ((n', v'), [x \leftarrow 0]u) \models \phi$. Alors $(s_1, v' : [x \leftarrow 0]u), \phi \in [\![\phi/n']\!]\rho$. Ainsi $(s, v' : u) \in [\![x \ \underline{\text{in}}\ (\phi/n')]\!]\rho$.

➜ **Cas** $\varphi = \forall\!\!\!\forall_S \phi$**.** Nous distinguons deux sous-cas, dépendant du fait que $S \cap \Sigma_u(n') = \varnothing$, ou pas.

– **Cas** $S \cap \Sigma_u(n') = \varnothing$. Nous montrons que si $s \overset{\epsilon(d)}{\Longrightarrow}_{S \cup \overline{\Sigma_u(n')}} s_1$ avec $d \in \mathbb{R}^+$, alors

$$(s_1, v' + d : u + d) \in \left[\!\left[\begin{array}{c} \phi/n' \wedge \bigwedge_{e\in E(n',\tau)} (g_e \Rightarrow r_e \ \underline{\text{in}}\ (\varphi/n'_e)) \\ \wedge \\ \bigwedge_{b\in \mathsf{Act}} \bigwedge_{e\in E(n',b)} (g_2 \Rightarrow [\overline{b}]\,(r_e \ \underline{\text{in}}\ (\varphi/n'_e))) \end{array} \right]\!\right] \rho$$

À cette fin, nous montrons que si $s_1 \xrightarrow{\tau}{}^{*} s_2$, alors $(s_2, v' + d : u + d) \in [\![\psi]\!]\rho$, pour chaque formule ψ apparaissant dans la conjonction du membre droit de l'équation définissant φ/n'.

Le cas qui est vraiment intéressant et différent des autres cas vus précédemment est

$$(s_2, v' + d : u + d) \in [\![\phi/n']\!]\rho$$

car les preuves pour les autres cas sont celles utilisées pour le cas $\varphi \equiv \phi_1 \wedge \phi_2$. Pour traiter ce cas, nous raisonnons de la manière suivante. Comme $S \cap \Sigma_u(n') = \varnothing$, nous avons que $(n', v') \xrightarrow{\epsilon(d)}_S (n', v' + d)$. En synchronisant cette transition d'attente de l'état (n', v') avec l'exécution

$$s \stackrel{\epsilon(d)}{\Longrightarrow}_{S \cup \overline{\Sigma_u(n')}} s_1 \xrightarrow{\tau}^{*} s_2$$

entraîne que

$$s \parallel (n', v') \quad \stackrel{\epsilon(d)}{\Longrightarrow}_S \quad s_2 \parallel (n', v' + d))$$

car la première composante est incapable de se synchroniser avec la deuxième composante sur des actions urgentes pendant l'attente. Par hypothèse, $(s_2 \parallel (n', v' + d), u + d) \models \phi$. Par définition de l'environnement ρ, nous obtenons que $(s_2, v' + d : u + d) \in [\![\phi/n']\!]\rho$, ce que nous voulions montrer.

– **Cas** $S \cap \Sigma_u(n') \neq \varnothing$. Les détails de la preuve sont identiques à ceux du cas $\varphi \equiv \phi_1 \wedge \phi_2$.

➜ **Cas** $\varphi \equiv \mathsf{max}(Z, \phi)$. Supposons que $s \xrightarrow{\tau}^{*} s_1$. Nous allons montrer que

$$(s_1, v' : u) \in \left[\!\!\left[\begin{array}{c} (\phi\{\mathsf{max}(Z,\phi)/Z\})/n' \wedge \bigwedge_{e \in E(n',\tau)} (g_e \Rightarrow r_e \ \underline{\mathsf{in}}\ (\varphi/n'_e)) \\ \wedge \\ \bigwedge_{b \in \mathsf{Act}} \ \bigwedge_{e \in E(n',b)} (g_e \ \underline{\mathsf{in}}\ [\overline{b}]\ (r_e \ \underline{\mathsf{in}}\ (\varphi/n'_e))) \end{array} \right]\!\!\right] \rho$$

À cette fin, supposons que $s_1 \xrightarrow{\tau}^{*} s_2$. Nous voulons montrer que $(s_2, v' : u) \in [\![\psi]\!]\rho$ où ψ est l'une des formules de l'ensemble

$$\{(\phi\{\mathsf{max}(Z,\phi)/Z\})/n'\} \cup \bigcup_{e \in E(n',\tau)} \{g_e \Rightarrow r_e \ \underline{\mathsf{in}}\ (\varphi/n'_e)\} \cup \bigcup_{\substack{b \in \mathsf{Act} \\ e \in E(n',b)}} \{g_e \Rightarrow [\overline{b}]\ (r_e \ \underline{\mathsf{in}}\ (\varphi/n'_e))\}$$

Considérons la formule $\psi \equiv (\phi\{\mathsf{max}(Z,\phi)/Z\})/n'$. Comme $s \xrightarrow{\tau}^{*} s_2$, il vient que

$$s \parallel (n', v') \xrightarrow{\tau}^{*} s_2 \parallel (n', v')$$

Par définition de la relation de satisfaction pour φ, nous avons que

$$(s_2 \parallel (n', v'), u) \models \phi\{\mathsf{max}(Z,\phi)/Z\}$$

Par définition de l'environnement ρ, nous obtenons que

$$(s_2, v' : u) \in [\![(\phi\{\mathsf{max}(Z,\phi)/Z\})/n']\!]\rho$$

Par conséquence, $(s_2, v' : u) \in [\![(\phi\{\mathsf{max}(Z,\phi)/Z\})/n']\!]\rho$, c'est ce que nous souhaitions obtenir.

Pour tous les autres cas de ψ, la preuve est similaire à celle pour $\varphi \equiv \phi_1 \wedge \phi_2$ à cause de la proposition 9.19.

La preuve de l'implication « Seulement si » est maintenant complète.
La preuve du théorème 9.36 est maintenant terminée. □

Chapitre 10

Une étude de cas : le protocole PGM

Ce chapitre est paru dans un rapport du projet RNRT Calife (sous-projet 4, « Techniques d'abstraction ») [BBP01].

Le travail que nous présentons dans ce chapitre est de nature complémentaire par rapport aux travaux effectués jusqu'à présent. En effet, dans ces derniers, nous avons proposé des modèles et des algorithmes pour faire du model-checking, mais nous avons toujours supposé que le modèle du système réel était créé. Cependant, en pratique, comme nous l'avons dit dans le chapitre introductif 1, les systèmes que l'on cherche à vérifier doivent être modélisés avant de pouvoir être vérifiés. Dans ce cadre, nous présentons ici une modélisation d'un protocole de communication, le protocole PGM (ce qui signifie *Pragmatic Multicast Protocol*), que nous avons faite dans le cadre du projet RNRT Calife[1]. Nous présentons ensuite les expériences conduites avec l'outil UPPAAL.

Nous avons décidé de modéliser ce protocole de manière à pouvoir l'implémenter dans l'outil UPPAAL (voir la partie 7.3.4 et le chapitre 9), nous justifierons ce choix dans la partie 10.2. La présentation de ce travail va se découper en plusieurs parties : nous commençons par décrire le protocole PGM en expliquant quelles sont les simplifications et hypothèses que nous allons faire dans notre modélisation. Nous décrivons aussi quelles sont les propriétés que nous allons vérifier. Ensuite, nous proposons notre modélisation de PGM en décrivant précisément comment nous modélisons les différents composants du protocole. Nous décrivons ensuite quelles sont les simulations et les vérifications que nous avons faites avec l'outil UPPAAL et nous présentons nos résultats. Nous terminerons par quelques remarques ainsi qu'un bilan de ce travail.

10.1 Description informelle du protocole

Ce protocole est commercialisé, nous pouvons par exemple citer l'adresse

`http ://www.cisco.com/warp/public/cc/pd/iosw/tech/_fr_xcst_ds.htm`

où le protocole PGM est présenté de la manière suivante :

[1]Le but du projet Calife est de construire un prototype pour un outil de vérification, alliant à la fois preuve formelle, model-checking et génération de test pour les protocoles de communication. La page web du projet se trouve à l'adresse suivante :

`http ://www.loria.fr/projets/calife`

«

- *Protocole de transport multipoint fiable pour les applications exigeant une livraison ordonnée, sans doublons et multipoint des données à partir de plusieurs sources et vers plusieurs destinataires*
- *Assure que le destinataire d'un groupe multipoint reçoit soit tous les paquets de données transmis et retransmis, ou qu'il peut détecter la perte irrémédiable de paquets*
- *Conçu pour les applications multipoint avec des exigences de base en matière de fiabilité*

»

Plusieurs autres descriptions de PGM peuvent être trouvées sur Internet.

En ce qui nous concerne, la spécification de PGM nous a été fournie sous la forme d'un cahier des charges de 115 pages [Spe01] qui décrit avec précisions le protocole. Certaines propriétés que doit vérifier ce protocole sont aussi indiquées.

10.1.1 Ce que fait PGM

PGM est un protocole de communication multicast. La topologie d'un réseau dans PGM est un graphe avec des nœuds qui sont soit des sources, soit des nœuds de réseau, soit des destinataires.

La source cherche à envoyer des données (supposées ordonnées) aux différents destinataires *via* le réseau supposé non fiable. Certaines données peuvent être perdues, il faut donc mettre en place une stratégie de recouvrement de ces données.

Les données envoyées par la source sont appelées des ODATA, pour Original DATA. La source envoie ces ODATA aux destinataires en suivant l'arborescence du réseau. Certaines données peuvent être perdues lors de leur transmission au niveau des nœuds de réseau.

Lorsque la source cesse d'envoyer des données pendant un certain temps, elle adresse aux destinataires des informations sur sa fenêtre de transmission, c'est-à-dire sur l'ensemble des données qu'elle a en mémoire et qu'elle est donc capable de renvoyer en cas de perte. Ces informations sont appelées des SPM, pour Source Path Messages. Elles permettent aux destinataires d'être en permanence au courant de l'état de la source.

La détection d'une perte de donnée se fait au niveau des destinataires soit à la réception d'une donnée, soit à la réception d'un SPM. Un destinataire peut détecter une perte de données grâce au fait que les données sont ordonnées : s'il reçoit une donnée dont le numéro est strictement supérieur à celui des données précédemment reçues, cela signifie qu'une ou plusieurs données intermédiaires ont été perdues ; s'il reçoit un SPM lui indiquant que la fenêtre de transmission de la source a avancé alors qu'il n'a pas reçu les données correspondantes, alors cela signifie qu'une ou plusieurs données ont été perdues.

Lorsque le destinataire détecte une perte de donnée, il envoie à son père dans le réseau un NAK (Negative AcKnowledgement) qui le transmet à son propre père, etc... jusqu'à la source. Ceci est réalisé grâce à la création au niveau de chaque nœud de réseau d'un « état de réparation ». Cette structure permet de garder en mémoire quels sont les RDATA en attente, de temporiser l'envoi des NAK et de recevoir les NCF (Negative ConFirmation) que le père envoie pour indiquer qu'il a bien reçu la demande de réparation.

À la réception d'un NAK, la source, si elle le peut, renvoie aux destinataires la donnée correspondante (elle est alors dite réparée, c'est ce que nous appelons une RDATA, comme Repair DATA). La source ne peut pas toujours envoyer des réparations car elle possède une fenêtre de transmission qui contient une quantité bornée de données (une mémoire finie), donc si la demande de réparation d'une donnée arrive trop tard, il se peut que la source n'ait plus ladite donnée en mémoire.

Spécificité de PGM. Le destinataire n'envoie un « accusé de réception » qu'en cas de perte (envoi des NAK), il n'envoie pas d'accusé de réception positif pour dire « j'ai bien reçu telle donnée », ce qui évite l'explosion du nombre d'accusés de réception qui se propagent dans le réseau.

Nous n'allons pas modéliser exactement le protocole PGM tel que décrit précédemment, mais nous allons faire au préalable quelques hypothèses simplificatrices.

10.1.2 Hypothèses simplificatrices

Nous faisons principalement deux hypothèse simplifatrices, la première sur la topologie du réseau et la seconde sur les pertes au niveau du réseau.

Topologie du réseau. Nous ne prenons en compte qu'une seule source. La topologie du réseau est invariante pendant une session. Le réseau a une forme d'arbre avec trois types de nœuds :

- la racine de l'arbre, appelée la *source*,
- les feuilles de l'arbre, appelées les *destinataires*,
- tous les autres nœuds de l'arbre, appelés les *nœuds de réseau*.

Pertes. Seules les (O/R)DATA peuvent être perdues, pas les NAK. Ceci entraîne en particulier les simplifications suivantes :

- Les nœuds de réseau et la source n'envoient pas d'accusé de réception des NAK (appelés NCF comme NAK ConFirmation), ceci n'étant pas nécessaire car les NAK sont supposés ne pas se perdre dans le réseau.
- Seul l'envoi d'un NAK est nécessaire lors de la détection d'une perte de données, il n'y a pas besoin d'en envoyer à une fréquence régulière.
- Pas de création d'états de réparation au niveau des nœuds de réseau.

10.1.3 Rôle des différents composants

Le protocole (simplifié) peut alors être schématisé comme sur la figure 10.1.

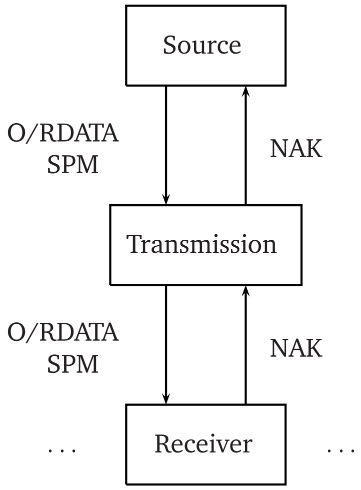


Figure 10.1: Schéma décrivant le protocole PGM

Nous allons plus précisément décrire le rôle des différents composants du réseau.

La source.

La source possède une fenêtre de transmission (TXW pour Transmit Window) qui garde en mémoire les numéros de toutes les données qu'elle conserve dans cette fenêtre et qu'elle peut donc encore envoyer. Cette fenêtre de transmission se déplace au fur et à mesure de l'envoi des données. La stratégie de déplacement de la fenêtre de transmission n'est pas fixée par le protocole.
La source envoie (avec une fréquence qui ne dépasse pas une certaine fréquence maximale) des données originales (ODATA) aux destinataires *via* le réseau. Juste après avoir envoyé une ODATA, elle envoie sur le réseau les informations sur sa fenêtre de transmission (SPM). Lorsque l'envoi de données est en veille, elle envoie seulement les informations sur sa fenêtre de transmission (SPM). Elle peut aussi recevoir de ses fils des accusés de réception négatifs (NAK), ce qui signifie que l'un au moins des destinataires a perdu la donnée correspondant à ce NAK. À la réception d'un NAK, si la donnée correspondante est toujours dans sa fenêtre de transmission, la source retient qu'elle devra envoyer une donnée réparée (RDATA) pour la donnée correspondant au NAK reçu. La source envoie indifféremment des ODATA ou des RDATA. Avant de modifier sa fenêtre de transmission, la source envoie toutes les RDATA en attente pour éviter au maximum des pertes de données.

Un destinataire.

Un destinataire possède une fenêtre de réception qui représente l'ensemble des données dont le statut n'est pas fixé (données détectées comme perdues mais pas encore recouvrées par exemple ou bien données d'indice plus grand que celui d'une donnée perdue). Un destinataire a deux rôles principaux : d'une part il reçoit des (O/R)DATA et des SPM et d'autre part il envoie des NAK lorsqu'il détecte une perte de données.
Une perte de donnée est détectée dans les deux cas suivants :
– réception d'une donnée d'indice i telle qu'au moins une donnée d'indice $j < i$ n'a pas été reçue,
– réception d'un SPM indiquant que la borne supérieure de la fenêtre de transmission est strictement supérieure à la plus grande donnée reçue.

Un nœud de réseau.

Il retransmet les (O/R)DATA de la racine vers les feuilles en pouvant perdre occasionnellement certaines des données (la fréquence de perte n'est pas précisée). Il retransmet aussi les NAK de bas en haut sans perte d'information.

10.1.4 Les propriétés à vérifier

Nous nous sommes intéressés aux deux propriétés fondamentales suivantes :

1. Pour toute donnée, le destinataire sait si elle est reçue ou s'il ne la recevra jamais (c'est la propriété qui est demandée dans [Spe01]). Nous appellerons cette propriété **Info-perte**.
2. Chaque donnée détectée comme étant perdue est recouvrée. C'est une propriété plus forte et plus intéressante que la précédente. On appellera cette propriété **Pas-de-perte**.

10.2 La modélisation du protocole PGM

Une modélisation de PGM reflétant fidèlement la description ci-dessus nécessiterait l'utilisation d'un grand nombre d'objets non bornés. Or ces structures de données rendent la vérification très difficile, voire indécidable. Nous allons donc abstraire le protocole en utilisant des structures bornées, pour obtenir un modèle implémentable dans UPPAAL (voir la partie 7.3.4 pour une description des algorithmes de UPPAAL et des références et voir le chapitre 9 pour une description précise du modèle « à la UPPAAL »).

Comme les NAK ne sont pas perdus, nous pouvons supposer qu'ils ne passent pas dans les nœuds de réseau mais qu'ils se transmettent directement du destinataire à la source. Nous schématisons cette simplification comme à la figure 10.2.

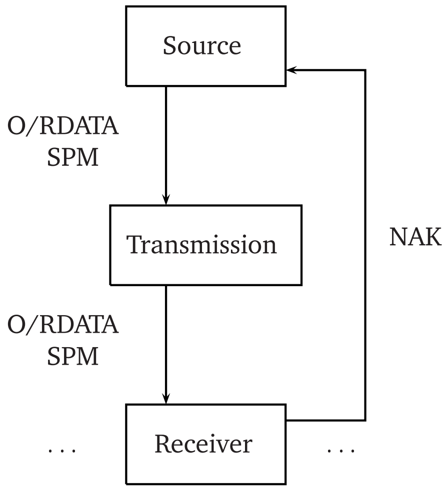
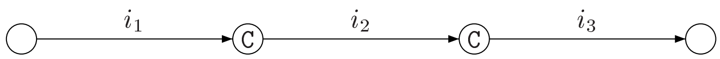


Figure 10.2: Schéma de la modélisation de PGM

Pour modéliser une séquence d'instructions qui doivent se faire de manière instantanée, nous utiliserons les *Committed Locations* d'UPPAAL. Si un état d'un automate est « committed », ce qui est noté par `C` sur les dessins, ceci signifie que lors d'une exécution, il faut immédiatement quitter cet état. Par exemple, supposons que nous souhaitions effectuer les instruction i_1, i_2 et i_3 de manière séquentielle mais instantanée. Une solution est de représenter ceci par un automate comme celui qui suit :



Si l'instruction i_1 est exécutée, alors tout de suite après, les instructions i_2 puis i_3 seront aussi exécutées, indépendamment de ce que font les autres automates de la composition parallèle.

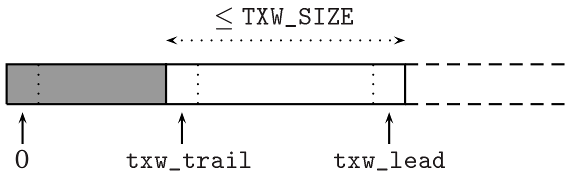
UPPAAL ne permet pas la manipulation de files de communication entre les différents automates. Pour pouvoir représenter les files de communication, nous utilisons les canaux de communication proposés dans UPPAAL et nous utilisons des variables partagées pour communiquer les informations de composant en composant.

Nous utilisons aussi des variables entières bornées et des tableaux. Les variables entières (bornées) vont nous servir à stocker ou/et à transmettre toutes les informations sur ce qui circule dans le réseau (données, accusés de réception). Les tableaux vont nous servir essentiellement à abstraire les files non bornées dont nous aurions besoin pour représenter les files de communication et les fenêtres de réception.

Nous utilisons aussi bien entendu les horloges pour temporiser les envois et les transmissions de données sur le réseau, elles vont être principalement utilisées au niveau des nœuds de réseau.

10.2.1 Modélisation de la source

Nous modélisons la fenêtre de transmission par deux variables, `txw_trail` et `txw_lead`, qui correspondent respectivement au numéro de la plus petite donnée qui se trouve dans la fenêtre de transmission et à la plus grande donnée qui se trouve dans cette même fenêtre de transmission.
La stratégie de déplacement de la fenêtre de transmission que l'on choisit est la suivante : elle a une taille maximale, qui correspond au paramètre `TXW_SIZE`, et lorsqu'elle est pleine, nous envoyons toutes les RDATA qui sont en attente et nous déplaçons la fenêtre de transmission (en la translatant). L'état de cette fenêtre peut être représenté par le schéma suivant :



A chaque instant, nous avons `txw_lead` − `txw_trail` ≤ `TXW_SIZE`. Nous décalons la fenêtre de transmission de `TXW_ADVANCE` lorsque `txw_lead` − `txw_trail` = `TXW_SIZE` et lorsque nous souhaitons envoyer une nouvelle ODATA.

La source communique avec les nœuds du réseau qui sont juste en dessous *via* les canaux de synchronisation `src_send_data` (envoi d'une (O/R)DATA) et `src_send_spm` (envoi d'un SPM) et grâce aux variables partagées `src_sqn` (numéro de la donnée envoyée), `src_trail` (borne inférieure de la fenêtre de transmission) et `src_lead` (borne supérieure de la fenêtre de transmission). Il n'y a pas de priorité entre l'envoi d'une ODATA ou d'une RDATA sauf lorsque la fenêtre de transmission est pleine, auquel cas l'envoi des RDATA est prioritaire sur l'envoi d'une nouvelle ODATA.

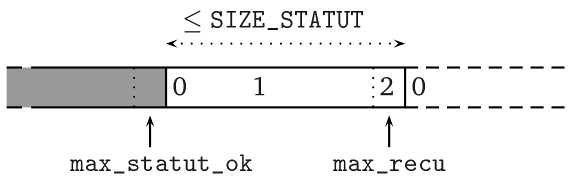
La source communique aussi avec les destinataires *via* le canal `src_receive_nak` (réception d'un NAK) grâce à la variable partagée `nak` (numéro du NAK). À la réception d'un NAK, la source le garde en mémoire dans le tableau `rdata`, ce qui lui permet de savoir quelles sont les données qu'elle devra envoyer en réparation ultérieurement (RDATA).

Notre modélisation de la source est représentée sur la figure 10.3 à la page 230.

10.2.2 Modélisation d'un destinataire

Nous représentons la fenêtre de réception par un tableau circulaire `statut` qui a une taille bornée (par le paramètre `SIZE_STATUT`). Deux numéros de données sont retenus : `max_recu` qui représente la plus grande donnée reçue et `max_statut_ok` qui représente la plus grande donnée pour laquelle nous sommes sûrs de l'état (reçue, perdue). Une case du tableau `statut` est dans l'un des trois états :

- état 0 si la case n'est pas occupée,
- état 1 s'il va falloir envoyer un NAK,
- état 2 si la donnée correspondante a bien été reçue.



Les deux indices `index_max_statut_ok` et `index_max_recu` permettent de savoir à quelle case du tableau correspondent les données `max_statut_ok` et `max_recu`.

À la réception d'une donnée `d`, que devons-nous faire ?

1. si $\mathtt{d} \in]\mathtt{max\_statut\_ok}; \mathtt{max\_statut\_ok} + \mathtt{SIZE\_STATUT}]$, il faut indiquer dans le tableau `statut` que la donnée `d` est reçue. Si les premières cases de statut sont dans l'état 2, il faut décaler le pointeur `max_statut_ok` (ainsi que `index_max_statut_ok`, évidemment).
2. si $\mathtt{d} \leq \mathtt{max\_statut\_ok}$, il n'y a rien à faire.
3. si $\mathtt{d} > \mathtt{max\_statut\_ok} + \mathtt{SIZE\_STATUT}$, nous décalons la fenêtre `statut` pour que `d` soit la plus grande donnée retenue par `statut`. Nous mettons à jour `max_recu` et il faut décaler `max_statut_ok` pour redimensionner à la taille `SIZE_STATUT`. C'est là que nous voyons si certaines données sont définitivement perdues.

À la réception d'un SPM `(rec_trail,rec_lead)`, que faire ?

1. si $\mathtt{rec\_lead} > \mathtt{max\_recu}$, alors, comme dans le cas 3. précédent, il faut décaler la fenêtre `statut`.
2. si $\mathtt{rec\_trail} > \mathtt{max\_statut\_ok+1}$, alors il faut décaler le `max_statut_ok` car les données qui sont inférieures au `rec_trail` ne peuvent plus être recouvrées et sont donc définitivement perdues.

Un destinataire communique avec son nœud père *via* les canaux `rec_receive_data` (réception d'une donnée) et `rec_receive_spm` (réception d'un SPM) grâce aux variables `rec_sqn` (numéro de la donnée qui arrive), `rec_trail` (borne inférieure de la fenêtre de transmission de la source) et `rec_lead` (borne supérieure de la fenêtre de transmission de la source).

Comme nous l'avons déjà vu au niveau de la modélisation de la source, les destinataires communiquent avec la source *via* le canal `src_receive_nak` grâce à la variable partagée `nak`. Ils doivent aussi être capables de détecter les pertes de données éventuelles et envoyer des NAK si une donnée perdue est détectée.

La partie « réception de données et de SPM » d'un destinataire peut se modéliser par l'automate de la figure 10.4 à la page 231. La partie « envoi de NAK » peut se modéliser par l'automate de la figure 10.5 à la page 232.

Un destinataire résulte donc de la composition parallèle des deux automates précédents. Ces deux automates ne communiquent en fait pas et agissent de manière indépendante.

10.2.3 Modélisation d'un nœud de réseau

Un nœud retransmet les envois de (O/R)DATA et de SPM. Il temporise aussi les envois de données (il y a un paramètre, `MAX_DATA_DELAI`, qui « impose » un délai de transmission pour les données et un paramètre, `SPM_DELAI`, qui « impose » un délai de transmission pour les SPM). C'est aussi à ce niveau que nous imposons l'envoi d'un SPM après chaque envoi de (O/R)DATA. La période d'envoi des SPM est entre `MIN_PERIODE_SPM` et `MAX_PERIODE_SPM`.
C'est au niveau d'un nœud de réseau que les pertes de données sont modélisées. Ici, la stratégie de perte de données borne le nombre maximal de données perdues « en même temps » par `MAX_NB_ERROR`. En effet, l'utilisation de structures bornées (tableaux de taille fixée) pour la source et le destinataire impose une borne au nombre d'erreurs. Par contre, l'envoi des NAK court-circuite ce module de transmission. L'automate de la figure 10.6 de la page 235 est le modèle d'un nœud de réseau lorsque nous avons deux destinataires. Si nous voulons rajouter des destinataires, il suffit de rajouter quelques transitions pour « brancher » les destinataires supplémentaires.

Cette partie de l'automate correspond à l'envoi des SPM.

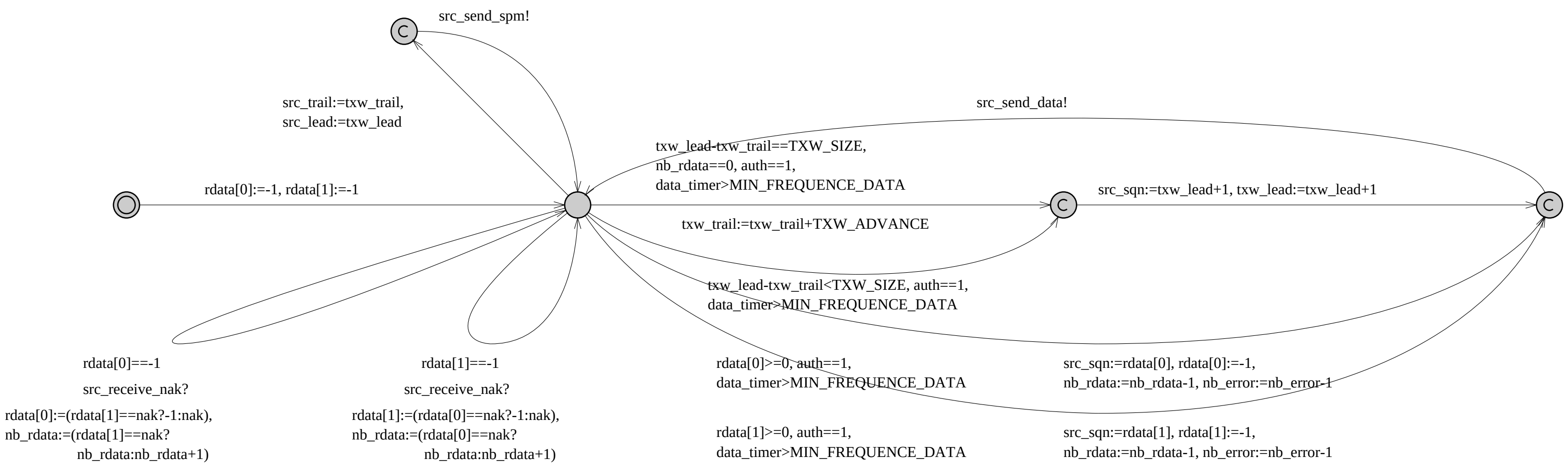


Figure 10.3: Modélisation de la source

Cette partie de l'automate correspond au stockage des RDATA en attente dans le tableau `rdata`.

Cette partie de l'automate sert à envoyer les ODATA et les RDATA. Les deux transitions du bas correspond à l'envoi des RDATA et les deux transitions centrales à l'envoi des ODATA (avec gestion de la fenêtre de transmission).

Notons que la macro `toto:=test?toto1:toto2` signifie que si `test` est vrai, alors la valeur de `toto1` est placée dans `toto`, alors que sinon, c'est la valeur de `toto2` qui est mise dans `toto`.

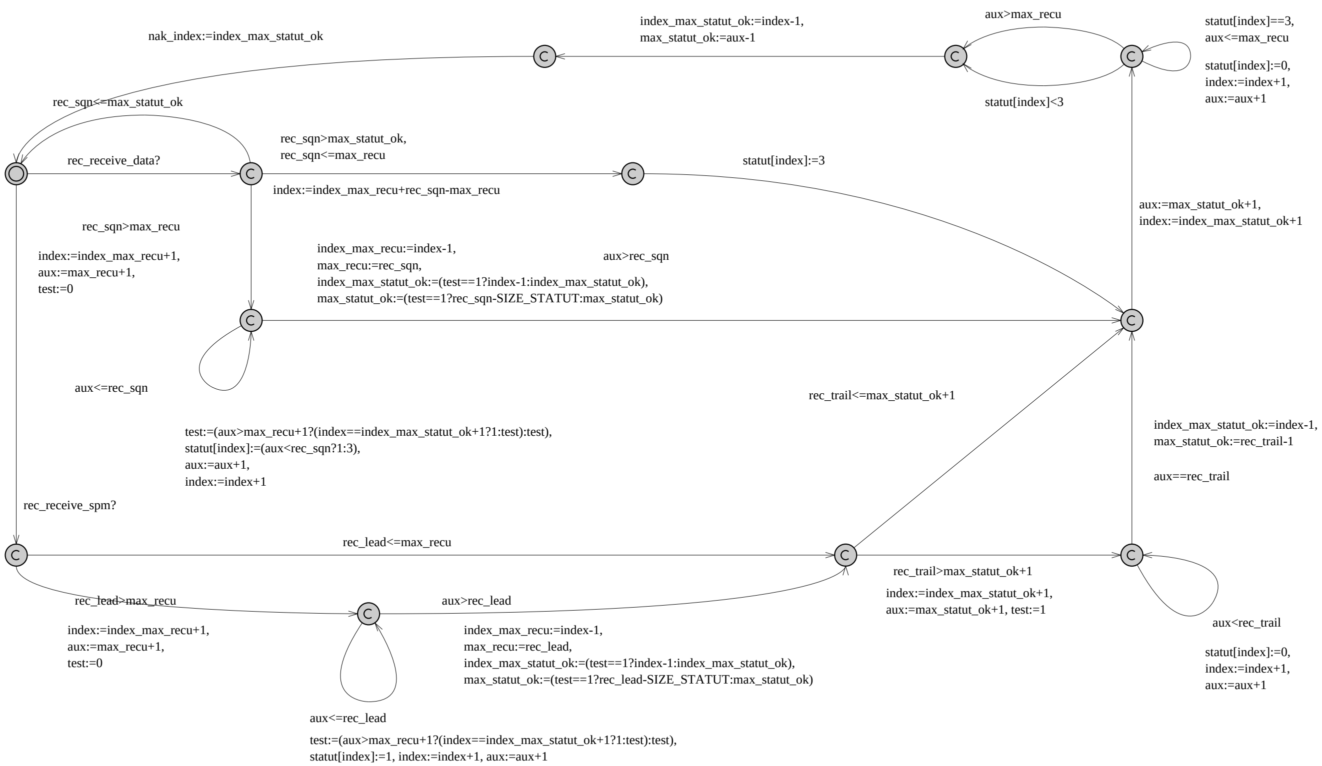


Figure 10.4: Modélisation d'un destinataire (1)

La partie du bas de cet automate correspond à la mise à jour de la fenêtre de réception lorsqu'un SPM est reçu. La partie centrale de l'automate correspond à la mise à jour de la fenêtre de réception lorsqu'une donnée (O/RDATA) est reçue. La partie du haut de l'automate correspond à la fin de la mise à jour de la fenêtre de transmission, cette partie est commune aux deux cas (arrivée d'un SPM ou d'une donnée).

La variable `test` est là pour tester si une donnée n'a pas été reçue alors que sa case dans le tableau disparaît à cause du décalage (plus tard, nous dirons alors que dans ce cas, la donnée est définitivement perdue).

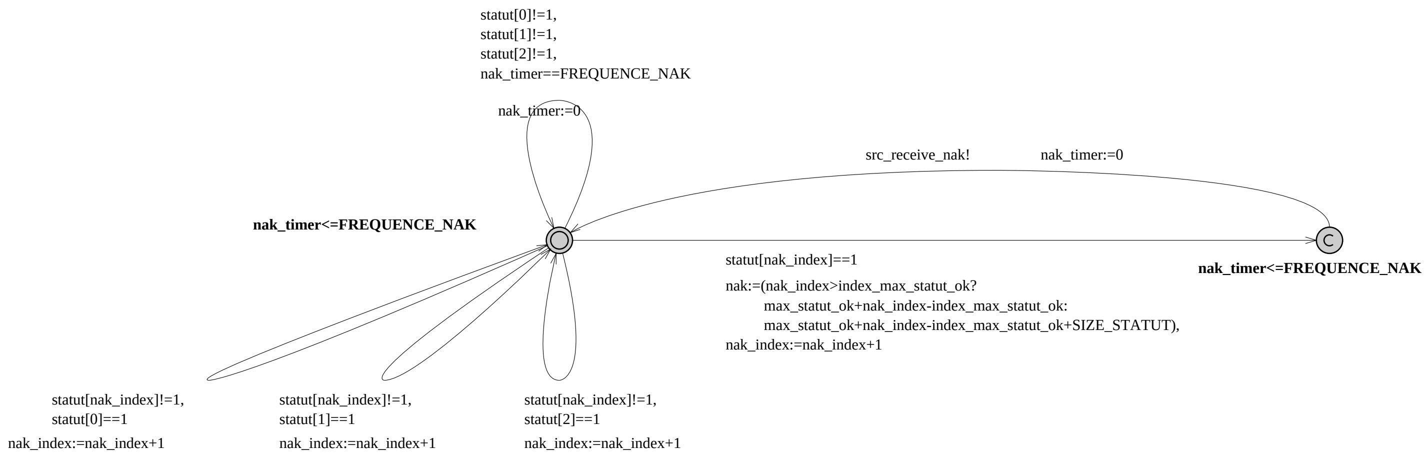


Figure 10.5: Modélisation d'un destinataire (2)

Cette partie régule l'envoi des NAK par le destinataire. La boucle du haut règle le cas où aucune perte n'a été détectée. Les boucles du bas servent à se déplacer dans le tableau `statut` à la recherche d'un NAK à envoyer.

Cette partie de l'automate sert à envoyer les NAK au moins toutes les `FREQUENCE_NAK` unités de temps.

Le nœud de réseau se synchronise d'une part avec la source *via* les canaux `src_send_data` et `src_send_spm` et d'autre part avec les destinataires *via* `rec_receive_data` et `rec_receive_spm`.

10.3 Simulation et vérification

Nous avons implémenté ce modèle dans l'outil UPPAAL (UPPAAL2k 3.2.4 plus précisément). Les destinataires ont été implémentés comme des « templates » dans UPPAAL. Chaque destinataire est donc une instanciation de ce template.
Nous allons voir que les deux propriétés que nous voulons tester s'expriment toutes les deux aisément par une formule d'accessibilité, elles sont donc exprimables dans le langage de model-checking d'UPPAAL.

Dans la partie 7.1.2, nous avons vu qu'il était possible d'exprimer des propriétés d'accessibilité grâce à des formules du genre EFφ qui signifie qu'il existe un chemin qui mène à un état où φ est vrai. En UPPAAL, il est possible d'écrire ce genre de propriété dès que φ ne contient pas d'opérateurs temporels (il n'est par contre pas possible d'imbriquer les opérateurs temporels comme AGEFψ). Par exemple, si `Node` est un état de l'automate `Source`, alors la formule EF`Source.Node` signifie que nous pouvons atteindre l'état `Node` de l'automate `Source`. Nous pouvons aussi tester l'accessibilité d'un état avec des précisions sur la valeur des horloges. Par exemple, pour tester que nous pouvons arriver dans l'état `Node` de `Source` avec une valeur de `x` inférieure à 3, nous pouvons écrire EF`(Source.Node and x<3)`.

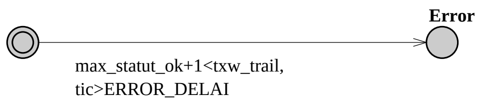
10.3.1 Les paramètres des simulations

Nous avons effectué plusieurs expériences en affectant différentes valeurs aux paramètres cités ci-avant. Nous les récapitulons dans le tableau 10.7. Les paramètres globaux du protocole ainsi que les paramètres de taille seront fixés pour toutes les expériences. Par contre, les paramètres de temps vont changer de valeurs lors des différentes expériences, c'est pour cette raison que nous avons mis un « ? » dans les cases qui leur correspondent.

Les deux parties qui suivent récapitulent les expériences effectuées en précisant les valeurs des paramètres, les options d'UPPAAL utilisées, le temps et la mémoire pris par la vérification (en utilisant une station Sun Ultra 220R, 2x450 MHz CPU, 2048 MB RAM sous Solaris 8).
Juste après avoir exposé les résultats, nous discutons des résultats obtenus.

10.3.2 Vérification de la propriété « Info-perte »

La propriété **Info-perte** se teste grâce à un « observateur extérieur » qui regarde si la fenêtre du destinataire (`statut`) correspond à la fenêtre de transmission ou bien si elle est en décalage.
L'automate que nous ajoutons est très simple :



L'horloge `tic` est une horloge qui est remise à zéro à chaque fois que le destinataire remet son tableau `statut` à jour. La valeur du paramètre `ERROR_DELAI` n'est pas fixée en avance, c'est une valeur que nous allons faire varier lors de nos expérimentations.

Paramètres globaux du protocole		
Nombre de destinataires		2
Nombre de nœuds de réseau		1
Paramètres de taille		
Nombre de données que la source peut envoyer	MAX_NB_DATA	10
Nombre de RDATA qui peuvent être en attente	MAX_NB_RDATA	2
Nombre d'erreurs que nous pouvons gérer en même temps	MAX_NB_ERROR	2
Taille de la fenêtre de réception	SIZE_STATUT	3
Taille de la fenêtre de transmission	TXW_SIZE	5
Décalage de la fenêtre de transmission	TXW_ADVANCE	2
Paramètres de temps		
Délai de transmission des SPM	SPM_DELAI	?
Période inf d'envoi des SPM	MIN_PERIODE_SPM	?
Période sup d'envoi des SPM	MAX_PERIODE_SPM	?
Borne inf du délai de transmission des données	MIN_DATA_DELAI	?
Borne sup du délai de transmission des données	MAX_DATA_DELAI	?
Période d'envoi des données	MIN_PERIODE_DATA	?
Période d'envoi des NAK	PERIODE_NAK	?

Tableau 10.7: Paramètres des expériences

Si `obs` est le nom de cet automate, la propriété s'exprime alors par la formule :

`EF(obs.Error)`

En effet, la propriété « une donnée est définitivement perdue mais le destinataire ne le sait pas » s'exprime exactement par le fait que `max_statut_ok+1 < txw_trail` avec la propriété temporisée que la valeur de l'horloge `tic` est supérieure à la valeur du paramètre `ERROR_DELAI`.

Le tableau 10.8 résume les expériences faites pour cette propriété (pour cause de problèmes de mémoire, les expériences faites se restreignent à un seul destinataire).

Le seul paramètre que nous faisons varier ici est le paramètre `ERROR_DELAI`. Il correspond au temps laissé au destinataire pour mettre sa fenêtre à jour. Les expériences effectuées laissent apparaître une valeur de seuil, 9, au-delà de laquelle la fenêtre de réception est en accord avec la fenêtre de transmission. En effet, avoir le résultat « Vrai » à la vérification signifie qu'à un moment donné, la connaissance du destinataire ne correspond pas à l'état de la fenêtre de transmission, même après une attente de `ERROR_DELAI` unités de temps. Par contre, quand le résultat de la vérification est « Faux », cela signifie que la connaissance de la fenêtre est à jour par rapport à ce que valait la fenêtre de transmission `ERROR_DELAI` unités de temps auparavant.

Remarque : En regardant le tableau 10.8, deux points doivent être soulignés. Tout d'abord, plus le paramètre `ERROR_DELAI` croît vers la valeur « critique » 9, plus la vérification prend du temps et plus il y a besoin de mémoire. Uppaal permet d'utiliser l'approximation par enveloppe convexe pour accélérer les calculs de son analyse en avant (voir la partie 7.2.1). Cependant, cette approximation calcule un ensemble d'états strictement plus grand que l'ensemble des états vraiment accessibles. La réponse du model-checker n'est alors *a priori* correcte que dans le cas où l'état testé n'est pas atteignable. C'est pour cette raison que nous ne l'avons utilisée qu'au delà de la « valeur critique » 9. Pour cette valeur, nous avons réalisé les calculs sans approximation et avec l'approximation par enveloppe convexe. Nous constatons que les calculs ont vraiment été plus efficaces dans le second cas.

Cette partie de l'automate sert à transmettre les données de la source aux destinataires. La boucle de droite modélise une perte de données (dans la limite de `MAX_NB_ERROR` en même temps).

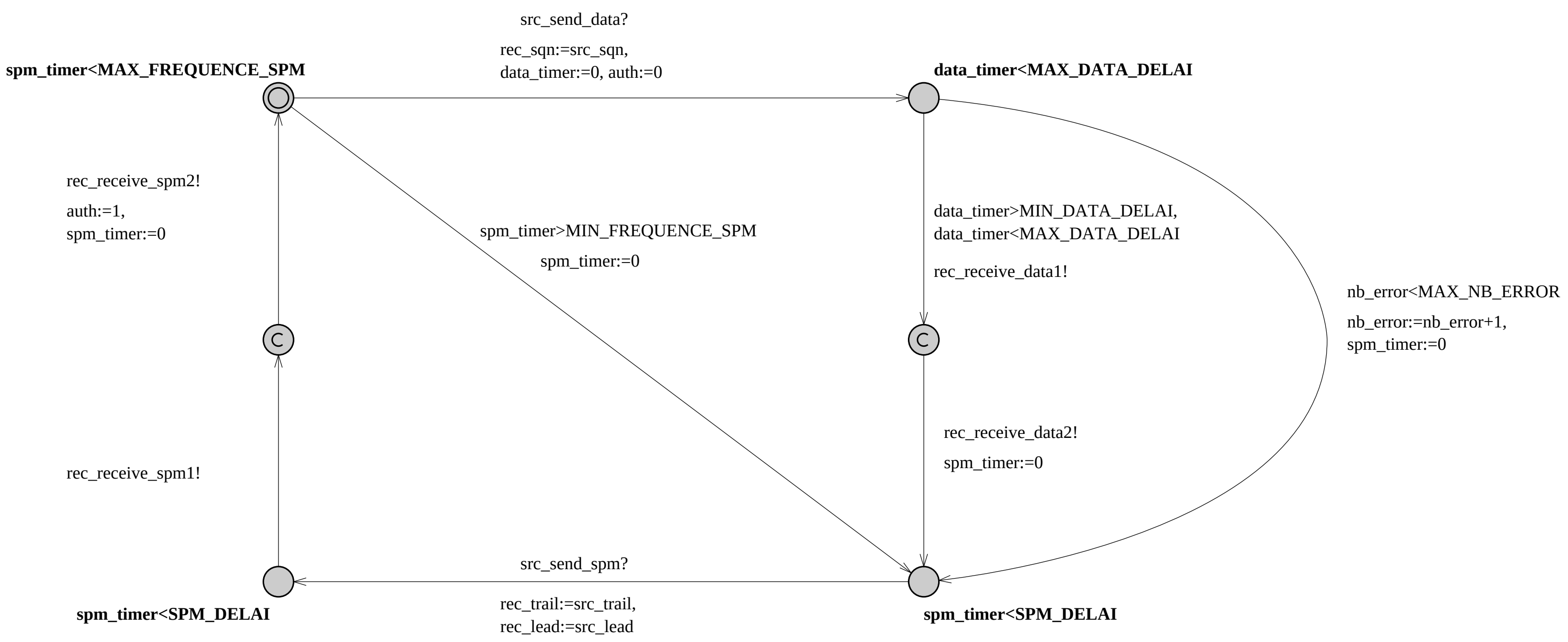


Figure 10.6: Modélisation d'un nœud de réseau

Cette partie de l'automate sert à transmettre les SPMs de la source aux destinataires. La transition diagonale centrale sert à court-circuiter l'envoi de données et à envoyer uniquement des SPMs.

C'est à ce niveau que les envois de données, de SPMs sont temporisés (durée de transfert sur le réseau, fréquence). Quasiment tous les paramètres de temps apparaissent dans cet automate.

Il ressort donc des expériences que le destinataire s'aperçoit toujours de la perte de données, mais avec un peu de retard sur le déplacement de la fenêtre de transmission. La valeur 9 correspond au temps de réaction du réseau, c'est à dire au temps qu'il faut pour que toute l'information sur la fenêtre de transmission soit passée de la source aux destinataires.

10.3.3 Vérification de la propriété « Pas-de-perte »

La propriété **Pas-de-perte** est plus contraignante, puisqu'elle vise à certifier que tout destinataire reçoit toutes les données, sans aucune perte. Pour étudier cette propriété, nous retirons l'automate `obs` (qui sert seulement à tester la propriété **Info-perte**), ainsi que le paramètre `ERROR_DELAI`. La propriété **Pas-de-perte** peut s'exprimer par la formule

```
EF(receiver1.test==1)
```

dans le cas d'un seul destinataire et par

```
EF(receiver1.test==1 or receiver2.test==1)
```

dans le cas de deux destinataires.
En effet, la variable `test` de l'automate `receiver` devient égale à 1 si, lors de la mise à jour du tableau `statut`, nous devons incrémenter `max_statut_ok` alors que la donnée qui manquait et qui correspondait à `max_statut_ok + 1` n'a pas été reçue. Nous constatons donc que la formule précédente correspond bien à la propriété souhaitée sauf pour les 5 dernières données : 5 représente ici la taille du tableau `statut`, c'est à dire la valeur du paramètre `SIZE_STATUT`. Nous ferons une remarque sur ce point juste après avoir présenté les résultats expérimentaux.

Nous effectuons des vérifications dans le cas d'un seul destinataire et dans le cas de deux destinataires. Le récapitulatif des résultats se trouve dans les deux tableaux 10.9 et 10.10.

Tous ces résultats font apparaître deux problèmes, liés aux paramètres du protocole, mais indépendants de notre modélisation.

- En analysant les deux tableaux 10.9 et 10.10, nous voyons que si l'envoi de NAK n'est pas assez rapide (paramètre `PERIODE_NAK`) par rapport à l'envoi des données au niveau de la source (paramètre `MIN_PERIODE_DATA`), la propriété **Pas-de-perte** n'est plus vérifiée, alors que si l'envoi des NAK est plus rapide, cette propriété est vérifiée (résultats des deux tableaux précédents). Donc, si la source peut envoyer ses données à une fréquence trop importante, il peut arriver qu'elle se méprenne sur ce que connaissent les destinataires. Imaginons qu'elle envoie une RDATA, celle-ci est donc enlevée du tableau `rdata`. Si cette RDATA est à nouveau perdue, la source a pu décaler, entre temps, sa fenêtre de transmission, pensant que les destinataires avaient reçu cette RDATA. Le NAK correspondant à cette perte de RDATA va arriver trop tard. Il est donc important que la fréquence d'envoi des données ne soit pas trop grande (par rapport à la vitesse d'envoi des NAK par les destinataires).
- En cherchant à modéliser la propriété **Pas-de-perte**, nous avons constaté un petit problème au niveau de la source. Les RDATA peuvent rester en attente trop longtemps au niveau de la source : par exemple, lorsque la source cesse d'envoyer des ODATA, rien ne l'oblige à envoyer les dernières RDATA qui sont dans `rdata`), donc le destinataire peut ne jamais recevoir les données réparées qu'il a demandées. Il faudrait alors « forcer » la source à envoyer les RDATA. C'est pour cette raison que la formule `EF(receiver.test==1)` teste seulement de façon partielle la propriété que nous souhaiterions voir vérifiée par le système : il se peut que les dernières données demandées en réparation ne soient jamais envoyées aux destinataires car la source n'a pas besoin de décaler sa fenêtre de transmission (s'il n'y a plus d'envoi de ODATA par exemple).

Les paramètres de temps autres que `ERROR_DELAI` sont fixés pour toutes les expériences. Voilà leurs valeurs :

SPM_DELAI	1
MIN_DATA_DELAI	2
MAX_DATA_DELAI	3
MIN_PERIODE_SPM	3
MAX_PERIODE_SPM	5
MIN_PERIODE_DATA	1
PERIODE_NAK	2

Nos résultats sont alors présentés dans le tableau suivant :

ERROR_DELAI	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**9**	**10**	**11**	**12**
Réponse	**Vrai**								**Faux**				
Options utilisées	Breadth-first[a] Active-clock reduction[b]									Breadth-first Active-clock reduction Convex-hull approximation[c]			
Temps (en s) Mémoire (en KB)	30 22992	41 30336	43 31240	62 43696	193 112464	898 445232	911 449776	1007 480728	1753 771280	550 222312	556 222312	552 222312	555 222312

[a] Ceci signifie que le parcours du graphe se fait en largeur.

[b] Les horloges non utilisées sont détectées et éliminées.

[c] À chaque étape du calcul, si deux zones ont été calculées pour un même état de contrôle, ce qui est conservé en mémoire, c'est la plus petite zone contenant ces deux zones.

Tableau 10.8: Résultat des expériences pour la propriété **Info-perte**

Un seul destinataire. (720 états de contrôle dans le produit synchronisé, 5 horloges, 13 paramètres, 25 variables bornées par des constantes de 2 à 10)

Les paramètres de temps autres que `PERIODE_NAK` et `MIN_PERIODE_DATA` sont fixés pour toutes les expériences. Voilà leurs valeurs :

`SPM_DELAI`	1
`MIN_DATA_DELAI`	2
`MAX_DATA_DELAI`	3
`MIN_PERIODE_SPM`	3
`MAX_PERIODE_SPM`	5

Nos résultats sont alors présentés dans le tableau suivant :

`PERIODE_NAK`	**2**		**3**			**4**		
`MIN_PERIODE_DATA`	**1**	**2**	**1**	**2**	**3**	**3**	**4**	**5**
Réponse	**Vrai**	**Faux**	**Vrai**	**Vrai**	**Faux**	**Vrai**	**Faux**	**Faux**
Options utilisées	Breadth-first Active-clock reduction						Breadth-first Active-clock reduction Convex-hull approximation	
Temps (en s) Mémoire (en KB)	36 27936	199 120720	43 33296	48 35920	292 174640	72 50752	215 115608	219 113432

Tableau 10.9: Résultats des expériences pour la propriété **Pas-de-perte**, un seul destinataire

Enfin, il faut bien noter que les vérifications ont été effectuées en donnant aux différents paramètres des valeurs peu élevées car, déjà pour ces faibles valeurs, nous avons été confrontés à des problèmes de mémoire et de temps de vérification très importants. Cependant, même avec ces petites valeurs des paramètres, nous avons pu détecter deux problèmes dans le protocole.

10.4 Conclusion

Dans ce travail, nous avons modélisé à partir d'un cahier des charges un protocole réel de communication. Nous avons alors vérifié deux propriétés fondamentales avec l'outil de model-checking UPPAAL.

La modélisation simplifiée de PGM que nous proposons utilise naturellement des horloges, des variables bornées et des tableaux. C'est la raison pour laquelle nous avons choisi l'outil UPPAAL pour l'implémenter car c'est, à notre connaissance, le seul outil vérifiant des systèmes temporisés qui offre ces structures de données. D'autre part, les modules de simulation et d'interface graphique d'UPPAAL se sont avérés très utiles lors de la phase de modélisation. Ils ont permis une étape intermédiaire de test du modèle, pour vérifier que les automates choisis correspondaient bien à ce que nous souhaitions modéliser. Une adaptation avec des variables non bornées pourrait bien sûr être envisagée dans HYTECH, mais avec un risque d'aggravation des problèmes d'espace mémoire.

Ce travail nous a donc permis d'être confrontés à des problèmes de vérification beaucoup plus pratiques. Nous avons construit un modèle à partir d'un cahier des charges. Notons que la modélisation que nous avons faite n'a strictement rien à voir avec celle réalisée par Yijia Chen dans [Che01]. Cette modélisation est très proche du protocole réel et utilise plusieurs files pour représenter les canaux de communications entre les différents composants. Elle était donc difficilement implémentable dans un des outils existants. C'est pour cette raison que nous ne l'avons pas utilisée pour réaliser ce travail.

Le problème de l'explosion combinatoire que nous avions évoqué dans le chapitre 1 est apparu assez clairement dans cette étude (voir par exemple le tableau 10.10 à la page 240).

Deux destinataires. (17280 états de contrôle dans le produit synchronisé, 6 horloges, 13 paramètres, 36 variables bornées de 2 à 10)

Les paramètres de temps autres que `PERIODE_NAK` et `MIN_PERIODE_DATA` sont fixés pour toutes les expériences. Voilà leurs valeurs :

`SPM_DELAI`	1
`MIN_DATA_DELAI`	2
`MAX_DATA_DELAI`	3
`MIN_PERIODE_SPM`	3
`MAX_PERIODE_SPM`	5

Nos résultats sont alors présentés dans le tableau suivant :

`PERIODE_NAK`	**3**	**3**	**3**	**3**
`MIN_PERIODE_DATA`	**1**	**2**	**3**	**4**
Réponse	**Vrai**	**Vrai**	**Faux**	**Faux**
Options utilisées	Breadth-first Active-clock reduction		Breadth-first Active-clock reduction Convex-hull approximation	
Temps (en s) Mémoire (en KB)	4743 1239832	5775 1402200	918 630608	939 597440

Notons l'énorme place mémoire (et le temps assez important) qu'ont nécessité les deux premières expériences...

Tableau 10.10: Résultats des expériences pour la propriété **Pas-de-perte**, deux destinataires

Quatrième partie

Conclusion

Chapitre 11

Conclusion et perspectives

L'objectif principal de ce travail était de participer au développement de modèles et d'algorithmes pour le model-checking des systèmes temporisés. Nous allons faire un bilan « diagonal » de ce travail, pour ne pas reprendre linéairement les chapitres un par un. Cette conclusion « complète » donc toutes celles présentées à la fin de chaque chapitre. Mes contributions peuvent par exemple se récapituler comme suit :

À la recherche d'un socle théorique solide pour les systèmes temporisés. Dans le cadre non temporisé, les langages formels reconnaissables forment une base théorique solide, ce qui en fait un modèle robuste, utilisable pour de nombreuses applications. Cherchant à proposer une base théorique aussi solide pour les systèmes temporisés, nous avons défini la notion de *langages de données*, une extension des langages temporisés. Dans ce nouveau cadre, nous avons alors proposé plusieurs formalismes **équivalents** :
- un formalisme algébrique à base de monoïdes (chapitre 5),
- une notion naturelle d'automates utilisant un nombre fini de registres (chapitre 5),
- un langage logique basée sur la logique monadique du second ordre avec des prédicats supplémentaires, appelés des prédicats de données (chapitre 6).

Ces différentes approches nous semblent très intéressantes et nous permettent de commencer à faire un rapprochement avec le cadre non temporisé. Notons qu'en outre, les expressions rationnelles telles que nous les avons définies dans le cadre temporisé (chapitre 4) semblent pouvoir être modifiées pour venir renforcer encore notre modèle. Tout ceci peut se résumer dans le tableau 11.1.
Les travaux que nous avons entamés dans ce domaine doivent être poursuivis : des sous-classes des langages de données reconnus (grâce à l'un des formalismes précédents) peuvent être naturellement définies, par exemple les langages de données apériodiques qui sont des langages de données reconnus grâce au formalisme algébrique par des monoïdes apériodiques. Il nous semble très important d'étudier ces langages, car, dans le cadre des langages formels, ceux-ci possédaient une mine de propriétés très intéressantes (équivalence avec la logique du premier ordre, avec la logique du temps linéaire, les langages sans étoile, etc...).

Les automates temporisés avec mises à jour, de la théorie à l'algorithmique. Nous avons défini et étudié de manière approfondie une extension des automates temporisés, les *automates temporisés avec mises à jour*. De larges classes de ce modèle ont alors été montrées décidables. La frontière entre les sous-classes décidables et les sous-classes indécidables ont été dessinées avec précision (chapitre 3). Nous avons alors prouvé que les sous-classes décidables n'étaient pas

Langages formels	**Langages de données**
Automates finis	***Automates de données (chapitre 5)***
Expressions rationnelles	Adaptation des résultats du chapitre 4
Caractérisation logique MSO(<), LTL	***Logique basée sur*** MSO(<) ***(chapitre 6)***
Caractérisation algébrique (en utilisant des monoïdes finis)	***Mécanisme algébrique (chapitre 5)***

Tableau 11.1: Comparaison langages formels/langages de données

plus expressives que les automates temporisés classiques, mais par contre, elles permettent de représenter de manière beaucoup plus concise beaucoup de systèmes (chapitre 3).

Nous avons ensuite étudié un algorithme implémentable pour tester l'accessibilité d'états d'automates temporisés avec mises à jour (chapitre 8). Cet algorithme prend en entrée un automate temporisé avec mises à jour (appartenant à une classe décidable) et construit un système d'inéquations Diophantiennes linéaires. En résolvant ce système, nous obtenons la valeur d'un paramètre qui nous permet de calculer une surapproximation de l'ensemble des états accessibles de l'automate (la surapproximation utilisée dépend de la valeur du paramètre calculée). Bien que cet algorithme calcule une surapproximation des états accessibles, grâce au choix du paramètre, l'algorithme reste correct vis-à-vis de l'accessibilité. Notons que la preuve de correction de cet algorithme, que nous avons détaillée, implique la correction de l'algorithme utilisé par exemple dans UPPAAL et KRONOS pour les automates temporisés classiques.

Nous avons aussi décrit une implémentation possible de cet algorithme en utilisant la structure de données des DBMs (chapitre 8), nul doute que d'autres structures de données comme les CDDs puissent aussi s'adapter à notre algorithme. La véritable implémentation de ce modèle dans un outil existant n'est plus très loin, il serait vraiment dommage de ne pas le faire, car les mises à jour apparaissent comme des macros très intéressantes pour la modélisation.

Les limites de l'accessibilité. Pour la plupart des systèmes, les propriétés d'accessibilité sont les plus simples à vérifier. Dans le cadre d'une extension des automates temporisés classiques (correspondant entre autres au modèle utilisé dans l'outil UPPAAL), nous avons étudié un algorithme permettant de vérifier des propriétés plus compliquées que de l'accessibilité juste grâce à un test d'accessibilité *via* la réduction utilisant les automates de test. En dehors de l'algorithme qui permet de calculer les automates de test, nous proposons une caractérisation complète de l'ensemble des propriétés qui peuvent être testées de cette façon. Nous utilisons pour cela une logique modale temporisée (chapitre 9). Remarquons qu'un outil permettant de construire un automate de test à partir d'une formule logique a été implémenté en grande partie par Augusto Burgueño [ABL98].

Où l'on apprend beaucoup de la pratique. Outre les travaux sur le développement de modèles et d'algorithmes pour le model-checking, nous avons aussi travaillé, dans le cadre du projet

RNRT Calife[1], sur un protocole réel de communication, le protocole PGM (chapitre 10). Ce travail, très différent de ceux effectués jusqu'à présent nous a réellement confrontés à la vérification en pratique. En effet, PGM n'est pas ce que l'on peut appeler un « exemple d'école », mais c'est au contraire un vrai système qui est implémenté et utilisé. Cette étude nous a permis de nous rendre compte de la difficulté réelle que représente la vérification en pratique et de l'impérieuse nécessité de disposer de modèles compacts (possédant des types de données évolués) et d'algorithmes très efficaces... le problème de l'explosion combinatoire n'est malheureusement pas une légende. En outre, ce travail nous a permis de mettre à jour deux problèmes dans le cahier des charges du protocole PGM [Spe01].

Nouvelles perspectives

Lorsque nous avons récapitulé les travaux réalisés sur les langages de données, nous avons décrit quelques perspectives pour continuer ce travail. Nous allons maintenant proposer deux voies possibles qui sont dans la lignée des travaux précédents et qui m'intéressent particulièrement.

Une extension des systèmes temporisés, les systèmes hybrides. Les variables que peuvent manipuler les automates temporisés, appelées *horloges*, ont toutes une pente égale à 1, c'est-à-dire qu'elles évoluent toutes en même temps et à la même vitesse, elles peuvent être comparées à des constantes ou entre elles et elles peuvent être remises à zéro. Les manipulations autorisées sont donc assez limitées. Dans les automates temporisés avec mises à jour, des opérations un peu plus générales que les remises à zéro ont été proposées. Ce modèle permet de représenter des systèmes temporisés de manière plus synthétique qu'avec les automates temporisés classiques. Cependant, dans ce modèle aussi, les variables sont des horloges et ont donc toujours une pente égale à 1. Ceci n'est pas adapté, par exemple à la modélisation d'un certain nombre de problèmes comme des problèmes d'ordonnancement. Pour pallier cette difficulté, une extension assez naturelle des automates temporisés vient tout de suite à l'esprit : il suffit d'autoriser les horloges à s'arrêter. Une horloge peut donc avoir soit une pente de 0, soit une pente de 1 et ceci peut changer lors des changements d'états. Ces nouveaux automates sont les *automates hybrides 0/1*, ils forment une sous-classe des automates hybrides [ACH$^+$95, HKPV95, Hen96, CL00b], modèle indécidable. Des travaux sur l'utilisation des automates temporisés et des automates hybrides pour modéliser des problèmes d'ordonnancement ont commencé à voir le jour [AM01, AM02] (par exemple).

Je souhaite étudier le modèle des automates hybrides 0/1 pour dégager des sous-classes décidables qui permettraient de modéliser des problèmes tels que l'ordonnancement de tâches. Je souhaite aussi étudier des algorithmes pour ces sous-classes décidables et proposer une structure de données adéquate à leur implémentation, les structures de données utilisées pour les automates temporisés classiques ne s'adaptant pas de manière immédiate aux automates hybrides 0/1.

Au-delà de la vérification, la synthèse de contrôleurs. Les systèmes réels ne fonctionnent pas souvent de manière indépendante et sont souvent plongés dans un environnement qui n'est pas toujours contrôlable. Il est alors utile de pouvoir construire des *contrôleurs* qui permettent au système d'avoir un comportement correct même si l'environnement intervient sur le système. Ce problème a été largement étudié dans le cadre classique non temporisé, ce sujet étant souvent mis en relation avec la théorie des jeux [Tho95, AHK97, AHKV98, KMTV00, KV00].
Dans un cadre temporisé, le problème est plus difficile et il n'a obtenu que des solutions partielles [WTH91, AMPS98, AM99, DM02].

[1] `http://www.loria.fr/calife/`

En particulier, dans tous les travaux précités, la construction d'un contrôleur requiert de connaître l'évolution de toutes les horloges de l'environnement, alors que ce dernier représente justement une composante du système non contrôlable voire non connue précisément. Je souhaite donc étudier le problème de la synthèse de contrôleurs pour les automates temporisés lorsque toutes les informations sur les horloges de l'environnement ne sont pas connues.

Par ailleurs, dans les travaux actuels, les seules propriétés des automates temporisés pour lesquelles il est possible de construire des contrôleurs sont des propriétés d'accessibilité d'états, voire des propriétés du type Büchi, Rabin, etc... (c'est-à-dire des propriétés liées aux états de contrôle du modèle). J'aimerais étudier le problème de la synthèse de contrôleurs pour les systèmes temporisés pour des langages de propriétés un peu plus expressifs, par exemple, je pense qu'il serait possible d'utiliser des langages logiques tels que celui que nous avons étudié dans le chapitre 9 ou celui étudié dans [LLW95].

Je souhaite aussi étudier le problème de la synthèse de contrôleurs pour l'extension des automates temporisés définie et étudiée dans [BFH+01a, BFH+01b, LBB+01], les automates temporisés avec coûts dans lesquels des coûts sont associés au franchissement d'une transition et à l'attente dans un état de contrôle. Le problème devient alors : est-il possible de construire un contrôleur qui permet de minimiser le coût des exécutions dans un automate temporisé ?

Index

Symboles

A

B

C

D

S

T

U

V

Z

Bibliographie

[ABB80] Autebert, J.-M., Beauquier, J., and Boasson, L. *Langages sur des alphabets infinis*. Discrete Applied Mathematics, vol. 2 :1–20, 1980.

[ABB+01] Amnell, T., Behrmann, G., Bengtsson, J., D'Argenio, P. R., David, A., Fehnker, A., Hune, T., Jeannet, B., Larsen, K. G., Möller, O., Pettersson, P., Weise, C., and Yi, W. Uppaal – *Now, Next, and Future*. In *Proc. Modelling and Verification of Parallel Processes (*Movep*2k)*, vol. 2067 of *Lecture Notes in Computer Science*, pp. 99–124. Springer-Verlag, 2001.

[ABBL98] Aceto, L., Bouyer, P., Burgueño, A., and Larsen, K. G. *The Power of Reachability Testing for Timed Automata*. In *Proc. 18th Conference on Foundations of Software Technology and Theoretical Computer Science (FST&TCS'98)*, vol. 1530 of *Lecture Notes in Computer Science*, pp. 245–256. Springer-Verlag, 1998.

[ABBL02] Aceto, L., Bouyer, P., Burgueño, A., and Larsen, K. G. *The Power of Reachability Testing for Timed Automata*. Theoretical Computer Science, 2002. To appear. Available as LSV Research Report LSV-01-6, `http://www.lsv.ens-cachan.fr/Publis/RAPPORTS_LSV/rr-lsv-2001-6.rr.ps`.

[Abd01] Abdulla, P. A. *Using (Timed) Petri Nets for Verification of Parametrized (Timed) Systems*. Invited Contribution in Verification of Parameterized Systems (VEPAS'01), ICALP'01 satellite workshop, 2001.

[ABK+97] Asarin, E., Bozga, M., Kerbrat, A., Maler, O., Pnueli, A., and Rasse, A. *Data-Structures for the Verification of Timed Automata*. In *Proc. International Workshop on Hybrid and Real-Time Systems (HART'97)*, vol. 1201 of *Lecture Notes in computer Science*, pp. 346–360. Springer-Verlag, 1997.

[ABL98] Aceto, L., Burgueño, A., and Larsen, K. G. *Model-Checking via Reachability Testing for Timed Automata*. In *Proc. 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, vol. 1384 of *Lecture Notes in Computer Science*, pp. 263–280. Springer-Verlag, 1998.

[ACD90] Alur, R., Courcoubetis, C., and Dill, D. *Model-Checking in Dense Real-Time*. In *Proc. 5th IEEE Symposium on Logic in Computer Science (LICS'90)*, pp. 414–425. IEEE Computer Society Press, 1990.

[ACD+92a] Alur, R., Courcoubetis, C., Dill, D., Halbwachs, N., and Wong-Toi, H. *An Implementation of Three Algorithms for Timing Verification Based on Automata Emptiness*. In *Proc. 13th IEEE Real-Time Systems Symposium (RTSS'92)*, pp. 157–166. IEEE Computer Society Press, 1992.

[ACD+92b] Alur, R., Courcoubetis, C., Dill, D. L., Halbwachs, N., and Wong-Toi, H. *Minimization of Timed Transition Systems*. In *Proc. 3rd International Conference on Concurrency Theory (CONCUR'92)*, vol. 630 of *Lecture Notes in Computer Science*, pp. 340–354. Springer-Verlag, 1992.

[ACD93] Alur, R., Courcoubetis, C., and Dill, D. *Model-Checking in Dense Real-Time*. Information and Computation, vol. 104(1) :2–34, 1993.

[Ace94] Aceto, L. *A Note on Complete Lattices and Tarski's Fixed-Point Theorem*, 1994. Available on http://www.cs.auc.dk/~luca/DAT4/lattices.ps, Aalborg University, Denmark.

[ACH+95] Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T. A., Pei-Hsin, H., Nicollin, X., Olivero, A., Sifakis, J., and Yovine, S. *The Algorithmic Analysis of Hybrid Systems*. Theoretical Computer Science, vol. 138 :3–34, 1995.

[ACHH93] Alur, R., Courcoubetis, C., Henzinger, T. A., and Ho, P.-H. *Hybrid Automata : an Algorithmic Approach to Specification and Verification of Hybrid Systems*. In *Proc. Hybrid Systems*, vol. 736 of *Lecture Notes in Computer Science*, pp. 209–229. Springer-Verlag, 1993.

[ACM97] Asarin, E., Caspi, P., and Maler, O. *A Kleene Theorem for Timed Automata*. In *Proc. 12th IEEE Symposium on Logic in Computer Science (LICS'97)*, pp. 160–171. IEEE Computer Society Press, 1997.

[ACM01] Asarin, E., Caspi, P., and Maler, O. *Timed Regular Expressions*, 2001. Submitted to the Journal of the Association for Computing Machinery. Available on http://www-verimag.imag.fr/~maler/Papers/arith.ps.gz.

[AD90] Alur, R. and Dill, D. *Automata for Modeling Real-Time Systems*. In *Proc. 17th International Colloquium on Automata, Languages, and Programming (ICALP'90)*, vol. 443 of *Lecture Notes in Computer Science*, pp. 322–335. Springer-Verlag, 1990.

[AD94] Alur, R. and Dill, D. *A Theory of Timed Automata*. Theoretical Computer Science, vol. 126(2) :183–235, 1994.

[AFH94] Alur, R., Fix, L., and Henzinger, T. A. *Event-Clock Automata : a Determinizable Class of Timed Automata*. In *Proc. 6th International Conference on Computer Aided Verification (CAV'94)*, vol. 818 of *Lecture Notes in Computer Science*, pp. 1–13. Springer-Verlag, 1994.

[AH91] Alur, R. and Henzinger, T. A. *Logics and Models of Real-Time : a Survey*. In *Proc. Real-Time : Theory in Practice, REX Workshop*, vol. 600 of *Lecture Notes in Computer Science*, pp. 74–106. Springer-Verlag, 1991.

[AH92] Alur, R. and Henzinger, T. A. *Back to the Future : towards a Theory of Timed Regular Languages*. In *Proc. 33rd IEEE Symposium on Foundations of Computer Science (FOCS'92)*, pp. 177–186. IEEE Computer Society Press, 1992.

[AH93] Alur, R. and Henzinger, T. A. *Real-Time Logics : Complexity and Expressiveness*. Information and Computation, vol. 104(1) :35–77, 1993.

[AH94] Alur, R. and Henzinger, T. A. *A Really Temporal Logic*. Journal of the Association for Computing Machinery, vol. 41 :181–204, 1994.

[AHK97] Alur, R., Henzinger, T. A., and Kupferman, O. *Alternating-time Temporal Logic*. In *Proc. 8th IEEE Symposium on Foundations of Computer Science (FOCS'97)*, pp. 100–109. IEEE Computer Society Press, 1997.

[AHKV98] Alur, R., Henzinger, T. A., Kupferman, O., and Vardi, M. *Alternating Refinement Relations*. In *Proc. 9th International Conference on Concurrency Theory (CONCUR'98)*, vol. 1466 of *Lecture Notes in Computer Science*, pp. 163–178. Springer-Verlag, 1998.

[AHV93] Alur, R., Henzinger, T. A., and Vardi, M. *Parametric Real-Time Reasoning*. In *Proc. 25th ACM Symposium on Theory of Computing*, pp. 592–601. ACM, 1993.

[AIPP00] ACETO, L., INGÓLFSDÓTTIR, A., PEDERSEN, M. L., and POULSEN, J. *Characteristic Formulae for Timed Automata*. Theoretical Informatics and Application, vol. 34 :565–584, 2000.

[AL99] ACETO, L. and LAROUSSINIE, F. *Is your Model-Checker on Time ?*. In *Proc. 24th International Symposium on Mathematical Foundations of Computer Science (MFCS'99)*, vol. 1672 of *Lecture Notes in Computer Science*, pp. 125–136. Springer-Verlag, 1999.

[AL02] ACETO, L. and LAROUSSINIE, F. *Is your Model-Checker on Time ? On the Complexity of Model-Checking for Timed Modal Logics*. Journal of Logic and Algebraic Programming, 2002. To appear. Available on http://www.lsv.ens-cachan.fr/Publis/PAPERS/AceLar-JLAP.ps.

[Alu91] ALUR, R. *Techniques for automatic verification of real-time systems*. Ph.D. thesis, Stanford University, USA, 1991.

[Alu99] ALUR, R. *Timed Automata*. In *Proc. 11th International Conference on Computer Aided Verification (CAV'99)*, vol. 1633 of *Lecture Notes in Computer Science*, pp. 8–22. Springer-Verlag, 1999.

[AM99] ASARIN, E. and MALER, O. *As Soon as Possible : Time Optimal Control for Timed Automata*. In *Proc. 4th International Workshop on Hybrid Systems : Computation and Control (HSCC'01)*, vol. 1569 of *Lecture Notes in Computer Science*, pp. 19–30. Springer-Verlag, 1999.

[AM01] ABDEDDAIM, Y. and MALER, O. *Job-Shop Scheduling using Timed Automata*. In *Proc. 13th International Conference on Computer Aided Verification (CAV'01)*, vol. 2102 of *Lecture Notes in Computer Science*, pp. 478–492. Springer-Verlag, 2001.

[AM02] ABDEDDAÏM, Y. and MALER, O. *Preemptive Job-Shop Scheduling using Stopwatch Automata*. In *Proc. 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, vol. 2280 of *Lecture Notes in Computer Science*, pp. 113–126. Springer-Verlag, 2002.

[AMPS98] ASARIN, E., MALER, O., PNUELI, A., and SIFAKIS, J. *Controller Synthesis for Timed Automata*. In *Proc. IFAC Symposium on System Structure and Control*, pp. 469–474. Elsevier, 1998.

[And95] ANDERSEN, H. R. *Partial Model-Checking (Extended Abstract)*. In *Proc. 10th IEEE Symposium on Logic in Computer Science (LICS'95)*, pp. 398–407. IEEE Computer Society Press, 1995.

[ari96] ARIANE *5 – Flight 501 Failure*, 1996. Report by the Inquiry Board, chairman : J-L Lions. Available on http://www.cnes.fr/espace_pro/communiques/cp96/rapport_501/rapport_501_1.html.

[Asa98] ASARIN, E. *Equations on Timed Languages*. In *Proc. 1st International Workshop on Hybrid Systems : Computation and Control (HSCC'98)*, vol. 1386 of *Lecture Notes in Computer Science*, pp. 1–12. Springer-Verlag, 1998.

[Bar89] BARRINGTON, D. A. M. *Bounded-Width Branching Programs Recognize Exactly Those Languages in NC^1*. Journal of Computer and Systems Science, vol. 38(1) :150–164, 1989.

[BBP01] BÉRARD, B., BOUYER, P., and PETIT, A. *Modélisation du protocole PGM et de certaines de ses propriétés en* UPPAAL, 2001. Fourniture 4.4 du projet RNRT Calife. Available on http://www.loria.fr/projets/calife/WebCalifePublic/FOURNITURES/F4.4.ps.gz.

[Büc62] BÜCHI, J. R. *On a Decision Method in Restricted Second Order Arithmetic*. In *Proc. (1960) Int Congr. Logic, Methodology and Philosophy of Science*, pp. 1–11. Stanford University Press, 1962.

[BD99] BOYER, M. and DIAZ, M. *Non Equivalence between Time Petri Nets and Time Stream Petri Nets*. In *Proc. 8th International Workshop on Petri Nets and Performance Modeling (PNPM'99)*, pp. 198–207. 1999.

[BD00] BÉRARD, B. and DUFOURD, C. *Timed Automata and Additive Clock Constraints*. Information Processing Letters, vol. 75(1–2) :1–7, 2000.

[BDFP00a] BOUYER, P., DUFOURD, C., FLEURY, E., and PETIT, A. *Are Timed Automata Updatable ?*. In *Proc. 12th International Conference on Computer Aided Verification (CAV'2000)*, vol. 1855 of *Lecture Notes in Computer Science*, pp. 464–479. Springer-Verlag, 2000.

[BDFP00b] BOUYER, P., DUFOURD, C., FLEURY, E., and PETIT, A. *Expressiveness of Updatable Timed Automata*. In *Proc. 25th International Symposium on Mathematical Foundations of Computer Science (MFCS'2000)*, vol. 1893 of *Lecture Notes in Computer Science*, pp. 232–242. Springer-Verlag, 2000.

[BDGP98] BÉRARD, B., DIEKERT, V., GASTIN, P., and PETIT, A. *Characterization of the Expressive Power of Silent Transitions in Timed Automata*. Fundamenta Informaticae, vol. 36(2) :145–182, 1998.

[BDM+98] BOZGA, M., DAWS, C., MALER, O., OLIVERO, A., TRIPAKIS, S., and YOVINE, S. KRONOS *: a Model-Checking Tool for Real-Time Systems*. In *Proc. 10th International Conference on Computer Aided Verification (CAV'98)*, vol. 1427 of *Lecture Notes in Computer Science*, pp. 546–550. Springer-Verlag, 1998.

[BF99] BÉRARD, B. and FRIBOURG, L. *Automated Verification of a Parametric Real-Time Program : the ABR Conformance Protocol*. In *Proc. 11th International Conference on Computer Aided Verification (CAV'99)*, vol. 1633 of *Lecture Notes in Computer Science*, pp. 96–107. Springer-Verlag, 1999.

[BFH90] BOUAJJANI, A., FERNANDEZ, J., and HALBWACHS, N. *Minimal Model Generation*. In *Proc. 2nd International Conference on Computer Aided Verification (CAV'90)*, vol. 431 of *Lecture Notes in Computer Science*, pp. 197–203. Springer-Verlag, 1990.

[BFH+01a] BEHRMANN, G., FEHNKER, A., HUNE, T., LARSEN, K. G., PETTERSSON, P., ROMIJN, J., and VAANDRAGER, F. *Efficient Guiding Towards Cost-Optimality in* UPPAAL. In *Proc. 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, vol. 2031 of *Lecture Notes in Computer Science*, pp. 174–188. Springer-Verlag, 2001.

[BFH+01b] BEHRMANN, G., FEHNKER, A., HUNE, T. S., LARSEN, K. G., PETTERSSON, P., ROMIJN, J., and VAANDRAGER, F. *Minimum-Cost Reachability for Priced Timed Automata*. In *Proc. 4th International Workshop on Hybrid Systems : Computation and Control (HSCC'01)*, vol. 2034 of *Lecture Notes in Computer Science*, pp. 147–161. Springer-Verlag, 2001.

[BFHR92] BOUAJJANI, A., FERNANDEZ, J., HALBWACHS, N., and RAYMOND, P. *Minimal State Graph Generation*. Science of Computer Programming, vol. 18(3) :247–269, 1992.

[BFKM02] BÉRARD, B., FRIBOURG, L., KLAY, F., and MONIN, J.-F. *A Compared Study of Two Correctness Proofs for the Standardized Algorithm of ABR Conformance*. Formal Methods in System Design, 2002. To appear. Available on
`http://www.lsv.ens-cachan.fr/~fribourg/PAPERS/fmsd01.ps`.

[BGK+96] BENGTSSON, J., GRIFFIOEN, W. D., KRISTOFFERSEN, K. J., LARSEN, K. G., LARSSON, F., PETTERSSON, P., and YI, W. *Verification of an Audio Protocol with Bus Collision*

Using UPPAAL. In *Proc. 8th International Conference on Computer Aided Verification (CAV'96)*, vol. 1102 of *Lecture Notes in Computer Science*, pp. 244–256. Springer-Verlag, 1996.

[BGP96] BÉRARD, B., GASTIN, P., and PETIT, A. *On the Power of Non-Observable Actions in Timed Automata*. In *Proc. 13th Annual Symposium on Theoretical Aspects of Computer Science (STACS'96)*, vol. 1046 of *Lecture Notes in Computer Science*, pp. 257–268. Springer-Verlag, 1996.

[BGS00] BORNOT, S., GOESSLER, G., and SIFAKIS, J. *On the Construction of Live Systems*. In *Proc. 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'00)*, vol. 1785 of *Lecture Notes in Computer Science*, pp. 109–126. Springer-Verlag, 2000.

[BL96] BENGTSSON, J. and LARSSON, F. UPPAAL*, a Tool for Automatic Verification of Real-Time Systems*. Master's thesis, Department of Computer Science, Uppsala University, Sweden, 1996. Available on `http://www.docs.uu.se/docs/rtmv/papers/bl :report96.ps.gz`.

[BLL+95] BENGTSSON, J., LARSEN, K. G., LARSSON, F., PETTERSSON, P., and YI, W. UPPAAL *– a Tool Suite for Automatic Verification of Real-Time Systems*. In *Proc. DIMACS/SYCON Workshop Hybrid Systems III : Verification and Control*, vol. 1066 of *Lecture Notes in Computer Science*, pp. 232–243. 1995.

[BLL+98] BENGTSSON, J., LARSEN, K. G., LARSSON, F., PETTERSSON, P., YI, W., and WEISE, C. *New Generation of* UPPAAL. In *Proc. International Workshop on Software Tools for Technology Transfer*. 1998.

[BLP+99] BEHRMANN, G., LARSEN, K. G., PEARSON, J., WEISE, C., YI, W., and LIND-NIELSEN, J. *Efficient Timed Reachability Analysis Using Clock Difference Diagrams*. In *Proc. 11th International Conference on Computer Aided Verification (CAV'99)*, vol. 1633 of *Lecture Notes in Computer Science*, pp. 341–353. Springer-Verlag, 1999.

[Bou01] BOUYER, P. *Updatable Timed Automata, an Algorithmic Approach*, 2001. Submitted to Formal Methods in System Design. Available as LSV Research Report LSV-01-12 on `http://www.lsv.ens-cachan.fr/Publis/RAPPORTS_LSV/rr-lsv-2001-12.rr.ps`.

[Bou02] BOUYER, P. *A Logical Characterization of Data Languages*. Information Processing Letters, 2002. To appear. Available as Research Report LSV-01-10 on `http://www.lsv.ens-cachan.fr/Publis/RAPPORTS_LSV/rr-lsv-2001-10.rr.ps`.

[BP99] BOUYER, P. and PETIT, A. *Decomposition and Composition of Timed Automata*. In *Proc. 26th International Colloquium on Automata, Languages, and Programming (ICALP'99)*, vol. 1644 of *Lecture Notes in Computer Science*, pp. 210–219. Springer-Verlag, July 1999.

[BP02] BOUYER, P. and PETIT, A. *A Kleene/Büchi-like Theorem for Clock Languages*. Journal of Automata, Languages and Combinatorics, 2002. To appear. Available on `http://www.lsv.ens-cachan.fr/Publis/PAPERS/BP-JALC2001.ps`.

[BPT01a] BOUYER, P., PETIT, A., and THÉRIEN, D. *An Algebraic Characterization of Data and Timed Languages*. In *Proc. 12th International Conference on Concurrency Theory (CONCUR'01)*, vol. 2154 of *Lecture Notes in Computer Science*, pp. 248–261. Springer-Verlag, 2001.

[BPT01b] BOUYER, P., PETIT, A., and THÉRIEN, D. *An Algebraic Approach to Data Languages and Timed Languages*, 2001. Submitted to Information and Computation.

[Brz62] BRZOZOWSKI, J. A. *A survey of regular expressions and their applications*. IRE transactions on Electronic Computers, vol. 3 :324–335, 1962.

[BS98] Bornot, S. and Sifakis, J. *On the Composition of Hybrid Systems*. In *Proc. 1st International Workshop on Hybrid Systems : Computation and Control (HSCC'98)*, vol. 1386 of *Lecture Notes in Computer Science*, pp. 69–83. Springer-Verlag, 1998.

[BST97] Bornot, S., Sifakis, J., and Tripakis, S. *Modeling Urgency in Timed Systems*. In *Proc. International Symposium Compositionality (COMPOS'97)*, vol. 1536 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[BT88] Barrington, D. and Thérien, D. *Finite Monoids and the Fine Structure of NC^1*. Journal of the Association for Computing Machinery, vol. 35(4) :941–952, 1988.

[BT01] Brinksma, E. and Tretmans, J. *Testing Transition Systems : An Annotated Bibliography*. In *Proc. Modelling and Verification of Parallel Processes (*Movep*2k)*, vol. 2067 of *Lecture Notes in Computer Science*, pp. 187–195. Springer-Verlag, 2001.

[BTY97] Bouajjani, A., Tripakis, S., and Yovine, S. *On-the-Fly Symbolic Model-Checking for Real-Time Systems*. In *Proc. 18th IEEE Real-Time Systems Symposium (RTSS'97)*, pp. 25–35. IEEE Computer Society Press, 1997.

[CES86] Clarke, E., Emerson, E., and Sistla, A. *Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications*. ACM Transactions on Programming Languages and Systems, vol. 8(2) :244–263, 1986.

[CG00] Choffrut, C. and Goldwurm, M. *Timed Automata with Periodic Clock Constraints*. Journal of Automata, Languages and Combinatorics, vol. 5(4) :371–404, 2000.

[CGP99] Clarke, E., Grumberg, O., and Peled, D. *Model-Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[Che01] Chen, Y. *Modelling PGM by P-Automata*, 2001. Calife draft.

[CJ98] Comon, H. and Jurski, Y. *Multiple Counters Automata, Safety Analysis and Presburger Arithmetic*. In *Proc. 10th International Conference on Computer Aided Verification (CAV'98)*, vol. 1427 of *Lecture Notes in Computer Science*, pp. 268–279. Springer-Verlag, 1998. Also available as LSV Research Report LSV–98–1, 1998.

[CJ99] Comon, H. and Jurski, Y. *Timed Automata and the Theory of Real Numbers*. In *Proc. 10th International Conference on Concurrency Theory (CONCUR'99*, vol. 1664 of *Lecture Notes in Computer Science*, pp. 242–257. Springer-Verlag, 1999.

[CL93] Cori, R. and Lascar, D. *Logique mathématique*. Masson, 1993.

[CL00a] Cassez, F. and Laroussinie, F. *Model-Checking for Hybrid Systems by Quotienting and Constraints Solving*. In *Proc. 12th International Conference on Computer Aided Verification (CAV'2000)*, vol. 1855 of *Lecture Notes in Computer Science*, pp. 373–388. Springer-Verlag, 2000.

[CL00b] Cassez, F. and Larsen, K. G. *The Impressive Power of Stopwatches*. In *Proc. 11th International Conference on Concurrency Theory (CONCUR'00)*, vol. 1877 of *Lecture Notes in Computer Science*, pp. 138–152. Springer-Verlag, 2000.

[Daw97] Daws, C. *Analyse par simulation symbolique des systèmes temporisés avec* Kronos. Research report, Verimag, 1997.

[Daw98] Daws, C. *Méthodes d'analyse de systèmes temporisés : de la théorie à la pratique*. Ph.D. thesis, Institut National Polytechnique de Grenoble, France, Oct. 1998.

[DGP97] Diekert, V., Gastin, P., and Petit, A. *Removing ε-Transitions in Timed Automata*. In *Proc. 14th Annual Symposium on Theoretical Aspects of Computer Science (STACS'97)*, vol. 1200 of *Lecture Notes in Computer Science*, pp. 583–594. 1997.

[Dil89] DILL, D. *Timing Assumptions and Verification of Finite-State Concurrent Systems*. In *Proc. of the Workshop on Automatic Verification Methods for Finite State Systems*, vol. 407 of *Lecture Notes in Computer Science*, pp. 197–212. Springer-Verlag, 1989.

[Dim01a] DIMA, C. *Real-Time Automata*. Journal of Automata, Languages and Combinatorics, vol. 6(1) :3–24, 2001.

[Dim01b] DIMA, C. *Théorie Algébrique des Langages Formels Temps Réel*. Ph.D. thesis, Université Joseph Fourier, Grenoble, France, 2001.

[DM02] D'SOUZA, D. and MADHUSUDAN, P. *Controller Synthesis for Timed Specifications*. In *Proc. 19th International Symposium on Theoretical Aspects of Computer Science (STACS'02)*, vol. 2285 of *Lecture Notes in Computer Science*, pp. 571–582. Springer-Verlag, 2002.

[Dom91] DOMENJOUD, E. *Solving Systems of Linear Diophantine Equations : An Algebraic Approach*. In *Proc. 16th International Symposium on Mathematical Foundations of Computer Science (MFCS'91)*, vol. 520 of *Lecture Notes in Computer Science*, pp. 141–150. Springer-Verlag, 1991.

[D'S00] D'SOUZA, D. *A Logical Characterization of Event Recording Automata*. In *Proc. 9th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'00)*, vol. 1926 of *Lecture Notes in Computer Science*, pp. 240–251. Springer-Verlag, 2000.

[DT98] DAWS, C. and TRIPAKIS, S. *Model-Checking of Real-Time Reachability Properties using Abstractions*. In *Proc. 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, vol. 1384 of *Lecture Notes in Computer Science*, pp. 313–329. Springer-Verlag, 1998.

[Duf97] DUFOURD, C. *Une extension d'un résultat d'indécidabilité pour les automates temporisés*. In *Proc. 9th Rencontres Francophones du Parallélisme (RenPar'97)*. 1997.

[DY96] DAWS, C. and YOVINE, S. *Reducing the Number of Clock Variables of Timed Automata*. In *Proc. 17th IEEE Real-Time Systems Symposium (RTSS'96)*, pp. 73–81. IEEE Computer Society Press, 1996.

[DZ98] DEMICHELIS, F. and ZIELONKA, W. *Controlled Timed Automata*. In *Proc. 9th International Conference on Concurrency Theory (CONCUR'98)*, vol. 1466 of *Lecture Notes in Computer Science*, pp. 455–469. Springer-Verlag, 1998.

[Goe01] GOESSLER, G. *Compositional Modelling of Real-Time-Systems - Theory and Practice*. Ph.D. thesis, Université Joseph Fourier, Grenoble, France, 2001.

[GS94] GORRIERI, R. and SILIPRANDI, G. *Real-Time System Verification using P/T Nets*. In *Proc. 6th International Conference on Computer Aided Verification (CAV'94)*, vol. 818 of *Lecture Notes in Computer Science*, pp. 14–26. Springer-Verlag, 1994.

[Hen96] HENZINGER, T. A. *The Theory of Hybrid Automata*. In *Proc. 11th Annual Symposim on Logic in Computer Science (LICS'96)*, pp. 278–292. IEEE Computer Society Press, 1996.

[Her99] HERRMANN, P. *Renaming is Necessary in Timed Regular Expressions*. Tech. rep., LIAFA 99/18, 1999.

[HHWT95a] HENZINGER, T. A., HO, P.-H., and WONG-TOI, H. HYTECH *: the Next Generation*. In *Proc. 16th IEEE Real-Time Systems Symposium (RTSS'95)*, pp. 56–65. IEEE Computer Society press, 1995.

[HHWT95b] HENZINGER, T. A., HO, P.-H., and WONG-TOI, H. *A User Guide to* HYTECH. In *Proc. 1st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'95)*, vol. 1019 of *Lecture Notes in Computer Science*, pp. 41–71. Springer-Verlag, 1995.

[HHWT97] HENZINGER, T. A., HO, P.-H., and WONG-TOI, H. HYTECH *: A Model-Checker for Hybrid Systems*. Journal of Software Tools for Technology Transfer, vol. 1(1–2) :110–122, Oct. 1997.

[HK94] HENZINGER, T. A. and KOPKE, P. W. *Undecidability Results for Hybrid Systems*. In *Proc. Workshop on Hybrid Systems and Autonomous Control*. 1994.

[HKPM97] HUET, G., KAHN, G., and PAULIN-MOHRING, C. *The Coq Proof Assistant - A Tutorial, Version 6.1*. Rapport technique 204, INRIA, 1997.

[HKPV95] HENZINGER, T. A., KOPKE, P. W., PURI, A., and VARAIYA, P. *What's Decidable about Hybrid Automata ?*. In *Proc. 27th ACM Symposium Theory of Computing (STOC'95)*, pp. 373–382. 1995.

[HKWT95] HENZINGER, T. A., KOPKE, P. W., and WONG-TOI, H. *The Expressive Power of Clocks*. In *Proc. 22nd International Colloquium on Automata, Languages, and Programming (ICALP'95)*, vol. 944 of *Lecture Notes in Computer Science*, pp. 335–346. Springer-Verlag, 1995.

[HLS$^+$93] HERTRAMPF, U., LAUTEMAN, C., SCHWENTICK, T., VOLLMER, H., and WAGNER, K. *On the Power of Polynomial Time Bit-Reductions*. In *Proc. 8th Conference on Structure in Complexity Theory*, pp. 200–207. IEEE Computer Society Press, 1993.

[HNSY94] HENZINGER, T. A., NICOLLIN, X., SIFAKIS, J., and YOVINE, S. *Symbolic Model-Checking for Real-Time Systems*. Information and Computation, vol. 111 :193–244, 1994.

[HRS98] HENZINGER, T. A., RASKIN, J.-F., and SCHOBBENS, P.-Y. *The Regular Real-Time Languages*. In *Proc. 25th International Colloquium on Automata, Languages, and Programming (ICALP'98)*, vol. 1443 of *Lecture Notes in Computer Science*, pp. 580–591. Springer-Verlag, July 1998.

[HRSF01] HUNE, T. S., ROMIJN, J., STOELINGA, M., and F., V. *Linear Parametric Model Checking of Timed Automata*. In *Proc. 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, vol. 2031 of *Lecture Notes in Computer Science*, pp. 189–203. Springer-Verlag, 2001.

[HSLL97] HAVELUND, K., SKOU, A., LARSEN, K. G., and LUND, K. *Formal Modelling and Analysis of an Audio/Video Protocol : An Industrial Case Study Using* UPPAAL. In *Proc. 18th IEEE Real-Time Systems Symposium (RTSS'97)*, pp. 2–13. IEEE Computer Society Press, 1997.

[HU79] HOPCROFT, J. E. and ULLMAN, J. D. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[JLS96] JENSEN, H. E., LARSEN, K. G., and SKOU, A. *Modelling and Analysis of a Collision Avoidance Protocol using* SPIN *and* UPPAAL. In *Proc. 2nd SPIN Verification Workshop on Algorithms, Applications, Tool Use, Theory*. American Mathematical Society, 1996.

[Kam68] KAMP, J. A. W. *Tense Logic and the Theory of Linear Order*. Ph.D. thesis, UCLA, Los Angeles, CA, USA, 1968.

[KF94] KAMINSKI, M. and FRANCEZ, N. *Finite-Memory Automata*. Theoretical Computer Science, vol. 134 :329–363, 1994.

[Kle56] KLEENE, S. C. *Representation of Events in Nerve Nets and Finite Automata*. In *Automata Studies*, pp. 3–41. Princeton University Press, 1956.

[KLL+97] KRISTOFFERSEN, K. J., LAROUSSINIE, F., LARSEN, K. G., PETTERSSON, P., and YI, W. *A Compositional Proof of a Real-Time Mutual Exclusion Protocol*. In *Proc. 7th International Joint Conference on Theory and Practice of Software Development (TAPSOFT'97)*, vol. 1214 of *Lecture Notes in Computer Science*, pp. 565–579. Springer-Verlag, 1997.

[KMTV00] KUPFERMAN, O., MADHUSUDAN, P., THIAGARAJAN, P., and VARDI, M. Y. *Open Systems in Reactive Environments : Control and Synthesis*. In *Proc. 11th International Conference on Concurrency Theory (CONCUR'00)*, vol. 1877 of *Lecture Notes in Computer Science*, pp. 92–107. Springer-Verlag, 2000.

[Koz83] KOZEN, D. *Results on the Propositional μ-Calculus*. Theoretical Computer Science, vol. 27 :333–354, 1983.

[KP95] KRISTOFFERSEN, K. J. and PETTERSSON, P. *Modelling and Analysis of a Steam Generator Using* UPPAAL. In *Proc. 7th Nordic Workshop on Programming Theory*. 1995.

[KV00] KUPFERMAN, O. and VARDI, M. Y. *μ-Calculus Synthesis*. In *Proc. 25th International Symposium on Mathematical Foundations of Computer Science (MFCS'2000)*, vol. 1893 of *Lecture Notes in Computer Science*, pp. 497–507. Springer-Verlag, 2000.

[Lar90] LARSEN, K. G. *Proof Systems for Satisfiability in Hennessy-Milner Logic with Recursion*. Theoretical Computer Science, vol. 72(2-3) :265–288, 1990.

[LBB+01] LARSEN, K. G., BEHRMANN, G., BRINKSMA, E., FEHNKER, A., HUNE, T., PETTERSSON, P., and ROMIJN, J. *As Cheap as Possible : Efficient Cost-Optimal Reachability for Priced Timed Automata*. In *Proc. 13th International Conference on Computer Aided Verification (CAV'01)*, vol. 2102 of *Lecture Notes in Computer Science*, pp. 493–505. Springer-Verlag, 2001.

[LL95] LAROUSSINIE, F. and LARSEN, K. G. *Compositional Model-Checking of Real-Time Systems*. In *Proc. 6th International Conference on Concurrency Theory (CONCUR'95)*, vol. 962 of *Lecture Notes in Computer Science*, pp. 529–539. Springer-Verlag, 1995.

[LL98] LAROUSSINIE, F. and LARSEN, K. G. *CMC : A Tool for Compositional Model-Checking of Real-Time Systems*. In *Proc. IFIP Joint International Conference Formal Description Techniques & Protocol Specification, Testing, and Verification (FORTE-PSTV'98)*, pp. 439–456. Kluwer Academic, 1998.

[LLPY97] LARSEN, K. G., LARSSON, F., PETTERSSON, P., and YI, W. *Efficient Verification of Real-Time Systems : Compact Data Structure and State-Space Reduction*. In *Proc. 18th IEEE Real-Time Systems Symposium (RTSS'97)*, pp. 14–24. IEEE Computer Society Press, 1997.

[LLW95] LAROUSSINIE, F., LARSEN, K. G., and WEISE, C. *From Timed Automata to Logic – and Back*. In *Proc. 20th International Symposium on Mathematical Foundations of Computer Science (MFCS'95)*, vol. 969 of *Lecture Notes in Computer Science*, pp. 27–41. Springer-Verlag, 1995.

[LPY95] LARSEN, K. G., PETTERSSON, P., and YI, W. *Compositional and Symbolic Model-Checking of Real-Time Systems*. In *Proc. 16th IEEE Real-Time Systems Symposium (RTSS'95)*, pp. 76–89. IEEE Computer Society Press, 1995.

[LPY97] LARSEN, K. G., PETTERSSON, P., and YI, W. UPPAAL *in a Nutshell*. Journal of Software Tools for Technology Transfer, vol. 1(1–2) :134–152, Oct. 1997.

[LT93] LEVESON, N. G. and TURNER, C. S. *Investigation of the Therac-25 Accidents*. IEEE Computer, vol. 26(7) :18–41, 1993.

[LWYP99] LARSEN, K. G., WEISE, C., YI, W., and PEARSON, J. *Clock Difference Diagrams*. Nordic Journal of Computing, vol. 6(3) :271–298, 1999.

[Mil89] MILNER, R. *Communication and Concurrency*. Prentice Hall International, 1989.

[Min67] MINSKY, M. *Computation : Finite and Infinite Machines*. Prentice Hall International, 1967.

[MP71] MCNAUGHTON, R. and PAPPERT, S. *Counter-Free Automata*. MIT Press, 1971.

[MP99] MUKHOPADHYAY, S. and PODELSKI, A. *Beyond Region Graphs : Symbolic Forward Analysis of Timed Automata*. In *Proc. 19th Conference on Foundations of Software Technology and Theoretical Computer Science (FST&TCS'99)*, vol. 1738 of *Lecture Notes in Computer Science*, pp. 232–244. Springer-Verlag, 1999.

[MY60] MCNAUGHTON, R. and YAMADA, H. *Regular Expressions and State Graphs for Automata*. IRE Trans. Electronic Computers, vol. EC-9 :39–47, 1960.

[NSV01] NEVEN, F., SCHWENTICK, T., and VIANU, V. *Towards Regular Languages over Infinite Alphabets*. In *Proc. 26th International Symposium on Mathematical Foundations of Computer Science (MFCS'01)*, vol. 2136 of *Lecture Notes in Computer Science*, pp. 560–572. Springer-Verlag, 2001.

[OY93] OLIVERO, A. and YOVINE, S. KRONOS *: a Tool for Verifying Real-Time Systems. User's Guide and Reference Manual*. VERIMAG, Grenoble, France, 1993.

[Par81] PARK, D. *Concurrency on Automata and Infinite Sequences*. In *CTCS'81*, vol. 104 of *Lecture Notes in Computer Science*, pp. 167–183. Springer-Verlag, 1981.

[Pet62] PETRI, C. *Kommunikation mit Automaten*. Ph.D. thesis, Institut für Instrumentelle Mathematik, Bonn, Germany, 1962.

[Pin86] PIN, J.-r. *Varieties of Formal Languages*. North Oxford, London et Plenum, New-York, 1986.

[Pin94] PIN, J.-r. *Logic on Words*. Bulletin of the European Association of Theoretical Computer Science, 1994. Vol. 54, pp. 145–165.

[Pin96] PIN, J.-r. *Logic, Semigroups and Automata on Words*. Annals of Mathematics and Artificial Intelligence, vol. 16 :343–384, 1996.

[Pnu77] PNUELI, A. *The Temporal Logic of Programs*. In *Proc. 18th IEEE Symposium on Foundations of Computer Science (FOCS'77)*, pp. 46–57. IEEE Computer Society Press, 1977.

[PP01] PERRIN, D. and PIN, J.-r. *Infinite Words*, 2001. Available on
`http://www.liafa.jussieu.fr/~jep/Resumes/InfiniteWords.html.`

[RS97a] RASKIN, J.-F. and SCHOBBENS, P.-Y. *State Clock Logic : a Decidable Real-Time Logic*. In *Proc. International Workshop on Hybrid and Real-Time Systems (HART'97)*, vol. 1201 of *Lecture Notes in Computer Science*, pp. 33–47. Springer-Verlag, 1997.

[RS97b] ROZENBERG, G. and SALOMAA, A., eds. *Handbook of Formal Languages*. Springer-Verlag, 1997.

[Rus01] RUSHBY, J. M. *Theorem Proving for Verification*. In *Proc. Modelling and Verification of Parallel Processes (*MOVEP*2k)*, vol. 2067 of *Lecture Notes in Computer Science*, pp. 39–57. Springer-Verlag, 2001.

[SBB+99] SCHNOEBELEN, P., BÉRARD, B., BIDOIT, M., LAROUSSINIE, F., and PETIT, A. *Vérification de logiciels : Techniques et outils de model-checking*. Vuibert, 1999.

[SBB+01] SCHNOEBELEN, P., BÉRARD, B., BIDOIT, M., LAROUSSINIE, F., and PETIT, A. *Systems and Software Verification - Model-Checking Techniques and Tools*. Springer-Verlag, 2001.

[Sch65] SCHÜTZENBERGER, M.-P. *On Infinite Monoids Having only Trivial Subgroups*. Information and Control, vol. 8 :190–194, 1965.

[SI94] STEFFEN, B. and INGÓLFSDÓTTIR, A. *Characteristic Formulae for Processes with Divergence*. Information and Computation, vol. 110(1) :149–163, 1994.

[Spe01] SPEAKMAN, T. *et al*. *PGM Specification*, 2001. Internet draft of the IETF (Internet Engineering Task Force), 115 pages.

[Tar55] TARSKI, A. *A Lattice-Theoretical Fixpoint Theorem and its Applications*. Pacific Journal of Mathematics, vol. 5 :285–309, 1955.

[Tho95] THOMAS, W. *On the Synthesis of Strategies in Infinite Games*. In *Proc. 12th Annual Symposium on Theoretical Aspects of Computer Science (STACS'95)*, vol. 900, pp. 1–13. Springer-Verlag, 1995.

[Tri98] TRIPAKIS, S. *L'analyse formelle des systèmes temporisés en pratique*. Ph.D. thesis, Université Joseph Fourier, Grenoble, France, Dec. 1998.

[TY96] TRIPAKIS, S. and YOVINE, S. *Analysis of Timed Systems Based on Time-Abstracting Bisimulations*. In *Proc. 8th International Conference on Computer Aided Verification (CAV'96)*, vol. 1102 of *Lecture Notes in Computer Science*, pp. 232–243. Springer-Verlag, 1996.

[TY01] TRIPAKIS, S. and YOVINE, S. *Analysis of Timed Systems using Time-Abstracting Bisimulations*. Formal Methods in System Design, vol. 18 :25–68, 2001.

[Wei99] WEIL, P. *Logique et automates, le théorème de Büchi (I)*, 1999. Cours de l'université de Bordeaux. Available on
`http://dept-info.labri.u-bordeaux.fr/ weil/cours_dea/notes_1.ps.`

[Wil94] WILKE, T. *Specifying Timed State Sequences in Powerful Decidable Logics and Timed Automata*. In *Proc. 3rd International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'94)*, vol. 863 of *Lecture Notes in Computer Science*, pp. 694–715. Springer-Verlag, Sep. 1994.

[Wol97] WOLPER, P. *Where could SPIN go next ? a Unifying Approach to Exploring Infinite State Spaces*. Slides for an invited talk at the *1997 SPIN Workshop*, 1997. Available on
`http://www.montefiore.ulg.ac.be/~pw/papers/talks.html.`

[WT94] WONG-TOI, H. *Symbolic Approximations for Verifying Real-Time Systems*. Ph.D. thesis, Stanford University, USA, Nov. 1994.

[WTH91] WONG-TOI, H. and HOFFMANN, G. *The Control of Dense Real-Time Discrete Event Systems*. In *Proc. 30th IEEE Conference on Decision and Control*, pp. 1527–1528. IEEE Computer Society Press, 1991.

[Yi90] YI, W. *Real-Time Behaviour of Asynchronous Agents*. In *Proc. 1st International Conference on Theory of Concurrency (CONCUR'90)*, vol. 458 of *Lecture Notes in Computer Science*, pp. 502–520. Springer-Verlag, 1990.

[Yi91] YI, W. *A Calculus of Real-Time Systems*. Ph.D. thesis, Chalmers University of Technology, Göteborg, Sweden, 1991.

[Yov97] YOVINE, S. KRONOS *: A Verification Tool for Real-Time Systems*. Journal of Software Tools for Technology Transfer, vol. 1(1–2) :123–133, Oct. 1997.

[Yov98] YOVINE, S. *Model-Checking Timed Automata*. In *School on Embedded Systems*, vol. 1494 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[YPD94] YI, W., PETTERSSON, P., and DANIEL, M. *Automatic Verification of Real-Time Communicating Systems by Constraint Solving*. In *Proc. 7th International Conference on Formal Description Techniques (FORTE'94)*, pp. 223–238. Chapman & Hall, 1994.