ENSC-2008 n° –

THÈSE

présentée à l'École Normale Supérieure de Cachan

pour obtenir le grade de

Docteur de l'École Normale Supérieure de Cachan

par : Arnaud SANGNIER

Spécialité : INFORMATIQUE

Vérification de systèmes avec compteurs et pointeurs

VERSION PRÉLIMINAIRE

Soutenue le 21 novembre 2008, devant un jury composé de :

– Alain FINKEL	co-directeur de thèse
– Yassine LAKHNECH	examinateur
– Étienne LOZES	co-directeur de thèse
– Andreas PODELSKI	rapporteur
– Pierluigi SAN PIETRO	examinateur
– Thuy NGUYEN	examinateur
– Marc ZEITOUN	rapporteur

Résumé

Au cours des dernières années, les méthodes formelles se sont avérées être une approche prometteuse pour garantir que le comportement d'un système informatique respecte une spécification donnée. Parmi les différentes techniques développées, le model-checking a été récemment étudié et appliqué avec succès à un grand nombre de modèles comme les systèmes à compteurs, les automates communicants (avec perte), les automates à pile, les automates temporisés, etc. Dans cette thèse, nous considérons deux modèles particuliers dans l'objectif de vérifier des programmes manipulant des variables entières et des variables de pointeurs. Dans une première partie, nous nous intéressons aux systèmes à compteurs. Nous commençons par définir ce modèle ainsi que ses différentes restrictions. Nous introduisons ensuite une sous-classe de systèmes à compteurs, appelée les machines à compteurs *reversal*-bornées, pour lesquelles de nombreux problèmes d'accessibilité sont décidables. Nous montrons que cette classe peut être étendue tout en gardant les résultats de décidabilité et nous prouvons qu'il est possible de décider si un Système d'Addition de Vecteurs avec États est *reversal*-borné, alors que cela n'est pas possible si l'on considère les systèmes à compteurs dans leur généralité. Nous finissons cette partie sur les systèmes à compteurs par l'étude de problèmes de model-checking de logiques temporelles. Les logiques temporelles que nous prenons en compte permettent de parler des données manipulées par le système. En particulier, nous montrons que le model-checking d'automates à un compteur déterministes avec des formules de la logique LTL avec registres est décidable, mais que cela n'est plus vrai lorsque l'hypothèse sur le déterminisme est supprimée. Dans une deuxième partie, nous introduisons le modèle des systèmes à pointeurs, qui est utilisé pour représenter des programmes manipulant des listes simplement chaînées. Nous donnons un algorithme qui traduit tout système à pointeurs en un système à compteurs qui lui est bisimilaire. Ceci nous permet de réutiliser les méthodes existantes pour l'analyse de systèmes à compteurs pour vérifier des programmes avec listes. Nous présentons ensuite une extension de la logique CTL* pour vérifier des propriétés temporelles sur de tels programmes, et nous étudions la décidabilité du problème de model-checking pour cette nouvelle logique. Finalement, dans une dernière partie, nous donnons une description de l'outil TOPICS (Translation of Programs Into Counter Systems) qui traduit un programme écrit dans un fragment syntaxique du langage C en un système à compteurs.

Mots-Clefs : Vérification formelle, Systèmes à compteurs, Programmes avec pointeurs, Logique temporelle, Model-checking

Abstract

In the past years, formal methods have shown to be a succesfull approach to ensure that the behavior of an informatic system will respect some properties. Among the different existing techniques, model-checking have been recently studied and successfully applied to a lot of models like counter systems, lossy channel systems, pushdown automata, timed automata, etc. In this thesis, we consider two different models to verify programs which manipulate integer variables and pointer variables. In a first part, we deal with counter systems. We define the model and the different restrictions which have been proposed. We then introduce a restricted class of counter systems, called the reversal-bounded counter machines, for which many reachability problems are decidable. We show that this class can be extended keeping the decidability results and we prove that we can decide whether a Vector Addition System with States is reversal-bounded or not, which is not possible for general counter systems. We then study the problem of model-checking counter systems with different temporal logics. The temporal logics we consider allow to speak about the data manipulated by the system. In particular, we show that the model-checking of deterministic one-counter automata with formulae of LTL with registers is decidable, and becomes undecidable when considering non deterministic one-counter automata and two counter automata. In a second part, we introduce the model of pointer systems, which is used to represent programs manipulating single linked lists. We propose an algorithm to translate any pointer system into a bisimilar counter system. This allows us to reuse existing techniques over counter systems to analyze these programs. We then propose an extension of CTL* to verify temporal properties for such programs, and we study the decidability of the model-checking problem for this new logic. Finally we present the tool TOPICS (Translation of Programs Into Counter Systems) which translates a C-like program with pointers and integer variables into a counter system.

Keywords : Formal verification, Counter systems, Programs with pointers, Temporal logic, Model-checking

Table des matières

Introduction

Pourquoi vérifier des systèmes informatiques ?

Dans notre société, les applications informatiques occupent une place prépondérante et de nombreux objets de la vie courante comme les téléphones portables, les automobiles, les fours, les machines à laver, etc, fonctionnent maintenant grâce à des programmes. Les systèmes informatiques sont également présents dans le matériel technologique de pointe comme dans les fusées, les centrales nucléaires ou encore les accélérateurs de particules. Or, il arrive que les programmes informatiques aient un comportement inattendu, on parle alors de bugs informatiques. Parmi les dysfonctionnements que l'on évoque le plus souvent, le bug qui a provoqué, le 4 juin 1996, la destruction de la fusée Ariane V arrive en tête. Cependant, il n'est pas nécessaire de remonter aussi loin dans le temps pour montrer que les bugs informatiques constituent un réel problème et force est de constater qu'ils sont toujours d'actualité. Ainsi :

– le 7 juin 2008, plusieurs bornes automatiques de la SNCF permettant de délivrer des billets acquis auparavant sur Internet furent victimes d'une défaillance informatique,
– le 27 août 2008, aux États-Unis, la défaillance d'un ordinateur dans le système de gestion des plans de vols a provoqué le retard de centaines d'avions dans les aéroports d'Atlanta, New-York, Washington et Chicago.

De tels bugs peuvent parfois mettre en danger des vies humaines, si par exemple le système de commande d'un avion tombait en panne, ou avoir des conséquences économiques dramatiques, comme ce fut le cas pour la fusée Ariane V. Aussi, afin d'éviter que de tels dysfonctionnements surviennent, des méthodes permettant de garantir le bon fonctionnement d'un système informatique sont développées, on parle alors de vérification de systèmes informatiques.

Contexte et besoins industriels d'EDF R&D

Cette thèse s'inscrit dans le cadre d'une collaboration entre le LSV et l'industriel EDF R&D. Les problématiques posées par EDF R&D rentrent dans le contexte de la sûreté de fonctionnement et de la sûreté de systèmes. La sûreté de fonctionnement consiste à démontrer qu'un système réalisera correctement les fonctions attendues et la sûreté d'un système consiste à garantir qu'il pourra éviter ou faire face à des situations dangereuses. Pour EDF R&D, ces problèmatiques de sûreté apparaissent à différents niveaux :

– au niveau des installations industrielles comme les centrales de production électrique, les réseaux de transport et de distribution,
– au niveau des composants principaux des installations industrielles comme les systèmes de contrôle-commande et les équipement électromécaniques,

– au niveau des composants élémentaires, comme les circuits intégrés et les cartes élémentaires.

La plupart des équipements électroniques présents dans les installations industrielles d'EDF, comme par exemple dans les systèmes de contrôle-commande, utilisent des programmes informatiques ; et sur le marché des équipements électroniques, on trouve actuellement de moins en moins de produits non programmés. Il est vrai que ces équipements électroniques programmés améliorent la sûreté de fonctionnement et la sûreté des installations industrielles car ils permettent de mettre en oeuvre des fonctionnalités avancées non réalisables avec des techniques non programmées. Cependant, la présence de tels équipements augmentent la complexité des systèmes et potentiellement le nombre de défauts résiduels de spécification et de conception. C'est précisément à ce niveau qu'il est nécessaire de développer des méthodes permettant de garantir le bon fonctionnement de programmes informatiques.

Afin de garantir la sûreté de fonctionnement des applications que l'on trouve dans les centrales nucléaires, EDF R&D souhaite être en mesure de vérifier que ces applications respectent des règles de conception et de construction approuvées par l'Autorité de Sûreté Nucléaire. Pour établir ces règles, les systèmes sont divisés en classes de sûreté. Il existe trois classes de sûreté :

1. La classe A : il s'agit de la classe la plus exigeante, les équipements qui appartiennent à cette classe sont souvent conçus spécifiquement pour le nucléaire,
2. la classe B : cette classe est également très exigeante, mais elle permet l'utilisation d'équipements non spécifiques au nucléaires,
3. la classe C : cette classe est relativement peu exigeante.

Les différents systèmes sont répartis à travers ces classes selon le rôle qu'ils jouent au sein de la centrale nucléaire. Pour chaque classe de sûreté, des règles de conception et de construction spécifiques sont définies.

En ce qui concerne les programmes informatiques, ceux qui se trouvent dans la classe A sont les moins difficiles à vérifier pour les raisons suivantes :

– EDF R&D possède les codes sources,
– il existe une documentation technique détaillée,
– ces programmes sont de taille raisonnable et sont écrits en suivant des règles de conception très strictes qui permettent d'éliminer systématiquement certains types de défauts. Par exemple, l'utilisation de mécanismes d'allocation dynamique de la mémoire y est interdite.

Pour garantir le bon fonctionnement des logiciels de la classe A, EDF R&D utilise principalement l'outil de vérification formelle `Polyspace` [Pol] qui permet de vérifier si le programme donné en entrée ne réalise pas d'erreur comme des débordements arithmétiques sur les entiers ou encore l'accès hors bornes d'un tableau. Notons que ce logiciel est également utilisé par d'autres industriels, comme par exemple Airbus, pour vérifier des logiciels embarqués dont la conception suit également des règles très strictes.

Les logiciels appartenant à la classe B sont quant à eux beaucoup plus difficiles à vérifier car il s'agit de programmes de taille plus importante, la documentation technique est moins accessible, et de plus les règles de conception pour ces logiciels sont moins strictes. En particulier les programmes de la classe B peuvent utiliser des mécanismes d'allocation dynamique de la mémoire. Pour garantir le bon fonctionnement de logiciels de catégorie B, EDF R&D souhaite développer des méthodes de

vérification sur des modèles représentatifs du code source, et c'est exactement là qu'interviennent les travaux sur la vérification développés dans cette thèse.

Différentes méthodes pour garantir le bon fonctionnement d'un système

Une approche classique pour vérifier qu'un programme fonctionne correctement consiste à réaliser un certain nombre de tests sur ce programme. Cependant, cette méthode pose différents problèmes. Tout d'abord, elle ne garantit pas que toutes les exécutions possibles du système ont été testées, en d'autres termes elle n'est pas exhaustive. De plus, les spécifications que doivent vérifier les systèmes sont souvent données de façon informelle et peuvent ainsi donner lieu à des imprécisions au niveau de leur interprétation.

Une solution pour palier à ce problème consiste à utiliser des méthodes formelles pour vérifier les systèmes informatiques. Le principe de ces méthodes est qu'elles se basent sur des formalismes mathématiques afin de donner une preuve formelle du bon fonctionnement des systèmes. Parmi ces méthodes, on distingue notamment :

– *La génération automatique de tests* : à partir d'une spécification fournie dans un langage formel, cette méthode consiste à générer automatiquement, en utilisant des outils mathématiques, un ensemble fini de tests, qui va être couvrant. Si pour chacun de ces tests, la spécification est vérifiée, alors on est sûr qu'elle est vérifiée pour l'ensemble du système. Une des difficultés est de trouver un ensemble de tests couvrants, ce qui n'est pas toujours possible.
– *La démonstration automatique* : cette méthode consiste à prouver de façon automatique que le système vérifie des propriétés données. Pour ce faire, elle construit une preuve du bon fonctionnement du système en utilisant des règles de déduction. Cependant, l'automatisation n'est pas toujours possible et l'aide humaine peut alors s'avérer nécessaire pour faire aboutir le processus de déduction.
– *L'interprétation abstraite* [CC77] : à partir d'une description formelle de la sémantique d'un langage de programmation, cette technique calcule des surapproximations des différentes exécutions d'un programme en utilisant des abstractions. Elle s'est avérée efficace pour vérifier de nombreux systèmes. L'algorithme de l'outil `Polyspace` , que nous avons mentionné plus tôt, utilise cette technique. Un des inconvénients de cette méthode est que comme elle calcule des surapproximations du comportement du programme, il se peut que parfois elle ne sache pas dire si une propriété est vérifiée ou non. Il faut alors raffiner les abstractions, ce qui peut s'avérer ardu.
– *Le model-checking* : il consiste à encoder le système dans un modèle mathématique ainsi que la propriété à vérifier, et ensuite à utiliser des méthodes mathématiques pour s'assurer que le modèle satisfait la propriété énoncée. L'avantage est qu'une fois que le modèle est extrait du système, cette technique est entièrement automatique, toutefois, il faut souvent faire des compromis sur l'expressivité du modèle et du langage de spécification pour pouvoir obtenir des algorithmes de vérification.

Remarquons que chacune de ces méthodes a ses avantages et ses inconvénients, et le plus souvent elles sont complémentaires. Il est aussi possible de coupler l'utilisation de ces techniques, ainsi de récents travaux utilisent l'interprétation abstraite sur des modèles issus du model-checking, un modèle pouvant en effet être vu comme une définition particulière de la sémantique d'un programme. Les travaux réalisés dans cette thèse rentrent dans le cadre de la vérification formelle par model-checking.

Le model-checking : une technique de vérification

Dans les dernières années, le model-checking a été beaucoup étudié et de nombreuses classes de modèles ont été proposées, citons par exemple les réseaux de Petri [Pet62], les automates à pile, les automates temporisés [AD94] et bien d'autres. Suivant les systèmes à analyser et les propriétés à vérifier, on choisira une classe de modèles plutôt qu'une autre. De plus, de nombreux travaux concernant les langages de spécification ont été réalisés, de façon à obtenir des formalismes permettant de décrire formellement des propriétés. Signalons qu'en 2007, le prix Turing a été décerné à Edmund Clarke, E. Allen Emerson et Joseph Sifakis pour leurs travaux sur le model-checking.

La méthode générale du model-checking travaille sur des modèles. Lorsque l'on fait du model-checking il faut donc établir :

1. Une classe de modèles permettant de décrire les systèmes étudiés,
2. Un langage de spécification formelle pour pouvoir exprimer des propriétés sur les modèles.

Le model-checking consiste ensuite à établir des algorithmes permettant de décider si étant donné un modèle M et une propriété ϕ écrite dans un langage de spécification, le modèle satisfait la propriété (ce qui est généralement noté $M \models \phi$). Remarquons que selon la classe de modèles et le langage de spécification pris en compte, il n'existe pas toujours d'algorithme permettant de résoudre le problème de model-checking. Ainsi le problème de l'arrêt des machines de Turing peut-être vu comme un cas particulier de problème de model-checking pour lequel la classe de modèles pris en compte sont les machines Turing et la propriété à vérifier est l'arrêt. Or ce problème est connu comme étant indécidable, c'est-à-dire qu'il n'existe pas un algorithme prenant en entrée une machine de Turing et qui répond oui si elle termine et non sinon. En revanche, lorsqu'il existe un algorithme permettant de répondre au problème du model-checking, un des aboutissements de cette méthode de vérification est l'implémentation dans un model-checker. Parmi les model-checkers les plus connus, on peut citer `SPIN` [Spi] qui permet de vérifier des propriétés exprimées en logique temporelle sur des modèles décrits dans le langage PROMELA, ou encore `UPPAAL` [Upp] qui permet d'analyser des automates temporisés.

Des systèmes finis et infinis

Une des principales difficultés lorsque l'on fait du model-checking est de trouver des classes de modèles et des langages de spécification pour lesquels les problèmatiques de vérification sont décidables. Lorsque l'on manipule un modèle, on parle d'exécutions et de configurations, les exécutions représentent les différents comportements possibles du modèle, quant aux configurations elles correspondent aux différents états dans lesquels peut se trouver le modèle au cours d'une exécution. On peut alors distinguer deux classes de modèles. Les systèmes finis qui correspondent à des modèles pour lesquels le nombre de configurations accessibles est borné et les systèmes infinis qui peuvent avoir une infinité de configurations accessibles. Savoir si, dans un modèle, une configuration est accessible ou non constitue un des problèmes de base du model-checking.

Dans le cas des systèmes finis, ce problème est décidable, il suffit en effet de calculer pas à pas l'ensemble des configurations accessibles et comme il n'y en a qu'un nombre fini, ce calcul termine. Le principal problème que l'on doit affronter est la représentation des ensembles de configurations qui

bien que finis peuvent être très grands (on parle alors du problème de l'explosion combinatoire).

Pour les systèmes infinis, la situation est différente, car le calcul pas à pas des configurations accessibles ne termine pas. De plus, pour la plupart de ces systèmes, les problèmes d'accessibilité sont indécidables. Il existe cependant des classes de systèmes infinis pour lesquels ce problème est décidable. Parmi les classes de systèmes infinis ayant un problème d'accessibilité décidable, nous pouvons par exemple citer les automates temporisés, les réseaux de Petri [Kos82, May84], les automates à pile ou encore les automates communicants avec perte [Fin94, CFP96, AJ96] (pour la version sans perte ce problème est indécidable). Une méthode consiste à trouver un moyen de calculer en un nombre fini d'étapes des ensembles possiblement infinis de configurations accessibles, on parle d'accélération. Pour mettre en place une telle méthode, il est nécessaire d'utiliser des représentations symboliques permettant de manipuler ces ensembles infinis.

Dans cette thèse nous nous intéresserons à deux classes particulières de systèmes infinis :

1. les systèmes à compteurs, qui sont des automates finis étendus avec des opérations permettant de manipuler des variables entières,
2. les systèmes à pointeurs, dont nous nous servons pour modéliser des programmes travaillant sur des listes simplement chaînées, et qui eux manipulent des variables de pointeurs.

Nous verrons que pour chacune de ces classes de systèmes, il est possible de définir des représentations symboliques.

Les logiques temporelles comme langage de spécification

Un enjeu du model-checking réside dans la définition de langages formels de spécification. Ainsi, en 1977, dans [Pnu77], Pnueli introduisit la logique temporelle linéaire LTL pour exprimer des propriétés sur les exécutions de structures de Kripke. Les structures de Kripke sont des graphes dans lesquels on associe à chaque noeud un ensemble fini de variables propositionnelles. Grâce à des opérateurs spécifiques, appelés opérateurs temporels, cette logique permet d'énoncer des propriétés du style :

– il existe une exécution arrivant dans un état vérifiant la propriété A,
– il existe une exécution passant infiniment souvent par une configuration satisfaisant la propriété A,
– il existe une exécution dont toutes les configurations vérifient la propriété A.

Pnueli montra que le problème de savoir si une structure de Kripke satisfait une formule de LTL est décidable. Cependant la logique LTL ne permet pas d'exprimer les différentes possibilités d'évolution d'un système à partir d'une configuration. Les exécutions possibles du modèle sont en effet chacune considérées de façon distincte. En 1982, dans [CE82], Clarke et Emerson introduisirent une autre logique temporelle, la logique temporelle branchante CTL , qui elle permet d'exprimer les différentes possibilités qu'à une exécution d'évoluer à un moment donné tout en utilisant des opérateurs temporels similaires à ceux de LTL . Là encore, les auteurs montrèrent que le problème de model-checking de formules de CTL sur des structures de Kripke est décidable. Les logiques LTL et CTL ayant une expressivité incomparable, la logique CTL* fut finalement introduite en 1983 dans [EH83] et les logiques LTL et CTL correspondent à des fragments de cette dernière logique.

Dans cette thèse, nous allons nous intéresser à des extensions de la logique CTL* pour caractériser les exécutions de systèmes à compteurs et à pointeurs. Les logiques temporelles que nous proposons permettent de décrire les exécutions de ces systèmes en parlant également de leurs configurations,

qui nous le rappelons appartiennent à des ensembles infinis. En effet, la logique CTL* utilisant uniquement des variables propositionnelles comme propositions atomiques ne permet pas, par exemple, d'écrire des spécifications prenant en compte des valeurs de compteurs. Or, de façon à pouvoir spécifier le plus précisément possible des propriétés sur des systèmes infinis, il est nécessaire d'ajouter à cette logique des mécanismes permettant de décrire les évolutions des configurations au cours du temps.

Contributions de la thèse

Comme nous l'avons dit, dans cette thèse nous nous intéressons à la vérification par model-checking de deux classes de systèmes infinis particulières, les systèmes à compteurs et les systèmes à pointeurs. Pour chacune de ces classes, nous proposons dans un premier temps des méthodes de vérification permettant de résoudre des problèmes d'accessibilité. Dans un deuxième temps nous définissons des formalismes logiques basés sur les logiques temporelles et permettant de décrire les exécutions de ces systèmes tout en fournissant des mécanismes pour spécifier des contraintes sur les configurations. Nous étudions alors la décidabilité de différents problèmes de model-checking mettant en jeu ces logiques.

Cette thèse est organisée de la façon suivante :

Première partie. Cette partie concerne l'étude des systèmes à compteurs. Au chapitre 1, nous commençons par donner la définition des systèmes à compteurs qui correspondent à des automates finis dont les transitions sont étiquetées par des formules de l'arithmétique de Presburger manipulant des variables entières. Nous rappelons aussi la définition de différentes sous-classes de systèmes à compteurs dignes d'intérêt comme les systèmes à compteurs Presburger-linéaires, les machines à compteurs et les Systèmes d'Addition de Vecteurs avec États (SAVE) qui sont équivalents aux réseaux de Petri. Nous introduisons ensuite les différents problèmes d'accessibilité que nous étudierons et nous rappelons les résultats d'indécidabilité concernant les machines de Minsky à deux compteurs et les résultats de décidabilité pour les SAVE et les machines à un compteur. Ce chapitre se termine par la présentation des résultats obtenus pour les systèmes à compteurs Presburger-linéaires plats et aplatissables, et en particulier nous décrivons le fonctionnement de l'outil FAST qui est utilisé pour calculer l'ensemble d'accessibilité de ces systèmes.
Dans [Iba78], Ibarra introduisit une classe de systèmes à compteurs, appelée les machines à compteurs *reversal*-bornées ("reversal-bounded" en anglais), caractérisées par le fait que, dans toute exécution, chaque compteur ne réalise qu'un nombre borné d'alternances entre les phases de croissance et de décroissance. Il prouva alors que les machines à compteurs *reversal*-bornées ont un ensemble d'accessibilité semi-linéaire qui peut effectivement être calculé. Dans le chapitre 2, nous étendons la classe des machines à compteurs *reversal*-bornées en considérant des machines pour lesquelles, dans toute exécution, chaque compteur réalise un nombre borné d'alternances au-dessus d'une borne donnée, en revanche le nombre d'alternances réalisées au-dessous de cette borne peut lui être infini. Nous prouvons que les machines vérifiant cette propriété ont toujours un ensemble d'accessibilité semi-linéaire. Nous montrons de plus que la propriété d'être *reversal*-bornée (en considérant la nouvelle définition) est indécidable en général, même dans les cas où une des bornes est fixée, cependant cette propriété devient décidable lorsque les deux bornes sont fixées ou lorsque l'on considère des SAVE.
Au chapitre 3, nous étudions des problèmes de model-checking sur des systèmes à compteurs. Dans un premier temps, nous rappelons la définition de la logique FOCTL*(Pr) introduite dans [DFGD06],

qui permet de spécifier des exécutions de systèmes à compteurs en utilisant des formules de Presburger comme proposition atomiques. Nous montrons alors que le problème de model-checking de formules de FOCTL*(Pr) est indécidable pour les machines à compteurs *reversal*-bornés, mais qu'en revanche le problème existentiel de l'accessibilité répétée d'un ensemble de configurations définissable dans l'arithmétique de Presburger est décidable pour cette classe de systèmes. Nous étudions ensuite, pour des automates à un compteur, des problèmes de model-checking de formules de logiques utilisant un mécanisme de stockage. Nous considérons deux logiques, la logique LTL avec registres introduite dans [DLN05] et la logique du premier ordre sur les mots de données. Dans de récents travaux [BMS$^+$06, DL06, DLN07], les problèmes de satisfiabilité ont été étudiés pour ces deux logiques. Dans cette thèse, l'idée est d'utiliser ces deux logiques pour caractériser les exécutions d'automates à un compteur. Ces exécutions peuvent en effet être vues comme des mots de données, l'alphabet étant l'ensemble des états de contrôle de l'automate et les données correspondent aux valeurs prises par l'unique compteur. Nous prouvons alors que le problème de model-checking pour chacune de ces deux logiques est PSPACE-complet dans le cas d'automates à un compteur déterministes . Nous montrons que ce problème devient indécidable lorsque les automates considérés ne sont plus déterministes, même lorsque les formules n'utilisent qu'un seul registre ou deux variables. Ces résultats d'indécidabilité contrastent avec le fait que de nombreux problèmes de vérification sont décidables sur les automates à un compteur.

Deuxième partie Dans cette partie, nous étudions les systèmes à pointeurs. Au chapitre 4, nous donnons la définition de cette classe de systèmes et nous expliquons comment ils modélisent des programmes manipulants des listes simplement chaînées. L'idée principale consiste à représenter le contenu de la mémoire comme un graphe dans lequel chaque noeud a au plus un successeur. Les configurations des systèmes à pointeurs sont alors des couples composés d'un état de contrôle et d'un tel graphe. Nous définissons de plus différentes problèmatiques de vérification sur ces systèmes à pointeurs, dont certaines sont spécifiques aux programmes manipulant dynamiquement la mémoire, comme l'absence d'erreur de segmentation ou l'absence de fuite mémoire. Nous présentons également les états mémoire symboliques, qui sont une représentation symbolique permettant de manipuler des ensembles infinis de configurations de systèmes à pointeurs.

Au chapitre 5, nous montrons qu'à partir d'un système à pointeurs, on peut construire un système à compteurs qui lui est bisimilaire. Nous donnons l'algorithme de cette traduction et nous expliquons comment utiliser les méthodes déjà existantes pour la vérification de systèmes à compteurs, en particulier l'algorihme de l'outil FAST , pour vérifier des systèmes à pointeurs.

Dans le chapitre 6, nous introduisons la logique CTL$^*_{mem}$ qui est une extension de la logique CTL* permettant de spécifier les exécutions de systèmes à pointeurs. Dans cette logique, nous pouvons utiliser les états mémoire symboliques comme propositions atomiques, ce qui permet de décrire les évolutions des configurations. Nous étudions ensuite la décidabilité de différents problèmes de vérification sur ces systèmes à pointeurs. Nous montrons ainsi que dans le cas de systèmes à pointeurs plats sans mise à jour destructive (c'est à dire sans opération modifiant la forme du graphe mémoire) et munis d'une configuration initiale acyclique, le problème de model-checking de formules de CTL$^*_{mem}$ est décidable. Nous prouvons également que dans le cas de systèmes à pointeurs plats sans mise à jour destructive et sans test d'alias, les problèmes d'accessibilité d'une erreur de segmentation et d'une fuite mémoire sont décidables, mais qu'en revanche le problème de model-checking de formules de CTL* est indécidable. Nous finissons par montrer que dans le cas général de programmes plats sans mise à jour destructive, les problèmes d'accessibilité d'une erreur de segmentation ou d'une fuite mémoire sont indécidables. Nous étendons ainsi et complètons les résultats de [BI07] sur la décidabilité

de problèmes de vérification dans le cas de systèmes à pointeurs plat.

Troisième partie Dans cette dernière partie, nous présentons l'outil TOPICS (Translation of Programs Into Counter Systems) qui implante l'algorithme de traduction décrit au chapitre 5. Cet outil prend en entrée une fonction écrite dans un restriction syntaxique du langage C et un fichier définissant la configuration initiale et il produit un système à compteurs au format de l'outil FAST . Il est ensuite possible de lancer FAST sur le système à compteurs obtenu pour vérifier si le programme considéré ne réalise pas d'erreur.

Première partie

Vérification de systèmes à compteurs

Chapitre 1

Systèmes à compteurs

Dans ce chapitre, nous introduisons les notions mathématiques dont nous nous servirons tout au long de cette thèse en focalisant plus particulièrement sur l'arithmétique de Presburger qui est une logique souvent utilisée en vérification, du fait qu'elle a un problème de satisfiabilité décidable.

Nous présentons ensuite la classe des systèmes à compteurs, qui correspondent au premier modèle sur lequel nous allons porter notre attention. Longtemps, en vérification, le comportement d'un système a été représenté par un automate fini, chaque état de l'automate correspondant à une configuration du système. De façon à modéliser des systèmes plus complexes, avec un nombre possiblement infini de configurations, les automates finis ont été augmentés avec un ensemble de variables entières, chaque transition réalisant des opérations sur ces variables, le modèle ainsi obtenu étant les systèmes à compteurs.

Nous présentons aussi dans ce chapitre les différents problèmes de vérification que nous souhaitons explorer, et donnons quelques résultats d'indécidabilité et de décidabilité dans le cadre des systèmes à compteurs. Finalement, nous rappelons le contexte théorique qui a mené à l'élaboration de l'outil FAST qui permet d'analyser des systèmes à compteurs.

1.1 Préliminaires

1.1.1 Notions mathématiques

Nombres L'ensemble des entiers positifs est noté $\mathbb{N}$ et l'ensemble des entiers strictement positifs est noté $\mathbb{N}^*$. L'ensemble des entiers relatifs est noté $\mathbb{Z}$. La relation d'ordre totale habituelle sur $\mathbb{Z}$ est notée $\leq$. Nous notons $\mathbb{N}_\omega$, l'ensemble $\mathbb{N} \cup \{\omega\}$ où ω est un symbole tel que $\omega \notin \mathbb{N}$ et pour tout $k \in \mathbb{N}_\omega$, $k \leq \omega$. Notons que $(\mathbb{N}_\omega, \leq)$ est encore total. Nous étendons les opérations classiques $+$ et $-$ à $\mathbb{N}_\omega$ de la façon suivante : pour tout $k \in \mathbb{Z},\ k + \omega = \omega$. Pour $k, l \in \mathbb{N}$ avec $k \leq l$, l'ensemble des entiers compris entre k et l est noté $[k..l] = \{i \in \mathbb{N} \mid k \leq i \leq l\}$.

Ensembles Pour deux ensembles E et F, on note $E \cap F$, $E \cup F$, $E \uplus F$, $E \backslash F$ et $E \times F$ respectivement l'intersection, l'union, l'union disjointe, la différence et le produit cartésien. On note $E \subseteq F$ si E est un sous-ensemble de F. L'ensemble vide est noté $\emptyset$. Le cardinal d'un ensemble fini F est noté $|F| \in \mathbb{N}$. Étant donné un ensemble E, nous notons $\mathcal{P}(E)$ l'ensemble des sous-ensembles de E, c'est-à-dire l'ensemble $\{F \mid F \subseteq E\}$.

Relations Étant donnés deux ensembles E et F, une relation R dans $E \times F$ est un sous-ensemble $R \subseteq E \times F$. La composée d'une relation R dans $E \times F$ et d'une relation R' dans $F \times G$ est la relation $R.R'$ dans $E \times G$ définie par $(e, e'') \in R.R'$ si il existe $e' \in E'$ tel que $(e, e') \in R$ et $(e', e'') \in R'$.
Une relation binaire sur un ensemble E est une relation dans $E \times E$. La relation Id_E sur E est la relation binaire définie par $Id_E = \{(e, e) \mid e \in E\}$. Étant donnée une relation binaire R sur E, pour $k \geq 1$, on note R^k la relation binaire sur E définie par l'induction $R^k = R.R^{(k-1)}$ et par convention $R^0 = Id_E$. La fermeture réflexive et transitive d'une relation binaire R sur E est la relation R^* définie par $R^* = \bigcup_{k \geq 0} R^k$.

Fonctions Une fonction $f : D \to E$ est une relation dans $D \times E$ telle que pour tout $d \in D$, il existe au plus un élément $e \in E$ vérifiant $(d, e) \in f$. On note $f(d) = e$ cet élément et E^D l'ensemble des fonctions de D vers E. Étant donnée une fonction $f : D \to E$, on définit le domaine **dom** de la façon suivante : $\mathbf{dom}(f) = \{d \in D \mid \exists e \in E \text{ tel que } e = f(d)\}$. On dit qu'une fonction $f' : D' \to E'$ étend une fonction $f : D \to E$ si $D \subseteq D'$ et $E \subseteq E'$ et si pour tout $d \in D$, on a $f'(d) = f(d)$. La restriction d'une fonction $f : D \to E$ à $D_0 \subseteq D$ est la fonction notée $f_{|D_0} : D_0 \to Y$ définie par $\mathbf{dom}(f_{|D_0}) = D_0 \cap \mathbf{dom}(f)$ et pour tout $x \in \mathbf{dom}(f_{|D_0})$, on a $f_{|D_0}(x) = f(x)$. Une fonction $f : D \to E$ est une fonction totale si $\mathbf{dom}(f) = D$, dans le cas contraire on parle de fonction partielle. Dans la suite, nous supposerons que toutes les fonctions sont totales sauf dans les cas où nous signalerons explicitement que la fonction est partielle. Étant donnés une fonction $f : D \to E$ et un ensemble $E' \subseteq E$, nous notons $f^{-1}(E')$ le sous-ensemble de D définie de la façon suivante : $f^{-1}(E') = \{d \in D \mid f(d) \in E'\}$ et de la même façon si $D' \subseteq D$, nous définissons $f(D') = \{f(d) \mid d \in D'\}$. Nous dirons qu'une fonction $f : D \to E$ est :
– injective, si pour tout $d, d' \in D$, $f(d) = f(d')$ implique $d = d'$,
– surjective, si pour tout $e \in E$, il existe $d \in D$ tel que $f(d) = e$,
– bijective, si f est injective et surjective.

Vecteurs Un vecteur $\mathbf{v}$ à $n \geq 1$ composantes (ou de dimension n) dans un ensemble E est un élément de E^n. Soit $\mathbf{v} \in E^n$, pour tout $i \in [1..n]$, nous notons $\mathbf{v}(i) \in E$ la i-ème composante de $\mathbf{v}$. Il nous arrivera parfois pour des vecteurs de E^n de numéroter les composantes de 0 à $n-1$. Pour des vecteur dans $\mathbb{Z}^n$, nous notons $\leq$ l'ordre défini composante par composante de la façon suivante, $\mathbf{v} \leq \mathbf{v}'$ si et seulement si pour tout $i \in [1..n]$, $\mathbf{v}(i) \leq \mathbf{v}'(i)$. Nous notons $\mathbf{0}$ le vecteur dont toutes les composantes sont égales à 0. Nous étendons aussi les opérations $+$ et $-$ aux vecteurs dans $\mathbb{Z}^n$, ainsi pour tout $\mathbf{v}, \mathbf{v}' \in \mathbb{Z}^n$, $\mathbf{v} + \mathbf{v}'$ est le vecteur défini par : pour tout $i \in [1..n]$, $(\mathbf{v} + \mathbf{v}')(i) = \mathbf{v}(i) + \mathbf{v}'(i)$ et $\mathbf{v} - \mathbf{v}'$ est le vecteur défini par : pour tout $i \in [1..n]$, $(\mathbf{v} - \mathbf{v}')(i) = \mathbf{v}(i) - \mathbf{v}'(i)$.
Soit $X = \{x_1, \ldots, x_n\}$ un ensemble fini de n variables. Nous confondrons parfois l'ensemble $\mathbb{N}^X$ des fonctions associant à chaque variable une valeur entière avec l'ensemble $\mathbb{N}^n$ des vecteurs de dimension n.

Ensembles fermés par le haut Soit $n \in \mathbb{N}^*$. Un ensemble $E \subseteq \mathbb{N}^n$ de vecteurs de dimension n est fermé par le haut si et seulement si pour tout élément $\mathbf{v} \in E$, si $\mathbf{v}' \in \mathbb{N}^n$ satisfait $\mathbf{v} \leq \mathbf{v}'$ alors $\mathbf{v}' \in E$. Comme la relation d'ordre $\leq$ sur $\mathbb{N}^n$ est un bel ordre, d'après [Hig52], nous avons le lemme suivant sur les sous-ensembles de $\mathbb{N}^n$ fermés par le haut.

Lemme 1.1 *Soit $n \in \mathbb{N}^*$. Tout sous ensemble de $\mathbb{N}^n$ fermé par le haut a un nombre fini d'éléments minimaux.*

Matrices Étant donnés deux entiers $m \geq 1$ et $n \geq 1$, on note $\mathcal{M}_{m,n}(E)$ l'ensemble des matrices à m lignes et n colonnes et à valeurs dans E. Soit $A = (A_{i,j})$ une matrice de $\mathcal{M}_{m,n}(E)$, pour tout $i \in [1..m]$ et pour tout $j \in [1..n]$, $A_{i,j}$ représente le coefficient de A en i-ème ligne et j-ème colonne.
Une matrice carrée de taille $n \geq 1$ est une matrice à n lignes et n colonnes. L'ensemble des matrices carrées $\mathcal{M}_{n,n}(E)$ est noté $\mathcal{M}_n(E)$.
Étant données deux matrices $A \in \mathcal{M}_{m,n}(\mathbb{Z})$ et $B \in \mathcal{M}_{n,l}(\mathbb{Z})$ avec $k, l, n \in \mathbb{N}$, on définit la matrice $C = A.B$ appartenant à $\mathcal{M}_{m,l}$ de la façon suivante : pour tout $i \in [1..m]$, et pour tout $j \in [1..l]$, $C_{i,j} = \Sigma_{k\in[1..n]} A_{i,k}.B_{k,j}$. Pour $n \in \mathbb{N}$, nous notons de plus $\mathcal{I}d_n \in \mathcal{M}_n(\mathbb{Z})$ la matrice $(A_{i,j})$ telle que pour tout $i \in [1..n]$, $A_{i,i} = 1$ et pour tout $i, j \in [1..n]$ vérifiant $i \neq j$, $A_{i,j} = 0$.
Les vecteurs de $\mathbb{Z}^n$ sont identifiés aux matrices de $\mathcal{M}_{n,1}(\mathbb{Z})$. Ainsi, pour une matrice $A \in \mathcal{M}_{m,n}(\mathbb{Z})$ et un vecteur $\mathbf{v} \in \mathbb{Z}^n$, on note $A.\mathbf{v}$ le vecteur de $\mathbb{Z}^m$ défini par : pour tout $i \in [1..m]$, $(A.\mathbf{v})(i) = \Sigma_{k=1}^{n} A_{i,k}.\mathbf{v}(k)$.

Monoïdes Un monoïde $(E, \cdot, e)$ est un triplet tel que E est un ensemble non vide, $\cdot : E \times E \to E$ est une fonction associative, ie pour tout $x, y, z \in E$, $(x \cdot y) \cdot z = x \cdot (y \cdot z)$ et e est un élément neutre, ie pour tout $x \in E$, $x \cdot e = e \cdot x = x$. Étant donné un alphabet fini Σ, on note Σ^* le monoïde sur Σ muni de la concaténation et de l'élément neutre ϵ. Σ^* est aussi appelé l'ensemble des mots finis sur Σ et ϵ le mot vide.
Un sous-ensemble de Σ^* est aussi appelé un langage. Nous nous servirons parfois de notions sur l'ensemble des langages réguliers, qui sont les langages reconnus par des automates finis, mais nous ne détaillons pas plus ici les résultats bien connus concernant cette classe de langages et qui sont disponibles dans [HU79].
Soit $n \in \mathbb{N}^*$. Le monoïde engendré multiplicativement par une matrice carrée $M \in \mathcal{M}_n(\mathbb{Z})$ est défini par $< M >= \{M^i \mid i \in \mathbb{N}\}$ avec par convention $M^0 = \mathcal{I}d_n$. Et pour un ensemble de matrices carrées $\mathcal{M} \subseteq \mathcal{M}_n(\mathbb{Z})$, le monoïde engendré par $\mathcal{M}$ est défini par $< \mathcal{M} >= \bigcup_{i\geq 0}\{M_1.M_2.\ldots.M_i \mid \forall j \in [1..i], M_j \in \mathcal{M}\}$.

Ensembles semi-linéaires Soit $n \in \mathbb{N}^*$. Un ensemble $E \subset \mathbb{N}^n$ est linéaire si il existe $k+1$ vecteurs $\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_k \in \mathbb{N}^n$ tel que $E = \{\mathbf{v} \in \mathbb{N}^n \mid \mathbf{v} = \mathbf{v}_0 + \lambda_1.\mathbf{v}_1 + \ldots + \lambda_k.\mathbf{v}_k$ avec $\lambda_i \in \mathbb{N}$ pour tout $i \in [1..n]\}$. Un ensemble semi-linéaire est une union finie d'ensembles linéaires.

Image de Parikh Étant donné un alphabet $\Sigma = \{a_1, \ldots, a_n\}$ à n lettres. Nous définissons la fonction $f_\Sigma : \Sigma^* \to \mathbb{N}^n$ qui à chaque mot $w \in \Sigma^*$ associe le vecteur $f_\Sigma(w)$ tel que pour tout $i \in [1..n]$, $f_\Sigma(w)(i) = \#a_i(w)$ où $\#a_i(w)$ représente le nombre d'occurences de a_i dans le mot w. Remarquons que $f_\Sigma(\epsilon) = \mathbf{0}$. Nous étendons cette fonctions aux langages inclus dans Σ^*, ainsi, pour un langage $L \subseteq \Sigma^*$, $f_\Sigma(L) = \bigcup_{w\in L}\{f_\Sigma(w)\}$. $f_\Sigma(L) \subseteq \mathbb{N}^n$ est appelé l'image de Parikh du langage L. Nous rappelons le lemme de Parikh pour les langages réguliers, qui est aussi valable pour la classe des langages hors-contexte :

Lemme 1.2 *[Par66] L'image de Parikh d'un langage régulier est un ensemble semi-linéaire.*

De plus, la preuve fournie dans [Par66], permet de construire effectivement l'image de Parikh d'un langage régulier.

1.1.2 L'arithmétique de Presburger

L'arithmétique de Presburger est la théorie du premier ordre sur les entiers avec l'addition mais pas la multiplication. L'ensemble des formules de l'arithmétique de Presburger peut être décrit par la grammaire suivante où t représente un terme et ϕ une formule :

$$\begin{array}{rcl} t & ::= & 0 \mid 1 \mid x \mid t + t \\ \phi & ::= & t = t \mid \neg\phi \mid \phi \vee \phi \mid \exists y.\phi \end{array}$$

où x et y appartiennent à un ensemble de variables.
L'ensemble $\mathbf{var}(\phi)$ des variables libres d'une formule de Presburger ϕ est défini par induction de la façon suivante :

- $\mathbf{var}(0) = \mathbf{var}(1) = \emptyset$,
- $\mathbf{var}(x) = \{x\}$,
- $\mathbf{var}(t + t') = \mathbf{var}(t) \cup \mathbf{var}(t')$,
- $\mathbf{var}(t = t') = \mathbf{var}(t) \cup \mathbf{var}(t')$,
- $\mathbf{var}(\neg\phi) = \mathbf{var}(\phi)$,
- $\mathbf{var}(\phi \vee \phi') = \mathbf{var}(\phi) \cup \mathbf{var}(\phi')$, et
- $\mathbf{var}(\exists y.\phi) = \mathbf{var}(\phi) \setminus \{y\}$.

Étant donné un ensemble X de variables, nous notons $\mathbf{Presb}(X)$, l'ensemble de formules de Presburger qui ont leurs variables libres dans X.

Soient une formule de Presburger ϕ et une fonction $f : \mathbf{var}(\phi) \to \mathbb{N}$. Nous définissons la fonction $\mathbf{App}_f$ qui à chaque terme de ϕ associe un entier de la façon suivante :

- $\mathbf{App}_f(0) = 0$,
- $\mathbf{App}_f(1) = 1$,
- $\mathbf{App}_f(x) = f(x)$, et,
- $\mathbf{App}_f(t + t') = App_f(t) + App_f(t')$.

Cette fonction, nous permet de définir la relation de satisfiabilité $f \models \phi$ par :

- $f \models t = t'$ si et seulement si $\mathbf{App}_f(t) = \mathbf{App}_f(t')$,
- $f \models \neg\phi$ si et seulement si $f \not\models \phi$,
- $f \models \phi \vee \phi'$ si et seulement si $f \models \phi$ ou $f \models \phi'$, et,
- $f \models \exists y.\phi$ si et seulement si il existe $f' : \mathbf{var}(\phi) \to \mathbb{N}$ étendant f et telle que $f' \models \phi$.

Deux formules de Presburger ϕ et ϕ' sont dites équivalentes (noté $\phi \equiv \phi'$) si pour toute fonction $f : \mathbf{var}(\phi) \cup \mathbf{var}(\phi') \to \mathbb{N}$, on a $f_{|\mathbf{var}(\phi)} \models \phi$ si et seulement si $f_{|\mathbf{var}(\phi')} \models \phi'$. Nous dirons qu'une formule de Presburger ϕ est valide si pour toute fonction $f : \mathbf{var}(\phi) \to \mathbb{N}$, on a $f_{|\mathbf{var}(\phi)} \models \phi$.
Nous interprétons ainsi les formules de Presburger sur l'ensemble $\mathbb{N}$ des entiers naturels. Nous pouvons étendre de plus la grammaire avec les opérateurs logiques classiques, comme le "`et`" (noté $\wedge$) qui se définit facilement par $\phi \wedge \phi' \equiv \neg(\neg\phi \vee \neg\phi')$ ou encore le quantificateur universel $\forall$ défini par $\forall y.\phi \equiv \neg\exists y.\neg\phi$. Nous notons, $true$, le symbole de vérité qui peut être défini par $true \equiv \exists y.y = y$ et $false$ sa négation. De la même façon, nous ajoutons à la grammaire, les opérateurs arithmétiques classiques $\neq$,$<$, $\leq$, $\geq$ et $>$, qui peuvent aussi être définis dans l'arithmétique de Presburger. Par exemple, $t < t' \equiv \exists y.(t' = t + y) \wedge \neg(y = 0)$.
La logique de Presburger est souvent utilisée en vérification. Ceci est du au fait que le problème de satisfiabilité, qui consiste à savoir si étant donnée une formule ϕ, il existe une fonction $f : \mathbf{var}(\phi) \to \mathbb{N}$

telle que $f \models \phi$, est décidable pour ce fragment logique [Pre29]. En revanche si l'on rajoute aux termes la multiplication, c'est à dire si l'on autorise des termes de la forme $t' * t$, le problème de satisfiabilité devient indécidable.

1.1.3 Les ensembles et relations définissables dans Presburger

Soit $n \in \mathbb{N}^*$. Étant donné un ensemble fini de variables $X = \{x_1, \ldots, x_n\}$ et un vecteur $\mathbf{a} \in \mathbb{N}^n$, nous définissons la fonction $f_{X,\mathbf{a}} : X \to \mathbb{N}$ telle que pour tout $i \in [1..n]$, $f_{X,\mathbf{a}}(x_i) = \mathbf{a}(i)$. Si ϕ est une formule dans $\mathbf{Presb}(X)$, le sous-ensemble $[\![\phi]\!]_X$ de $\mathbb{N}^n$ est défini par :

$$[\![\phi]\!]_X = \{\mathbf{a} \in \mathbb{N}^n \mid f_{X,\mathbf{a}} \models \phi\}$$

Notation 1.3 *Il nous arrivera parfois d'utliser la notation $[\![\phi]\!]$ au lieu de $[\![\phi]\!]_X$ lorsque il n'y aura pas de confusion possible sur l'ensemble X des variables considéré.*

Ceci nous permet de définir les ensembles définissables dans Presburger.

Definition 1.4 (Ensemble définissable dans Presburger) *Soit $n \in \mathbb{N}^*$. Un ensemble $E \subseteq \mathbb{N}^n$ est définissable dans Presburger si et seulement si il existe un ensemble fini de variables $X = \{x_1, \ldots, x_n\}$ et une formule de Presburger $\phi \in \mathbf{Presb}(X)$ tels que $E = [\![\phi]\!]_X$.*

Nous rappelons maintenant le lien qu'il existe entre les ensembles définissables dans Presburger et les ensembles semi-linéaires.

Théorème 1.5 *[GS66] Soit $E \subseteq \mathbb{N}^n$. E est définissable dans Presburger si et seulement si E est semi-linéaire.*

Exemple 1.6 *L'ensemble E_1 des entiers pairs est définissable dans Presburger. En effet, si l'on considère la formule $\phi_1 \equiv \exists y.x = y + y$, on a bien $E_1 = [\![\phi_1]\!]$.*

La proposition suivante fournit un exemple d'ensemble qui n'est pas définissable dans Presburger.

Lemme 1.7 *L'ensemble $\{2^n \mid n \in \mathbb{N}\}$ des puissances de 2 n'est pas définissable dans Presburger.*

Preuve : Nous raisonnons par l'absurde. Supposons que l'ensemble $E_2 = \{2^n \mid n \in \mathbb{N}\}$ est définissable dans Presburger. D'après le théorème 1.5, E_2 serait semi-linéaire. Comme E_2 est infini, on en déduit qu'il existe un ensemble linéaire infini $L \subseteq E_2$. Supposons qu'il existe $a_0, ..., a_k \in \mathbb{N}$ tel que $L = \{a_0 + \lambda_1.a_1 + \ldots + \lambda_k.a_k \mid \forall i \in [1..k], \lambda_i \in \mathbb{N}\}$. Supposons de plus que $a_1 = \mathbf{Max}(\{a_i \mid i \in [1..k]\})$. Comme L est infini, on a forcément $a_1 > 0$ et de plus, il existe $n \in \mathbb{N}$ tel que $2^n \in L$ et $a_1 < 2^n$. Par définition de L, $2^n + a_1 \in L$ et donc il existe m tel que $2^n + a_1 = 2^m$ avec $n + 1 \leq m$. Comme $a_1 > 0$, on en déduit $2^n \leq a_1$. Cette dernière inégalité constitue une contradiction avec le fait que $a_1 < 2^n$. □

Notation 1.8 *Étant donné un ensemble fini $X = \{x_1, ..., x_n\}$ de variables, nous notons $X' = \{x'_1, .., x'_n\}$ l'ensemble des variables "primées" obtenu à partir de X et $\mathbf{Presb}(X, X')$ l'ensemble des formules de Presburger ϕ telles que $\mathbf{var}(\phi) \subseteq X \uplus X'$. Pour deux fonctions $f : X \to \mathbb{N}$ et $f' : X' \to \mathbb{N}$, et une formule $\phi \in \mathbf{Presb}(X, X')$, nous notons $(f, f') \models \phi$ le fait que la fonction $g : X \uplus X' \to \mathbb{N}$, définie par $g_{|X} = f$ et $g_{|X'} = f'$, vérifie $g \models \phi$. Nous notons ainsi $[\![\phi]\!]_{X,X'} = \{(\boldsymbol{a}, \boldsymbol{a'}) \in \mathbb{N}^n \times \mathbb{N}^n \mid (f_{X,\boldsymbol{a}}, f_{X',\boldsymbol{a'}}) \models \phi\}$. De plus, parfois pour deux fonctions $f : X \to \mathbb{N}$ et $f' : X \to \mathbb{N}$, nous écrirons $(f, f') \models \phi$ si et seulement si la fonction $f'' : X' \to \mathbb{N}$, définie telle que pour tout $x'_i \in X'$, $f''(x'_i) = f'(x_i)$, vérifie $(f, f'') \models \phi$.*

Nous précisons la définition 1.4 dans le cas particulier des relations binaires.

Definition 1.9 (Relation binaire définissable dans Presburger) *Une relation binaire $R \subseteq \mathbb{N}^n \times \mathbb{N}^n$ est définissable dans Presburger si et seulement si il existe deux ensembles finis $X = \{x_1, \ldots, x_n\}$ et $X' = \{x'_1, \ldots, x'_n\}$ de variables et une formule de Presburger $\phi \in$ **Presb**(X, X') tels que $R = [\![\phi]\!]_{X,X'}$.*

1.1.4 Systèmes de transitions étiquetés

Pour représenter le comportement des différents modèles manipulés, nous utilisons des systèmes de transitions étiquetés par un alphabet.

Definition 1.10 (Système de transitions étiqueté) *Un système de transitions étiqueté est un triplet $TS = \langle C, \Sigma, \rightarrow \rangle$ tel que :*

- *C est un ensemble dont les éléments sont appelés des configurations,*
- *Σ est un alphabet fini, et,*
- *$\rightarrow \subseteq C \times \Sigma \times C$ est une relation appelée la relation de transition.*

Nous appellerons système de transitions, un système de transitions étiqueté. Soit $TS = \langle C, \Sigma, \rightarrow \rangle$ un système de transitions étiqueté. Pour $c, c' \in C$ et $a \in \Sigma$, nous adoptons la notation $c \xrightarrow{a} c'$ pour exprimer que $(c, a, c') \in \rightarrow$. De plus, nous notons $c \rightarrow c'$ si il existe $a \in \Sigma$ tel que $c \xrightarrow{a} c'$. Nous étendons aussi la relation $\rightarrow$ à des mots de Σ^*. Ainsi, nous avons pour tout $c \in C$, $c \xrightarrow{\epsilon} c$ et si $a \in \Sigma$ et $\sigma \in \Sigma^*$, nous définissons la relation $\xrightarrow{a.\sigma}$ par induction de la façon suivante :

$$c \xrightarrow{a.\sigma} c'' \text{si et seulement si il existe } c' \in S \text{ tel que } c \xrightarrow{a} c' \xrightarrow{\sigma} c''$$

Un système de transitions initialisé est une paire (TS, c_0) où $TS = \langle C, \Sigma, \rightarrow \rangle$ est un système de transitions étiqueté et $c_0 \in C$ est une configuration initiale.
Une exécution π d'un système de transitions TS est une suite finie de transitions $c_0 \xrightarrow{a_1} c_1, c_1 \xrightarrow{a_2} c_2, \ldots, c_{k-1} \xrightarrow{a_k} c_k$ notée plus simplement $c_0 \xrightarrow{a_1} c_1 \xrightarrow{a_2} \ldots \xrightarrow{a_k} c_k$. Nous considérerons parfois des exécutions infinies qui correspondront à des suites infinies de transitions.

1.2 Les systèmes à compteurs

1.2.1 Systèmes à compteurs

Definition 1.11 (Système à compteurs) *Un système S à n compteurs est un triplet $S = \langle Q, X, E \rangle$ tel que :*

- *Q est un ensemble fini d'états de contrôle,*
- *$X = \{x_1, \ldots, x_n\}$ est un ensemble fini de compteurs,*
- *$E \subseteq Q \times$ **Presb**$(X, X') \times Q$ est un ensemble fini de transitions, chacune étiquetée par une formule de Presburger dont les variables libres sont dans $X \uplus X'$ avec $X' = \{x'_1, \ldots, x'_n\}$.*

Une configuration d'un système à n compteurs $S = \langle Q, X, E \rangle$ est une paire $(q, \mathbf{v})$ où $q \in Q$ est un état de contrôle et $\mathbf{v} : X \rightarrow \mathbb{N}$ est une fonction qui à chaque compteur associe une valeur entière. Le comportement d'un système à compteurs $S = \langle Q, X, E \rangle$ est donné par un système de transitions étiqueté $TS(S) = \langle Q \times \mathbb{N}^X, E, \rightarrow \rangle$ où :

– $Q \times \mathbb{N}^X$ est l'ensemble des configurations,
– E est l'ensemble de transitions de S, et,
– $\rightarrow \subseteq (Q \times \mathbb{N}^X) \times E \times (Q \times \mathbb{N}^X)$ est une relation définie de la façon suivante, pour tout $(q, \mathbf{v})$, $(q', \mathbf{v}')$ dans $Q \times \mathbb{N}^X$, pour tout $e \in E$:

$$(q, \mathbf{v}) \xrightarrow{e} (q', \mathbf{v}') \text{ si et seulement si } e = (q, \phi, q') \text{ et } (\mathbf{v}, \mathbf{v}') \models \phi$$

Remarquons que les valeurs des compteurs ne peuvent pas être négatives.
Un système à compteurs est un objet syntaxique dont la sémantique est fournie par son système de transitions associé.

Nous appelons un système à n compteurs initialisé une paire (S, c_0) où $S = \langle Q, X, E \rangle$ est un système à n compteurs et $c_0 \in Q \times \mathbb{N}^n$ est une configuration initiale. À un système à compteurs initialisé (S, c_0), nous associons un système de transitions initialisé $TS(S, c_0) = (TS(S), c_0)$. Lorsque il n'y aura pas d'ambiguïté possible grâce aux notations, nous appellerons également système à compteurs un système à compteurs initialisé.

Exemple 1.12 *La figure 1.1 donne un exemple de système à un compteur. Lorsque le système est dans l'état q_1, il peut passer dans l'état q_2 en choisissant une valeur de compteur différente de celle courant et de l'état q_2, il peut retourner en q_1 sans changer la valeur du compteur. La figure 1.2 présente une modélisation du problème de Syracuse. Le système à compteurs n'a qu'un seul état et à chaque étape soit la valeur du compteur est paire et alors le système la divise par 2 soit elle est impaire, et il la multiplie par 3 et lui ajoute 1. Le problème de Syracuse consiste à montrer qu'à partir de n'importe quelle valeur initiale le système arrive toujours dans une configuration où la valeur du compteur vaut 0. Sur des exemples, cela semble être vraie, mais il n'existe pas encore de preuve de ce résultat.*

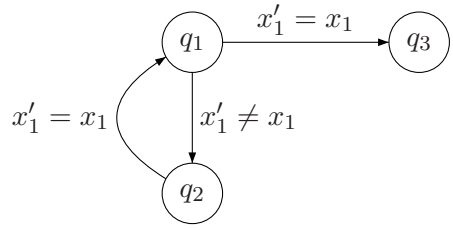


FIGURE 1.1 – Un système à compteurs

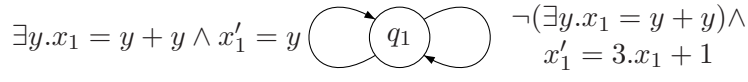


FIGURE 1.2 – Un système à compteurs encodant le problème de Syracuse

Les systèmes à compteurs sont étudiés depuis 40 ans et leur étude trouve en particulier des applications dans la vérification de systèmes informatiques.

1.2.2 Systèmes à compteurs fonctionnels et linéaires

Dans la définition des systèmes à compteurs, les transitions sont étiquetées par des formules de Presburger qui donnent des conditions sur les valeurs des compteurs avant et après le franchissement de la transition. Dans ces systèmes, étant données une configuration $(q, \mathbf{v})$ et une transition (q, ϕ, q'), le nombre de configurations $(q', \mathbf{v}')$ vérifiant $(q, \mathbf{v}) \xrightarrow{\phi} (q', \mathbf{v}')$ peut être infini. Nous définissons ici une sous-classe de systèmes à compteurs, pour laquelle, étant donné une configuration $(q, \mathbf{v})$ et une transition (q, ϕ, q'), il existe au plus une configuration $(q', \mathbf{v}')$ telle que $(q, \mathbf{v}) \xrightarrow{\phi} (q', \mathbf{v}')$. Nous limitons ainsi le non-déterminisme au choix de la transition à franchir. Pour obtenir une telle classe de systèmes, les formules de Presburger présentes dans le système à n compteurs caractériseront une fonction partielle de $\mathbb{N}^n$ dans $\mathbb{N}^n$.

Definition 1.13 (Système à compteurs fonctionnel) *Un système à n compteurs fonctionnel $S = \langle Q, X, E\rangle$ est un système à compteurs tel que pour tout $(q, \phi, q') \in E$, il existe une fonction partielle $f : \mathbb{N}^n \to \mathbb{N}^n$ vérifiant pour tout $\boldsymbol{v}, \boldsymbol{v}' \in \mathbb{N}^n$, $(\boldsymbol{v}, \boldsymbol{v}') \in [\![\phi]\!]$ si et seulement si $\boldsymbol{v} \in \mathbf{dom}(f)$ et $\boldsymbol{v}' = f(\boldsymbol{v})$.*

Definition 1.14 (Fonction Presburger-linéaire) *[FL02] Une fonction partielle $f : \mathbb{N}^n \to \mathbb{N}^n$ est dite Presburger-linéaire si et seulement si il existe un triplet $(\psi, A, \boldsymbol{b})$ tel que $\psi \in \mathbf{Presb}(X)$ pour un ensemble fini de variables $X = \{x_1, \ldots, x_n\}$, $A \in \mathcal{M}_n(\mathbb{Z})$ et $b \in \mathbb{Z}^n$ et :*

- $\mathbf{dom}(f) = [\![\psi]\!]_X$, *et,*
- *pour tout $\boldsymbol{v} \in \mathbf{dom}(f)$, $f(\boldsymbol{v}) = A.\boldsymbol{v} + b$.*

Remarquons que pour une fonction Presburger-linéaire $f : \mathbb{N}^n \to \mathbb{N}^n$, la relation binaire sur $\mathbb{N}^n$ définie par $\{(\mathbf{v}, \mathbf{v}') \in \mathbb{N}^n \times \mathbb{N}^n \mid \mathbf{v} \in \mathbf{dom}(f) \text{ et } \mathbf{v}' = f(\mathbf{v})\}$ est définissable dans Presburger. Ces fonctions Presburger-linéaires sont étudiées en détail dans [Ler03].

Definition 1.15 (Système à compteurs linéaire) *Un système à n compteurs linéaire $S = \langle Q, X, E\rangle$ est un système à compteurs tel que pour tout $(q, \phi, q') \in E$, il existe une fonction Presburger-linéaire f vérifiant pour tout $\boldsymbol{v}, \boldsymbol{v}' \in \mathbb{N}^n$, $(\boldsymbol{v}, \boldsymbol{v}') \in [\![\phi]\!]$ si et seulement si $\boldsymbol{v} \in \mathbf{dom}(f)$ et $\boldsymbol{v}' = f(\boldsymbol{v})$.*

Notation 1.16 *Nous noterons parfois $f = (\psi, A, \boldsymbol{b})$ pour une fonction Presburger-linéaire f. De plus, lorsque nous considérerons une transition (q, ϕ, q') d'un système à compteurs linéaire, nous utiliserons parfois directement la notation $(q, (\psi, A, \boldsymbol{b}), q')$ ou (q, f, q') avec $f = (\psi, A, \boldsymbol{b})$. Nous adopterons également cette notation dans le système de transitions associé à un système à compteurs linéaire.*

Pour une transition $(q, (\psi, A, \mathbf{b}), q')$ d'un système à compteurs linéaire, la formule ψ est appelée la garde de la transition et la paire $(A, \mathbf{b})$ l'action. Lorsque nous représenterons un système à compteurs linéaire graphiquement, comme par exemple sur la figure 1.3, sur chaque transition nous ferons systématiquement suivre la garde par un point d'interrogation "?". Quand la garde sera équivalente à $true$, nous ne la mentionnerons pas et nous n'indiquerons que les actions changeant les valeurs de compteurs. Ainsi pour la transition entre q_1 et q_3 sur la figure 1.3, nous avons implicitement la garde équivalente à $true$ et l'action équivalente à $x'_1 = x_1$.

Exemple 1.17 *Le système à un compteur représenté sur la figure 1.1 n'est pas linéaire. En revanche, nous pouvons "simuler" son comportement par le système à compteurs de la figure 1.3. Pour simuler la transition de q_1 vers q_3 étiquetée par la formule $x'_1 \neq x_1$, nous ajoutons deux boucles partant de l'état de contrôle q_1, l'une qui décrémente le compteur à chaque coup et l'autre qui l'incrémente.*

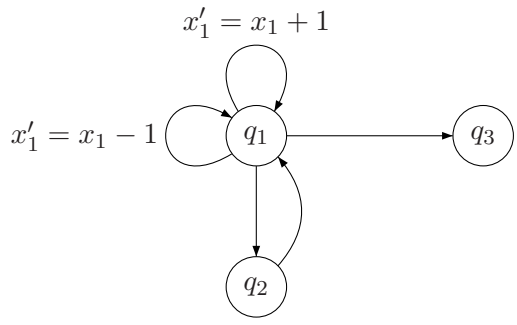


FIGURE 1.3 – Un système à compteurs linéaire

1.2.3 Machines de Minsky et machines à compteurs

Dans [Min67], Minsky introduit un modèle pour représenter des "machines" de calcul réalisant des opérations sur des variables entières. Une machine de Misnky manipule deux variables entières x_1 et x_2 et est composée d'une séquence finie d'instructions, chacune d'elle pouvant avoir une des deux formes suivantes :

1. $l : x_i := x_i + 1$; goto l'
2. l : if $x_i = 0$ then goto l' else $x_i := x_i - 1$; goto l''

où $i \in \{1, 2\}$ et l, l' et l'' sont des étiquettes précédant chacune des instructions. Nous proposons maintenant un codage sous forme de systèmes à compteurs de ces machines.

Étant donné un ensemble fini $X = \{x_1, \ldots, x_n\}$ de n variables, nous définissons les trois types de formules dans $\mathbf{Presb}(X, X')$ suivantes, pour tout $i \in [1..n]$:

– $\mathbf{inc}(x_i) \equiv x_i' = x_i + 1 \wedge \bigwedge_{j \in [1..n] \setminus \{i\}} x_j' = x_j$,
– $\mathbf{dec}(x_i) \equiv x_i' = x_i - 1 \wedge \bigwedge_{j \in [1..n] \setminus \{i\}} x_j' = x_j$, et
– $\mathbf{ifzero}(x_i) \equiv x_i = 0 \wedge \bigwedge_{j \in [1..n]} x_j' = x_j$.

Intuitivement, $\mathbf{inc}(x_i)$ revient à incrémenter la valeur du compteur x_i, $\mathbf{dec}(x_i)$ à la décrémenter et $\mathbf{ifzero}(x_i)$ à tester si elle vaut zéro.

Definition 1.18 (Machine de Minsky) *Une machine de Minsky à n compteurs est un système à compteurs $S = \langle Q, X, E \rangle$, avec $X = \{x_1, \ldots, x_n\}$, tel que pour tout $(q, \phi, q') \in E$, il existe un $i \in [1..n]$ tel que $\phi \equiv \mathbf{inc}(x_i)$ ou $\phi \equiv \mathbf{dec}(x_i)$ ou $\phi \equiv \mathbf{ifzero}(x_i)$.*

La définition de machines de Minsky que nous proposons ici est un peu différente de celle proposée par Minsky car nous introduisons du non-déterminisme. En effet, lorsque la machine est dans un état de contrôle, elle a le choix entre différentes actions possibles et non pas une seule comme c'est le cas dans le modèle proposé originellement. De plus, nous autorisons la machine à réaliser un test à zéro, sans forcément décrémenter le compteur correspondant si le test n'est pas satisfait. Nous donnons maintenant la définition de machines de Minsky déterministes, cette restriction est plus proche du modèle proposé par Minsky dans [Min67].

Definition 1.19 (Machine de Minsky déterministe) *Une machine de Minsky à n compteurs $S = \langle Q, X, E\rangle$, avec $X = \{x_1, \ldots, x_n\}$, est déterministe si et seulement si pour tout $q \in Q$, l'ensemble $S_q = \{e \in E \mid \exists(\phi, q').e = (q, \phi, q')\}$ vérifie les propriétés suivantes :*

1. *$|S_q| \leq 2$, et,*
2. *si $|S_q| = 1$, alors il existe $i \in [1..n]$ et $q' \in Q$ tel que $S_q = \{(q, \textbf{inc}(x_i), q')\}$, et,*
3. *si $|S_q| = 2$, alors il existe $i \in [1..n]$ et $q', q'' \in Q$ tels que nous avons $S_q = \{(q, \textbf{dec}(x_i), q'), (q, \textbf{ifzero}(x_i), q'')\}$.*

Remarquons que les machines de Minsky représentent bien une sous-classe de systèmes à compteurs linéaires.
Nous proposons maintenant une extension de ces machines de Minsky en autorisant des gardes un peu plus complexes que le test à zéro et aussi en autorisant les compteurs à être incrémentés et décrémentés non plus d'une unité mais d'un nombre fini d'unités en une transition. Nous appelons ce nouveau modèle les machines à compteurs car il se situe entre les machines de Minsky et les systèmes à compteurs. Dans un premier temps nous définissons les translations gardées dont nous nous servirons pour étiqueter les transitions de ces machines à compteurs.

Definition 1.20 (Translation gardée) *Une fonction partielle $t : \mathbb{N}^n \to \mathbb{N}^n$ est une translation gardée si et seulement si il existe $\# \in \{=, \leq\}^n$, $\mu \in \mathbb{N}^n$ et $\delta \in \mathbb{Z}^n$ avec $0 \leq \mu + \delta$ tels que :*

– $\textbf{dom}(t) = \{\boldsymbol{v} \in \mathbb{N}^n | \mu \# \boldsymbol{v}\}$, *et,*
– *pour tout $\boldsymbol{v} \in \textbf{dom}(t)$, $t(\boldsymbol{v}) = \boldsymbol{v} + \delta$.*

Soit $n \in \mathbb{N}^*$. Nous appelons T_n l'ensemble des translations gardées à valeurs dans $\mathbb{N}^n$. Remarquons qu'une translation gardée est une fonction Presburger-linéaire. Nous définissons maintenant les systèmes à compteurs dont les transitions sont étiquetées par des translations gardées.

Definition 1.21 (Machine à compteurs) *Une machine à n compteurs $S = \langle Q, X, E\rangle$ est un système à compteurs linéaire tel que pour tout $(q, \phi, q') \in E$, il existe une translation gardée t vérifiant pour tout $\boldsymbol{v}, \boldsymbol{v}' \in \mathbb{N}^n$, $(\boldsymbol{v}, \boldsymbol{v}') \in [\![\phi]\!]$ si et seulement si $\boldsymbol{v} \in \textbf{dom}(t)$ et $\boldsymbol{v}' = t(\boldsymbol{v})$.*

Notons juste que toute machine de Minsky est une machine à compteurs.

Notation 1.22 *Nous noterons parfois $t = (\#, \mu, \delta)$ une translation gardée t. De plus, de la même façon que pour les systèmes à compteurs linéaires, lorsque nous considérerons une transition (q, ϕ, q') d'un système à compteurs linéaire, nous utiliserons parfois directement la notation $(q, (\#, \mu, \delta), q')$.*

Il est facile d'encoder le comportement d'une machine à n compteurs dans une machine de Minsky. Par exemple, un incrément de d unités dans la machine à compteurs peut-être réalisé par d transitions incrémentant le même compteur dans une machine de Minsky. Une construction similaire peut-être utilisée pour un décrément de d unités. Quant aux tests, pour tester si la valeur d'un compteur est supérieure ou égale à une constante d dans la machine à compteurs, il suffit de décrémenter d fois le compteur dans la machine de Minsky et de l'incrémenter ensuite de nouveau d fois, en effet la machine de Minsky ne pourra passer cette suite de transitions que dans le cas où la valeur du compteur concerné est plus grande que d (un compteur ne pouvant pas prendre une valeur négative !). Pour tester si la valeur d'un compteur est égale à une constante d, on utilise la même astuce en insérant un test

à zéro entre la phase de décrémentation et celle d'incrémentation. Néanmoins ce codage modifie le comportement des compteurs et en particulier le nombre de fois où un compteur alterne entre une phase où il est décrémenté et une phase où il est incrémenté. Or, comme nous le verrons par la suite, ce nombre d'alternances sera un facteur déterminant lorsque nous considérerons les machines à compteurs *reversal*-bornées dans le chapitre 2. C'est pour cela que nous étudions ces deux modèles.

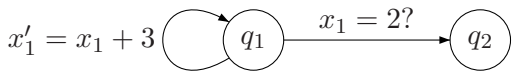


FIGURE 1.4 – Une machine M_1 à un compteur

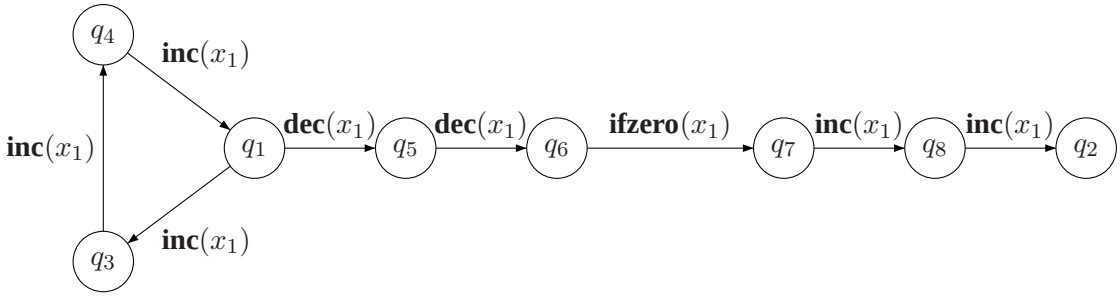


FIGURE 1.5 – Une machine de Minsky pour simuler M_1

Exemple 1.23 *La figure 1.4 nous donne un exemple de machine à compteurs et la figure 1.5 un exemple de machine de Minsky qui "simule" la première machine. Nous avons construit cette machine de Minsky en appliquant le processus décrit ci-dessus. Sur ces deux exemples, on peut constater, que lorsque la machine à compteurs réalise une opération pour changer la valeur d'un compteur, la machine de Minsky peut faire la même opération mais elle a besoin de plusieurs pas, c'est à dire de franchir plusieurs transitions.*

1.2.4 Réseaux de Petri et SAVE

Parmi les modèles utilisés dans la vérification, les réseaux de Petri [Pet62] occupent une place importante que ce soit dans le milieu universitaire ou bien chez les industriels. Nous montrons ici le lien qui existe entre les réseaux de Petri et les machines à compteurs.
Dans un premier temps nous rappelons la définition des Réseaux de Petri.

Definition 1.24 (Réseau de Petri) *Un réseau de Petri est un* 5*-uplet* $\langle P, T, W^-, W^+, \boldsymbol{M}_0 \rangle$ *tel que :*

- P *est un ensemble fini de places,*
- T *est un ensemble fini de transitions tel que* $P \cap T = \emptyset$*,*
- $W^- : T \to \mathbb{N}^{|P|}$ *est l'application d'incidence arrière,*
- $W^+ : T \to \mathbb{N}^{|P|}$ *est l'application d'incidence avant,*
- $M_0 \in \mathbb{N}^{|P|}$ *est un marquage (ou une configuration), appelée le marquage initiale.*

Intuitivement, un marquage indique le nombre de jetons présents dans chaque place. Étant donnés un marquage M et une transition $t \in T$, pour franchir la transition t, il faut d'abord que chaque place ait un nombre minimum de jetons, donné par le vecteur $W^-(t)$, le franchissement de la transition provoque ensuite la consommation du nombre de jetons fourni par $W^-(t)$, et la production de nouveaux jetons, le nombre de jetons ajoutés dans chaque place étant celui indiqué par le vecteur $W^+(t)$.

Plus formellement le comportement d'un réseau de Petri $\langle P, T, W^-, W^+, M_0\rangle$ peut être décrit par un système de transitions initialisé $(\langle \mathbb{N}^{|P|}, T, \rightarrow\rangle, M_0)$ où :

– $\mathbb{N}^{|P|}$ est l'ensemble des marquages,
– T est l'alphabet d'étiquetage,
– $\rightarrow \subseteq \mathbb{N}^{|P|} \times T \times \mathbb{N}^{|P|}$ est la relation définie de la façon suivante, pour tout $M, M' \in \mathbb{N}^{|P|}$, pour tout $t \in T$:

$$M \xrightarrow{t} M' \text{ si et seulement si } W^-(t) \leq M \text{ et } M' = M - W^-(t) + W^+(t)$$

– M_0 est le marquage initial.

Les réseaux de Petri ont une propriété intéressante qui est la monotonie et qui peut être exprimée par la proposition suivante :

Proposition 1.25 *Soit $\langle P, T, W^-, W^+, M_0\rangle$ un réseau de Petri et $(\langle \mathbb{N}^{|P|}, T, \rightarrow\rangle, M_0)$ le système de transitions initialisé qui lui est associé. Pour tout $M_1, M_2 \in \mathbb{N}^{|P|}$ et $t \in T$, si $M_1 \xrightarrow{t} M_2$ alors pour tout $M_1' \in \mathbb{N}^{|P|}$ tel que $M_1 \leq M_1'$, il existe un marquage $M_2' \in \mathbb{N}^{|P|}$ vérifiant $M_1' \xrightarrow{t} M_2'$ et $M_2' - M_2 = M_1' - M_1$.*

Ceci est dû au fait qu'avant de franchir une transition, on teste pour chaque place si le nombre de jetons est supérieur ou égal à une constante, et qu'il n'est pas possible de tester si il est égal à une constante.

Exemple 1.26 *La figure 1.6 donne un exemple de réseau de Petri. Traditionnellement les places sont représentées par des cercles et les transitions par des rectangles. Les jetons placés dans les places indiquent le marquage. Ce réseau de Petri peut représenter deux processus partageant une ressource avec un système d'exclusion mutuelle. Les places p_1 et p_2 correspondent aux états du premier processus, en p_1 le processus attend la ressource et en p_2 le processus possède la ressource, symétriquement p_3 et p_4 représentent les états du deuxième processus. Il y a un jeton dans la place p_5 lorsque la ressource est libre. À partir du marquage initial représenté, on voit qu'il n'est pas possible d'avoir au même instant un jeton dans la place p_2 et un dans la place p_4, ce qui caractérise le fait que les deux processus ne peuvent posséder la ressource en même temps.*

Dans [KM69], les auteurs introduisent les systèmes d'addition de vecteurs qui sont équivalents aux réseaux de Petri. Ces modèles peuvent être interprétés comme des machines à compteurs avec un unique état de contrôle et ne pouvant pas utiliser l'égalité dans leurs gardes, mais seulement le test supérieur ou égal. Les Systèmes d'Addition de Vecteurs avec États (SAVE) ont ensuite été introduits dans [HP79]. Il s'agit en fait de système d'additions de vecteur auxquels des états de contrôle ont été ajoutés. Nous donnons ici la définition que nous utiliserons pour les SAVE, ce qui permet de faire le lien avec les machines à compteurs.

Definition 1.27 (Système d'addition de vecteurs avec états) *Une machine à n compteurs $S = \langle Q, X, E\rangle$ est un système d'addition de vecteurs avec états (SAVE) si et seulement si pour tout $(q, t, q') \in E$, t est une translation gardée $(\#, \mu, \delta)$ telle que $\# = (\leq, \ldots, \leq)$.*

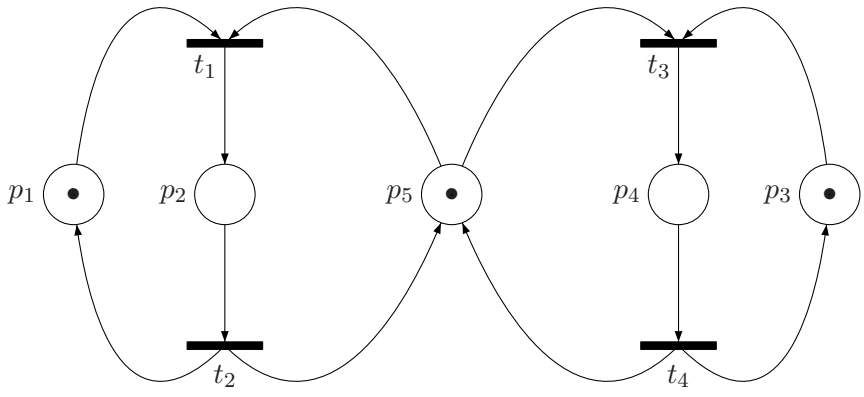


FIGURE 1.6 – Un réseau de Petri R_1

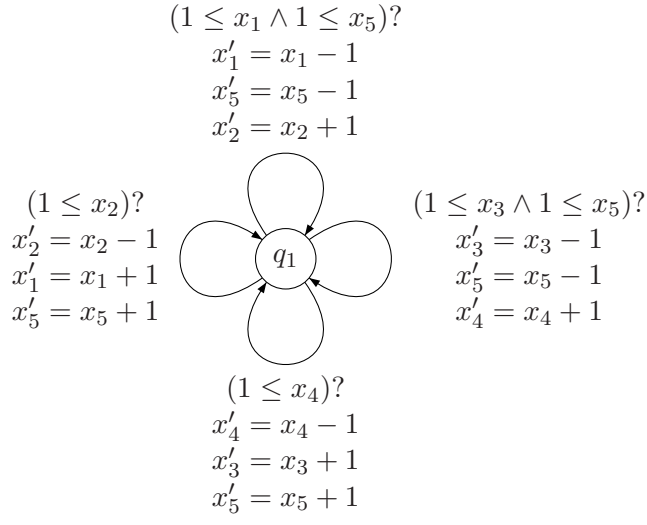


FIGURE 1.7 – Un SAVE S_1 qui simule R_1

Originellement, les systèmes d'addition de vecteurs et les SAVE étaient définis sans garde sur la relation de transition, mais il est toujours possible de simuler une garde sans test d'égalité en utilisant le fait que les valeurs des compteurs ne descendent jamais en dessous de 0, comme c'est le cas dans l'encodage des machines à compteurs par des machines de Minsky que nous avons expliqué précédemment. Remarquons qu'un réseau de Petri avec n places peut facilement être encodé dans un SAVE de dimension n avec un état de contrôle, chaque transition du SAVE simulant le franchissement d'une transition du réseau de Petri.

Exemple 1.28 *La figure 1.7 fournit un exemple d'encodage en SAVE du réseau de Petri de la figure 1.6. Pour simuler complètement ce réseau de Petri, la configuration initiale du SAVE présenté doit être* $(q_1, (1, 0, 1, 0, 1))$. *À chaque étape la valeur du compteur* x_i *(pour* $i \in [1..5]$*) représente le nombre de jetons présents dans la place* p_i.

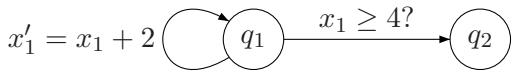


FIGURE 1.8 – Un SAVE S_2

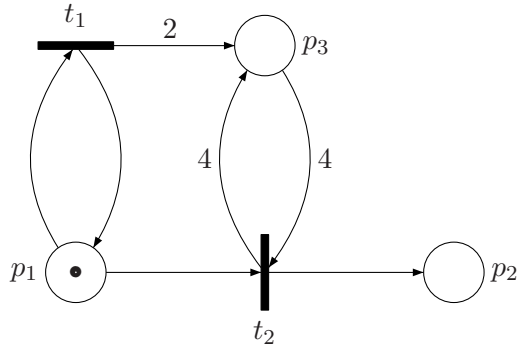


FIGURE 1.9 – Un réseau de Petri simulant S_2

Les SAVE sont équivalents aux systèmes d'addition de vecteurs et par conséquent aux réseaux de Petri. Il est en effet possible de simuler le comportement d'un SAVE de dimension n ayant m états de contrôle avec un réseau de Petri avec $n + m$ places. Les n premières places servant à encoder les n compteurs et les m dernières places sont utilisées pour encoder l'état de contrôle en mettant un jeton dans la place correspondant à l'état de contrôle courant. Il existe des codages plus astucieux que celui que nous proposons ici, ainsi dans [HP79], il est montré que tout SAVE de dimension n peut être encodé dans un système d'addition de vecteurs de dimension $n + 3$, équivalent à un réseau de Petri à $n + 3$ places. Différentes propriétés mathématiques des réseaux de Petri et des SAVE ainsi que les méthodes développées pour analyser ces modèles sont décrites précisément dans [Reu89]. Dans

la suite, nous nous intéresserons principalement aux SAVE, et les propriétés qui seront valables pour eux le seront aussi pour les réseaux de Petri.

Exemple 1.29 *Les figures 1.8 et 1.9 donnent un exemple de SAVE et un encodage possible dans un réseau de Petri.*

1.2.5 Schéma récapitulatif

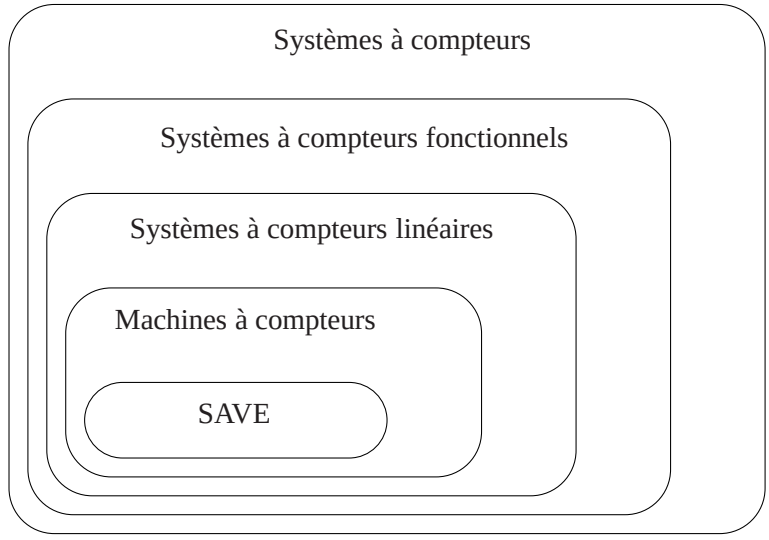


FIGURE 1.10 – Schéma récapitulatif des différentes classes de systèmes à compteurs introduites

Le schéma de la figure 1.10 récapitule les différentes classes de systèmes à compteurs que nous avons introduites précédemment et montre comment elles sont incluses les unes dans les autres (au niveau syntaxique).
Pour finir, dans [DFGD06], la proposition suivante a été énoncée :

Proposition 1.30 *Savoir si un système à compteurs est un système à compteurs fonctionnel ou un système à compteurs linéaire ou une machine à compteurs est décidable.*

Ainsi, étant donné un système à compteurs quelconque on peut savoir à quelle classe il appartient.

1.3 Problématique de la vérification

1.3.1 Différents problèmes d'accessibilité

Comme nous l'avons signalé précédemment, les systèmes à compteurs vont nous servir de modèles pour la vérification. Nous présentons maintenant les premiers problèmes de vérification auxquels nous allons nous intéresser. De façon intuitive, chacun de ces problèmes peut être vu comme un problème de "sûreté". À chaque fois, une configuration ou un ensemble de configurations est donné, et nous souhaitons vérifier si une exécution du système arrive dans cet ensemble. On parle de "sûreté" car l'ensemble de configurations donné peut typiquement représenter un ensemble de mauvais états

du système. Afin de vérifier des systèmes à compteurs, nous nous intéressons à leur comportement donné par le système de transitions qui leur est associé et en particulier aux configurations qui sont accessibles.
Soient $S = \langle Q, X, E\rangle$ un système à compteurs, $TS = \langle Q \times \mathbb{N}^X, E, \rightarrow\rangle$ le système de transitions qui lui est associé et $c_0 \in Q \times \mathbb{N}^X$ une configuration initiale. L'ensemble d'accessibilité du système à compteurs initialisé (S, c_0) est $\mathbf{Reach}(S, c_0) = \{c \in Q \times \mathbb{N}^X \mid c_0 \rightarrow^* c\}$. De plus, nous appelons relation d'accessibilité de S, la relation $\rightarrow^*$. Nous aurons parfois besoin de calculer les successeurs "en un coup" d'un ensemble de configurations. Ainsi si $C \subseteq Q \times \mathbb{N}^X$ est un ensemble de configurations, nous notons $\mathbf{Post}(S, C)$ l'ensemble de configurations accessibles "en un coup" à partir de C, c'est à dire l'ensemble $\{c' \in Q \times \mathbb{N}^X \mid \exists c \in C$ tel que $c \rightarrow c'\}$.

Dans un système à compteurs, une des premières questions que l'on peut se poser est de savoir si un état de contrôle est accessible, c'est à dire savoir si il existe une exécution dans le système de transitions qui, à partir d'une configuration initiale, amène à un état de contrôle donné. En effet, il se pourrait que cet état de contrôle représente une erreur dans le système modélisé par le système à compteurs, et donc il est important de savoir le système peut se trouver dans un tel état. Nous définissons le *problème d'accessibilité d'un état de contrôle* de la façon suivante :

Entrées : Un système à compteurs $S = \langle Q, X, E\rangle$, une configuration initiale $c_0 \in Q \times \mathbb{N}^X$ et un état de contrôle $q \in Q$,
Question : Existe-t-il une valuation $\mathbf{v} \in \mathbb{N}^X$ telle que $(q, \mathbf{v}) \in \mathbf{Reach}(S, c_0)$?

Si la réponse est oui, nous dirons que q est accessible dans (S, c_0). Ce problème nous donne des informations sur les états de contrôle accessibles, mais pas sur les valeurs que prennent les compteurs lorsque le système arrive dans l'état de contrôle donné. Nous définissons maintenant le *problème d'accessibilité d'une configuration* :

Entrées : Un système à compteurs $S = \langle Q, X, E\rangle$ et deux configurations $c_0, c \in Q \times \mathbb{N}^X$.
Question : Est-ce-que $c \in \mathbf{Reach}(S, c_0)$?

Si la réponse est oui, nous dirons que c est accessible dans (S, c_0). Nous pouvons étendre les problèmes précédents en considérant non plus une configuration (ou un ensemble fini de configurations), mais un ensemble possiblement infini de configurations. Nous avons vu dans les préliminaires que les formules de Presburger étaient un formalisme logique décidable permettant de manipuler des ensembles de vecteurs d'entiers. Nous proposons ici d'utiliser la logique de Presburger pour encoder des configurations. Étant donné un système à compteurs $S = \langle Q, X, E\rangle$, nous appelons configuration symbolique une paire (q, ϕ) telle que $q \in Q$ et $\phi \in \mathbf{Presb}(X)$. Ceci nous permet de définir le *problème d'accessibilité symbolique* :

Entrées : Un système à compteurs $S = \langle Q, X, E\rangle$, une configuration initiale $c_0 \in Q \times \mathbb{N}^X$ et une configuration symbolique $(q, \phi) \in Q \times \mathbf{Presb}(X)$.
Question : Existe-t-il $\mathbf{v} \times \mathbb{N}^X$ tel que $\mathbf{v} \models \phi$ et $(q, \mathbf{v}) \in \mathbf{Reach}(S, c_0)$?

Finalement, nous sommes partis à chaque fois d'une configuration initiale, mais il est aussi intéressant de vérifier des propriétés pour un ensemble de configurations initiales. Étant donnés un système à compteurs $S = \langle Q, X, E\rangle$ et une configuration symbolique initiale $(q_0, \phi_0) \in Q \times \mathbf{Presb}(X)$, nous étendons la définition d'ensemble d'accessibilité et définissons l'ensemble d'accessibilité symbolique

$\mathbf{Reach}(S,(q_0,\phi_0)) = \{(q,\mathbf{v}) \in Q \times \mathbb{N}^X \mid \exists \mathbf{v}_0 \in \mathbb{N}^X \text{ tel que } \mathbf{v}_0 \models \phi_0 \text{ et } (q,\mathbf{v}) \in \mathbf{Reach}(q,\mathbf{v}_0)\}$. Nous définissons ainsi le *problème d'accessibilité symbolique généralisé* :

Entrées : Un système à compteurs $S = \langle Q, X, E\rangle$ et deux configurations symboliques (q_0,ϕ_0) et (q,ϕ) dans $Q \times \mathbf{Presb}(X)$.
Question : Existe-t-il $\mathbf{v} \in \mathbb{N}^X$ tel que $\mathbf{v} \models \phi$ et $(q,\mathbf{v}) \in \mathbf{Reach}(S,(q_0,\phi_0))$?

Nous allons par la suite nous intéresser à la décidabilité des ces différents problèmes. De plus, pour le problème d'accessibilité d'un état de contrôle par exemple, si S est un système à compteurs, nous dirons que le problème d'accessibilité d'un état de contrôle est décidable pour S si il est décidable quelle que soit la configuration initiale donnée et l'état de contrôle donné. De la même façon, nous dirons que ce problème est décidable pour (S, c_0) où c_0 est une configuration initiale si ce problème est décidable pour S avec la configuration initiale c_0 quelque soit l'état de contrôle donné. Nous adaptons ce discours selon le problème considéré.

Exemple 1.31 *Nous reprenons le SAVE de la figure 1.7 qui encode le réseau de Petri représentant la propriété d'exclusion mutuelle de la figure 1.6. Vérifier la propriété d'exclusion mutuelle, à savoir que deux processus ne possèdent pas la ressource au même instant, peut être fait en résolvant un problème d'accessibilité symbolique avec comme configuration symbolique la paire* (q_1,ϕ) *où* $\phi \equiv x_2 \geq 1 \wedge x_4 \geq 1$. *En effet, si la réponse à ce problème d'accessibilité symbolique est positive, alors il existe une exécution qui place en même temps un jeton dans la place* p_2 *et un dans la place* p_4*, ce qui signifie qu'à ce moment les deux processus possèdent la ressource.*

Remarquons de plus que les problème d'accessibilité d'un état de contrôle et d'une configuration sont des cas particuliers du problème d'accessibilité symbolique, qui est lui-même un cas particulier du problème d'accessibilité symbolique généralisé. Ainsi lorsque le problème d'accessibilité symbolique généralisé sera décidable alors tous les autres problèmes le seront aussi et de la même façon lorsque le problème d'accessibilité symbolique sera décidable, les problèmes d'accessibilité d'un état de contrôle et d'une configuration le seront aussi.

Exemple 1.32 *Par exemple, si nous voulons savoir, pour un système à* n *compteurs* $S = \langle Q, X, E\rangle$*, si un état de contrôle* q *est accessible à partir d'une configuration initiale* $(q_0,\mathbf{v}_0)$*, cela peut-être fait en résolvant le problème d'accessibilité généralisé avec comme configuration symbolique initiale la paire* (q_0,ϕ_0) *où* $\phi_0 = \bigwedge_{i\in[1..n]} x_i = \mathbf{v}_0(x_i)$ *et comme configuration symbolique* $(q, true)$ *pour encoder l'état de contrôle* q*.*

1.3.2 Calcul de l'ensemble d'accessibilité et de la relation d'accessibilité

Une méthode pour résoudre les problèmes d'accessibilité précédents consiste à calculer l'ensemble d'accessibilité. Lorsque l'on sait que le nombre de configurations accessibles à partir d'une configuration initiale est fini, cela ne pose pas de problème. Un premier problème réside dans le fait que dans la plupart des cas il n'est pas possible de déterminer à l'avance si le nombre de configurations accessibles sera fini ou non. Un autre problème apparaît de plus lorsque cet ensemble d'accessibilité est infini. En effet, comment peut-on alors le représenter et le manipuler de façon à être capable de répondre aux problèmes d'accessibilité ? Nous avons déjà donné l'embryon d'une solution à ce problème dans la sous-section précédente en montrant qu'il était possible de représenter un ensemble infini de configurations en utilisant des formules de Presburger.

Cette technique qui consiste à représenter un ensemble infini de configurations d'un système par une formule a d'abord été introduite dans [KMM+97] où les auteurs proposent de représenter des ensembles de configurations par des formules d'une logique dérivée de la logique monadique du second ordre. Dans [BJNT00], les auteurs ont élaboré une approche similaire en montrant que pour la plupart des systèmes à états infinis il était possible d'encoder un ensemble de configurations dans les mots d'un langage régulier, c'est ce qu'on appelle le "regular model-checking". D'une façon générale, cette approche qui raisonne sur des ensembles infinis de configurations en manipulant une représentation symbolique est connue sous le nom de model-checking symbolique. Nous donnons ici une adaptation de cette méthode pour le cadre particulier des systèmes à compteurs.

De façon à manipuler uniquement des formules de Presburger, nous supposerons que l'ensemble Q des états de contrôle d'un système à n compteurs $S = \langle Q, X, E\rangle$ est un sous-ensemble de $\mathbb{N}$ égal à $[1..|Q|]$. Ceci nous permet d'encoder les configurations de S sur des vecteurs $\mathbf{v}$ à $n+1$ composantes dans lesquels $\mathbf{v}(0)$ est toujours utilisé pour encoder l'état de contrôle. Le système de transitions associé à S devient alors $TS(S) = \langle \mathbb{N}^{n+1}, E, \rightarrow\rangle$ et pour une configuration initiale c_0, nous avons $\mathbf{Reach}(S, c_0) \subseteq \mathbb{N}^{n+1}$. Pour un ensemble de compteurs $X = \{x_1, \ldots, x_n\}$, nous notons X_0 l'ensemble de variables $\{x_0, x_1, \ldots, x_n\}$ obtenu à partir de X en ajoutant la variable x_0 qui encodera les états de contrôle du système à compteurs.

Soient $S = \langle Q, X, E\rangle$ un système à n compteurs, $TS(S) = \langle \mathbb{N}^{n+1}, E, \rightarrow\rangle$ le système de transitions qui lui est associé et c_0 une configuration initiale. Supposons maintenant que l'ensemble $\mathbf{Reach}(S, c_0)$ d'accessibilité S soit définissable dans Presburger, et qu'il existe un algorithme permettant d'obtenir une formule de Presburger $\phi \in \mathbf{Presb}(X_0)$ telle que $\mathbf{Reach}(S, c_0) = [\![\phi]\!]$. Nous dirons alors que l'ensemble $\mathbf{Reach}(S, c_0)$ est effectivement définissable dans Presburger. L'adverbe effectivement indiquant que l'on peut calculer, de façon effective (ie avec un algorithme), la formule de Presburger correspondant à l'ensemble. Il est important de pouvoir obtenir la formule correspondant à l'ensemble de Presburger, car c'est cette formule que nous manipulons pour résoudre les questions d'accessibilité. Ainsi si l'on sait qu'un ensemble est définissable dans Presburger sans être capable de calculer la formule correspondante, cela peut poser problème. Nous avons alors la proposition suivante concernant les systèmes à compteurs initialisés ayant un ensemble d'accessibilité effectivement définissable dans Presburger :

Proposition 1.33 *Le problème d'accessibilité symbolique est décidable pour les systèmes à compteurs (S, c_0) ayant un ensemble d'accessibilité* $\mathbf{Reach}(S, c_0)$ *effectivement définissable dans Presburger.*

Preuve : Soient $S = \langle Q, X, E\rangle$ un système à compteurs, c_0 une configuration initiale de $TS(S)$ et $(q, \phi) \in Q \times \mathbf{Presb}(X)$ une configuration symbolique de $TS(S)$. Comme nous l'avons vu précédemment, nous pouvons encoder (q, ϕ) dans une formule ϕ' de $\mathbf{Presb}(X_0)$. Supposons que $\mathbf{Reach}(S, c_0)$ est effectivement définissable dans Presburger et soit ψ la formule de $\mathbf{Presb}(X_0)$ telle que $[\![\psi]\!] = \mathbf{Reach}(S, c_0)$. Le problème de l'accessibilité symbolique revient à décider si il existe $\mathbf{v} \in [\![\phi']\!] \cap [\![\psi]\!]$. Or $[\![\phi']\!] \cap [\![\psi]\!] = [\![\phi' \wedge \psi]\!]$, et comme ϕ' et ψ sont des formules de Presburger, $\phi' \wedge \psi$ aussi. Le problème d'accessibilité symbolique se réduit donc à un problème de satisfiabilité pour une formule de Presburger, qui est un problème décidable. □

Remarquons que si l'ensemble d'accessibilité est définissable dans Presburger mais qu'il n'y a pas d'algorithme permettant de calculer la formule correspondante, alors la proposition précédente n'est

plus vraie.
Pour l'instant nous n'avons évoqué que les cas où la configuration initiale était constituée d'un état de contrôle et d'une valuation de compteurs. Regardons maintenant ce qui se passe lorsqu'un ensemble de configurations initiales est fourni sous forme symbolique par une formule de Presburger, comme c'est le cas pour le problème d'accessibilité symbolique généralisé. La proposition 1.33 s'étend alors au problème d'accessibilité symbolique généralisé :

Proposition 1.34 *Le problème d'accessibilité symbolique généralisé est décidable pour les systèmes à compteurs S munis d'une configuration symbolique initiale (q_0, ϕ_0) et tels que* **Reach**$(S, (q_0, \phi_0))$ *est effectivement définissable dans Presburger.*

La preuve de cette proposition est similaire à celle de la proposition 1.33.

Soient $S = \langle Q, E, X\rangle$ un système à n compteurs et $TS(S) = \langle \mathbb{N}^{n+1}, E, \rightarrow\rangle$ le système de transitions associé. Lorsque nous encodons les configurations dans $\mathbb{N}^{n+1}$, la relation d'accessibilité $\rightarrow^*$ est une relation binaire sur $\mathbb{N}^{n+1}$. Dans les propositions précédentes, nous donnons à chaque fois l'ensemble des configurations initiales, et nous souhaitons que l'ensemble d'accessibilité pour ces configurations initiales soit définissable dans Presburger. Mais il se peut que pour une configuration initiale donnée, l'ensemble d'accessibilité soit définissable dans Presburger et pour une autre non. La proposition suivante fournit une condition suffisante pour éviter ce problème.

Proposition 1.35 *Soient $S = \langle Q, E, X\rangle$ un système à n compteurs et $TS(S) = \langle \mathbb{N}^{n+1}, E, \rightarrow\rangle$ le système de transitions qui lui est associé. Si la relation $\rightarrow^*$ est effectivement définissable dans Presburger, alors pour toute configuration symbolique initiale $(q_0, \phi_0) \in Q \times$* **Presb**$(X)$*, l'ensemble d'accessibilité symbolique* **Reach**$(S, (q_0, \phi_0))$ *est effectivement définissable dans Presburger.*

Preuve : Soient $S = \langle Q, E, X\rangle$ un système à n compteurs et $TS(S) = \langle \mathbb{N}^{n+1}, E, \rightarrow\rangle$ le système de transitions qui lui est associé. Supposons que la relation $\rightarrow^*$ soit effectivement définissable dans Presburger et soit $\phi_{\rightarrow} \in$ **Presb**(X_0, X_0') la formule de Presburger vérifiant $\rightarrow^* = [\![\phi_{\rightarrow}]\!]_{X_0, X_0'}$. Soit $(q_0, \phi_0) \in Q \times$ **Presb**(X) une configuration symbolique initiale. Nous l'encodons dans une formule $\phi_0' \in$ **Presb**(X_0). Supposons que $X_0 = \{x_0, \ldots, x_n\}$. Nous définissons la formule $\phi = \exists x_0.\exists x_1.\ldots.\exists x_n.\phi_0' \wedge \phi_{\rightarrow}$. Nous avons alors $\phi \in$ **Presb**(X_0') et de plus **Reach**$(S, (q_0, \phi_0)) = [\![\phi]\!]_{X_0'}$. En effet si $\mathbf{v}' \in \mathbb{N}^{n+1}$, nous avons les équivalences suivantes :

$$\begin{array}{rcl} \mathbf{v}' \in \textbf{Reach}(S, (q_0, \phi_0)) & \Leftrightarrow & \exists \mathbf{v} \in [\![\phi_0']\!] \text{ such that } \mathbf{v}' \rightarrow^* \mathbf{v}' \\ & \Leftrightarrow & \exists \mathbf{v} \in [\![\phi_0']\!] \wedge (\mathbf{v}, \mathbf{v}') \in [\![\phi_{\rightarrow}]\!] \\ & \Leftrightarrow & \mathbf{v}' \in [\![\phi]\!] \end{array}$$

□

De par les propositions 1.33 et 1.35, nous obtenons facilement le corollaire suivant :

Corollaire 1.36 *Le problème d'accessibilité symbolique généralisé est décidable pour les systèmes à n compteurs S dont le système de transitions associé $TS(S) = \langle \mathbb{N}^{n+1}, E, \rightarrow\rangle$ est tel que la relation $\rightarrow^*$ est effectivement définissable dans Presburger.*

1.3.3 Quelques résultats d'indécidabilité et de décidabilité

1.3.3.1 Indécidabilité

Dans [Min67], Minsky prouva que les machines de Minsky à deux compteurs étaient équivalentes aux machines de Turing. Or, de nombreux problèmes, comme par exemple le fameux problème de l'arrêt, sont indécidables pour les machines de Turing, même lorsque l'on considère uniquement des machines de Turing déterministes. Pour une machine de Minsky déterministe, le *problème de l'arrêt* peut être défini de la façon suivante :

Entrées : Une machine de Minsky déterministe $S = \langle Q, X, E \rangle$ et une configuration initiale $c_0 \in Q \times \mathbb{N}^X$;
Question : Existe-t-il $(q, \mathbf{v}) \in Q \times \mathbb{N}^X$ tel que $(q, \mathbf{v}) \in \mathbf{Reach}(S, c_0)$ et tel que $\mathbf{Reach}(S, (q, \mathbf{v})) = \emptyset$.

Minsky prouva alors le théorème suivant :

Théorème 1.37 *[Min67] Le problème de l'arrêt des machines de Minsky déterministes à deux compteurs est indécidable.*

Remarquons qu'il est ensuite facile de réduire le problème de l'arrêt au problème d'accessibilité d'un état de contrôle, ceci car la machine considérée est déterministe, par conséquent l'unique façon pour que $\mathbf{Reach}(S, (q, \mathbf{v})) = \emptyset$ est qu'il n'y ait pas de transition partant de l'état q. Le problème de l'arrêt revient donc à vérifier si il existe un état $q \in Q$ accessible dans (S, c_0) tel qu'il n'y a aucune transition partant de q. De la même façon, il est aussi possible de réduire le problème de l'arrêt au problème d'accessibilité d'une configuration, en faisant décroître tous les compteurs jusqu'à zéro pour chaque état $q \in Q$ n'ayant aucune transition sortante et en testant ensuite si la configuration $(q, (0, 0))$ est accessible. Ceci nous permet de déduire que :

Corollaire 1.38 *Les problèmes d'accessibilité d'un état de contrôle et d'une configuration sont indécidables pour les machines de Minsky déterministes à deux compteurs.*

De plus ce théorème implique que tous les problèmes d'accessibilité présentés précédemment sont indécidables pour les machines à compteurs et les systèmes à compteurs linéaires, fonctionnels et relationnels. Il est vrai que nous avons donné des conditions suffisantes, pour décider des problèmes d'accessibilité, concernant l'ensemble d'accessibilité d'un système à compteurs, mais il s'avère que ces conditions sont indécidables pour des machines de Minsky à deux compteurs, en effet :

Théorème 1.39 *[Bar05] Savoir si l'ensemble d'accessibilité d'une machine de Minsky à deux compteurs initialisée est définissable dans Presburger est un problème indécidable.*

Dans [Bar05], la preuve est faite pour les machines à 4 compteurs, nous proposons ici une version avec 2 compteurs.

Preuve : Nous réduisons le problème d'accessibilité d'un état de contrôle dans les machines de Minsky à deux compteurs. Soient $S_1 = \langle Q_1, X, E_1 \rangle$ une machine de Minsky à deux compteurs, $c_0 = (q_0, \mathbf{v}_0)$ une configuration initiale de $TS(S_1)$ et $q_f \in Q_1$. À partir de S_1, nous construisons une nouvelle machine de Minsky à deux compteurs $S_2 = \langle Q_2, X, E_2 \rangle$ comme indiqué à la figure 1.11. La particularité du système S_2 est que $\mathbf{Reach}(S_2, c_0) \cap \{(q_4, (m, 0)) \mid m \in \mathbb{N}\} = \{(q_4, (2^n, 0)) \mid n \in$

$\mathbb{N} \setminus \{0\}\}$. Nous voyons ainsi apparaître l'ensemble des puissances de 2 qui n'est pas définissable dans Presburger (cf lemme 1.7). Nous complétons encore S_2 en ajoutant un état de contrôle p de façon à obtenir une dernière machine de Minsky à deux compteurs $S_3 = \langle Q_2 \cup \{p\}, X, E_3 \rangle$ avec

$$\begin{aligned} E_3 = \quad & E_2 \cup \{(q_f, \mathbf{inc}(x_1), p)\} \\ & \cup \{(p, \mathbf{inc}(x_1), p), (p, \mathbf{dec}(x_1), p), (p, \mathbf{inc}(x_2), p), (p, \mathbf{dec}(x_2), p)\} \\ & \cup \{(p, \mathbf{ifzero}(x_1), q) \mid q \in Q_2\} \cup \{(p, \mathbf{inc}(x_1), q) \mid q \in Q_2\} \end{aligned}$$

Ainsi, par construction de (S_2, c_0), nous pouvons déduire que q_f est accessible dans S_0 si et seulement si il est accessible dans (S_0, c_0). De plus, par construction de E_3, nous avons que si q_f est accessible dans (S_3, c_0) alors $\mathbf{Reach}(S_3, c_0) = Q_3 \times \mathbb{N}^2$. Ainsi si q_f est accessible dans (S_3, c_0), $\mathbf{Reach}(S_3, c_0)$ est définissable dans Presburger. Et si q_f n'est pas accessible, nous avons la même propriété que pour S_2 concernant les configurations accessibles dans l'état q_4 à savoir que $\mathbf{Reach}(S_3, c_0) \cap \{(q_4, (m, 0)) \mid m \in \mathbb{N}\} = \{(q_4, (2^n, 0)) \mid n \in \mathbb{N} \setminus \{0\}\}$, ce qui implique que si q_f n'est pas accessible dans (S_3, c_0), $\mathbf{Reach}(S_3, c_0)$ n'est pas définissable dans Presburger. Nous en concluons donc que $\mathbf{Reach}(S_3, c_0)$ est définissable dans Presburger si et seulement si q_f est accessible dans (S_1, c_0) qui est un problème indécidable d'après le théorème 1.37. □

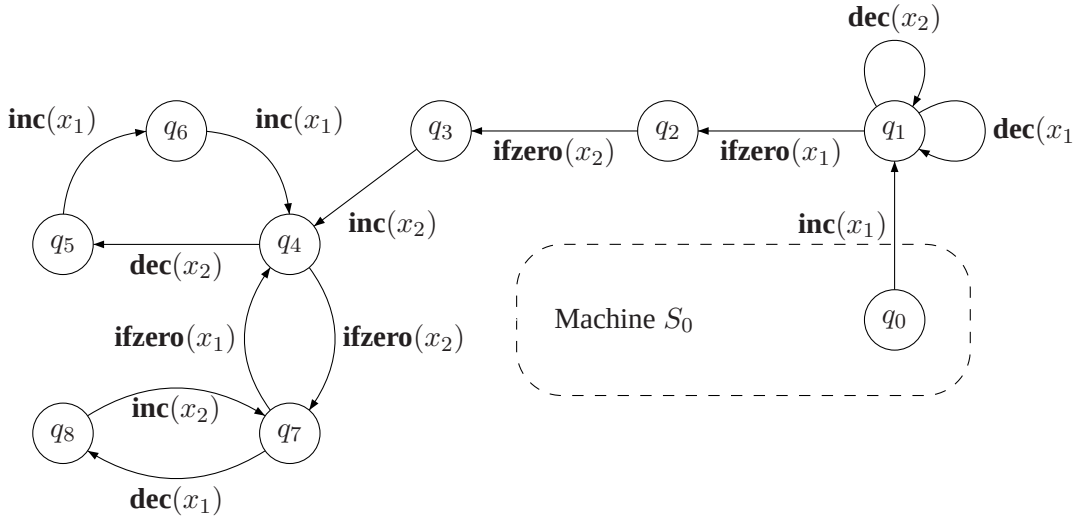


FIGURE 1.11 – Machine de Minsky à deux compteurs S_2

Nous obtenons alors aussi le résultat suivant :

Corollaire 1.40 *Soient S une machine de Minsky à deux compteurs et (q_0, ϕ_0) une configuration symbolique de $TS(S)$. Savoir si l'ensemble d'accessibilité symbolique* $\mathbf{Reach}(S, (q_0, \phi_0))$ *est définissable dans Presburger est indécidable.*

De la même façon, nous avons également le théorème d'indécidabilité suivant concernant la relation d'accessibilité d'une machine de Minsky à deux compteurs :

Théorème 1.41 *Savoir si la relation d'accessibilité d'une machine de Minsky à deux compteurs est définissable dans Presburger est indécidable.*

Preuve : Là encore, nous réduisons le problème d'accessibilité d'un état de contrôle dans les machines de Minsky à deux compteurs. Soient $S_1 = \langle Q_1, X, E_1 \rangle$ une machine de Minsky à deux compteurs, $c_0 = (q_0, (0,0))$ une configuration initiale de $TS(S_1)$ et $q_f \in Q_0$. Remarquons que nous supposons que les valeurs des compteurs de la configuration initiale sont toutes à zéro, mais cela se fait sans perte de généralités, car il est toujours possible de simuler une machine de Minsky initialisée par une machine de Minsky initialisée avec toutes les valeurs de compteurs à zéro. Nous considérons ensuite la machine à deux compteurs $S_3 = \langle Q_3, X, E_3 \rangle$ construite à partir de S_1 dans la preuve du théorème 1.39. Et nous construisons une quatrième machine de Minsky à deux compteurs $S_4 = \langle Q_3 \cup \{p', p''\}, X, E_4 \rangle$ de telle sorte que :

$$\begin{array}{rl} E_4 = & E_3 \cup \{(q, \textbf{inc}(x_1), p') \mid q \in Q_3\} \\ & \cup \{(p', \textbf{dec}(x_1), p'), (p', \textbf{ifzero}(x_1), p'')\} \\ & \cup \{(p'', \textbf{dec}(x_1), p''), (p'', \textbf{dec}(x_2), p''), (p'', \textbf{ifzero}(x_2), q_0)\} \end{array}$$

Soit $TS(S_4) = \langle Q_4 \times \mathbb{N}^2, E_4, \rightarrow \rangle$ le système de transitions associé à S_4. L'idée derrière cette construction est que pour toute configuration $c \in Q_4 \times \mathbb{N}^2$, nous avons $c_0 \in \textbf{Reach}(S_4, c)$. De plus, en utilisant les propriétés de S_3 et S_4, si q_f est accessible dans (S_4, c_0) alors $\textbf{Reach}(S_4, c_0) = (Q_4 \setminus \{p''\} \times \mathbb{N}^2) \cup (\{p''\} \times \{0\} \times \mathbb{N})$. Nous en déduisons que si q_f est accessible :

$$\begin{array}{rl} \rightarrow^* = & (Q_4 \times \mathbb{N}^2) \times (Q_4 \setminus \{p''\} \times \mathbb{N}^2) \\ & \cup (Q_4 \times \mathbb{N}^2) \times (\{p''\} \times \{0\} \times \mathbb{N}) \\ & \cup \{((p'', n, m), (p'', n', m')) \mid n' \leq n \wedge m' \leq m\} \end{array}$$

Ainsi si q_f est accessible, la relation $\rightarrow^*$ est définissable dans Presburger. De même que pour S_3, si q_f n'est pas accessible alors $\textbf{Reach}(S_4, c_0) \cap \{(q_4, (m, 0)) \mid m \in \mathbb{N}\} = \{(q_4, (2^n, 0)) \mid n \in \mathbb{N} \setminus \{0\}\}$ qui n'est pas définissable dans Presburger, par conséquent la relation $\rightarrow^*$ n'est pas non plus définissable dans Presburger. Si nous récapitulons, nous avons que q_f est accessible dans (S_4, c_0) si et seulement si la relation d'accessibilité $\rightarrow^*$ est définissable dans Presburger. Comme de plus, q_f est accessible dans (S_4, c_0) si et seulement si il est accessible dans (S_1, c_0), nous en déduisons le résultat d'indécidabilité. □

Remarquons que ces théorèmes d'indécidabilité sont également valables pour les machines à compteurs et pour les systèmes à compteurs fonctionnels et relationnels, car une machine de Minsky à deux compteurs est un cas particulier de ces différents modèles.

1.3.3.2 Décidabilité

Les preuves des résultats d'indécidabilité précédents utilisent à chaque fois des machines de Minsky à deux compteurs, mais,si l'on restreint le modèle en considérant des SAVE ou en n'utilisant qu'un seul compteur, on obtient des résultats de décidabilité. Ainsi, en ce qui concerne les SAVE, le résultat suivant a été prouvé :

Théorème 1.42 *[Kos82, May84] Le problème d'accessibilité d'une configuration est décidable pour les SAVE.*

Ce théorème est extrèmement important pour la vérification des systèmes à compteurs, car les SAVE constituent une classe non triviale de systèmes. De plus, ce résultat montre que les tests d'égalité ajoutent beaucoup de puissance. En effet, lorsque l'on a une machine à compteurs sans test d'égalité (ie un SAVE) alors le problème d'accessibilité d'une configuration devient décidable. Notons que

pour les SAVE, on ne connaît pas encore la complexité exacte de ce problème, mais on sait qu'il est EXPSPACE-difficile [CLM76].

Le résultat d'indécidabilité du théorème 1.37 fonctionne avec deux compteurs, mais lorsque les machines manipulent un unique compteur, il n'est plus vrai. En effet, les machines de Minsky à un compteur peuvent être vues comme des restrictions d'automates à pile avec un alphabet de pile à deux lettres dont une des deux est utilisée seulement pour marquer le bas de la pile . Un automate à pile est une structure de contrôle finie dont les transitions peuvent empiler une lettre sur le sommet de la pile où dépiler la lettre se trouvant au sommet. Ainsi si l'alphabet de pile ne contient que deux lettres, par exemple $\perp$ et a, incrémenter un compteur revient à empiler un a, décrémenter à dépiler un a et le test à zéro peut être réalisé en dépilant la lettre $\perp$ et en l'empilant de nouveau juste après, bien entendu il faut prendre soin d'empiler le symbole $\perp$ au début. Dans [MS85], les auteurs ont prouvé que la logique monadique du second ordre était décidable pour les systèmes de transition associés aux automates à pile, ce qui implique en particulier que le problème d'accessibilité d'un état de contrôle est décidable pour les automates à pile (la logique ne parlant pas des configurations). Plus tard, dans [Cau92], Caucal montra que, pour chaque état de contrôle, l'ensemble des contenus possibles de la pile au cours de l'exécution est reconnu par un automate fini, qu'il est possible de calculer. En utilisant, le fait que l'image de Parikh d'un langage régulier est un ensemble semi-linéaire dont on peut calculer la formule de Presburger correspondante (cf lemme 1.2), on en déduit aisément le théorème suivant :

Théorème 1.43 *[Cau92] L'ensemble d'accessibilité d'une machine à un compteur initialisée est effectivement définissable dans Presburger.*

Nous verrons par la suite qu'il existe d'autres classes de systèmes à compteurs pour lesquelles certains problèmes d'accessibilité sont décidables.

1.4 Systèmes à compteurs linéaires plats ou aplatissables

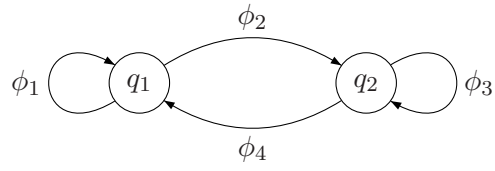
1.4.1 Accélération dans les systèmes à compteurs linéaire plats

Nous avons vu que les problèmes d'accessibilité étaient indécidables pour les machines de Minsky à deux compteurs, mais qu'il était possible d'obtenir la décidabilité en faisant des restrictions sur le modèle considéré en prenant par exemple des SAVE ou des machines à un compteur. Nous montrons maintenant qu'il est possible d'avoir la décidabilité des problèmes d'accessibilité considérés précédemment en utilisant d'autres restrictions.

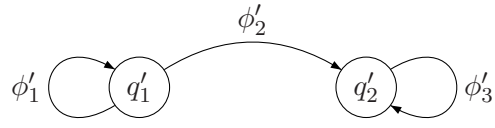
Dans [CJ98], les auteurs ont étudié une sous-classe des systèmes à compteurs pour laquelle les formules étiquetant les transitions sont des conjonctions de formules atomiques de la forme $x\#y' + c$ ou $x\#y + c$ ou $x'\#y' + c$ ou $x\#c$ ou $x'\#c$ où x, y sont des variables de compteurs, $c \in \mathbb{Z}$ et $\# \in \{<, \leq, =, \geq, >\}$. Ils ont montré que si le graphe associé à la structure de contrôle du système à compteurs ne contient pas de boucle imbriquée, certains problèmes d'accessibilité deviennent décidables. C'est ainsi que la classe des systèmes à compteurs plats, c'est-à-dire sans boucle imbriquée, a été introduite. Plus tard, dans [FL02], Finkel et Leroux se sont intéressés à la classe des systèmes à compteurs linéaires plats. Nous rappelons dans cette partie les résultats qu'ils ont alors obtenus.

Dans un premier temps, nous rappelons la définition de systèmes à compteurs plats. Soit $S = \langle Q, X, E\rangle$ un système à compteurs. Nous appelons cycle élémentaire de E une séquence de transitions (q_1, ϕ_1, q'_1) $(q_2, \phi_2, q'_2) \dots (q_m, \phi_m, q'_m)$ telle que pour tout $i \in [1, m-1]$, $q'_i = q_{i+1}$ et $q'_m = q_1$ et pour tout $i, j \in [1..m]$, si $i \neq j$ alors $q_i \neq q_j$. Ainsi un cycle élémentaire d'un système à compteurs correspond à une boucle dans le graphe de la structure de contrôle dans laquelle le seul noeud visité deux fois est le noeud de départ (qui correspond au noeud d'arrivée).

Definition 1.44 (Système à compteurs plat) *[CJ98] Un système à compteur $S = \langle Q, X, E\rangle$ est plat, si pour tout état de contrôle $q \in Q$, q apparaît dans au plus un cycle élémentaire.*



Système à compteurs non plat



Système à compteurs plat

FIGURE 1.12 – Systèmes à compteurs non plats et plats

Exemple 1.45 *La figure 1.12 donne deux exemples, un exemple de système à compteurs non plat et un exemple de système à compteurs plat. Le premier système à compteur de cette figure est non plat car les états q_1 et q_2 appartiennent chacun à deux cycles élémentaires. En effet, si l'on prend par exemple l'état q_1, il appartient au cycle élémentaire (q_1, ϕ_1, q_1) et au cycle élémentaire $(q_1, \phi_2, q_2), (q_2, \phi_4, q_1)$.*

Dans [FL02], les auteurs ajoutent une autre hypothèse, en plus de la platitude, sur les systèmes à compteurs qu'ils analysent de façon à obtenir une relation d'accessibilité définissable dans Presburger. Ils considèrent ainsi le monoïde des matrices d'un système à compteurs linéaire.

Definition 1.46 (Monoïde d'un système à compteurs linéaire) *Le monoïde d'un système à n compteurs linéaire $S = \langle Q, X, E\rangle$ est le monoïde $< S >$ engendré multiplicativement par l'ensemble de matrices $\{A \in \mathcal{M}_n(\mathbb{Z}) \mid \exists (q, (\psi, A, b), q') \in Q \times \mathbf{Presb}(X) \times \mathbb{Z}^n \times Q$ tel que $(q, (\psi, A, \boldsymbol{b}), q') \in E\}$.*

Nous dirons que S est un système à compteurs linéaire à monoïde fini si $< S >$ est fini. Avec cette définition, on peut ainsi donner une condition suffisante sur les systèmes à compteurs linéaires, qui permet d'assurer que la relation d'accessibilité est semi-linéaire. En effet :

Théorème 1.47 *[FL02] Si S est un système à compteurs linéaire plat et à monoïde fini, alors la relation d'accessibilité de S est effectivement définissable dans Presburger.*

Remarquons que les conditions suffisantes proposées par le théorème précédent sont décidables. En effet, comme un système à compteurs a un nombre fini d'états de contrôle et de transitions, il est facile de vérifier si il est plat. Quant à la finitude du monoïde du système à compteurs, elle peut aussi être décidée [MS77, Ler03, Bar05]. De plus, par les propositions 1.35 et 1.34, on en déduit le corollaire suivant :

Corollaire 1.48 *Le problème d'accessibilité symbolique généralisé est décidable pour les systèmes à compteurs linéaires plats à monoïde fini.*

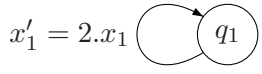


FIGURE 1.13 – Système à compteurs S_1 linéaire plat mais à monoïde infini

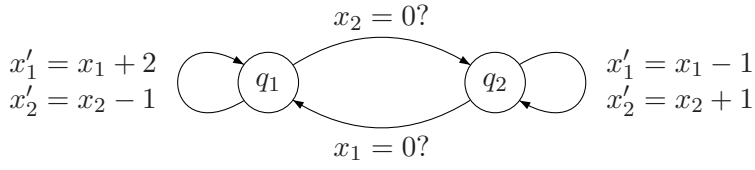


FIGURE 1.14 – Système à compteurs S_2 linéaire à monoïde fini mais non plat

Remarquons que les hypothèses du théorème 1.47 ne peuvent pas être supprimées. En effet, le système à compteurs S_1 représenté à la figure 1.13 est plat mais son monoïde n'est pas fini et sa relation d'accessibilité n'est pas définissable dans Presburger car $\mathbf{Reach}(S_1, (q_1, 0)) = \{(q_1, 2^m) \mid m \in \mathbb{N}\}$ (cf lemme 1.7). Remarquons de plus que ce système travaille sur un unique compteur, ce qui prouve que le résultat du théorème 1.43 n'est plus valable lorsque l'on considère la classe des systèmes linéaires à un compteur. De la même façon, le système à compteurs S_2 de la figure 1.14 est à monoïde fini mais il n'est pas plat, et sa relation d'accessibilité n'est pas définissable dans Presburger car l'ensemble $\mathbf{Reach}(S_2, (q_1, (0, 1)) \cap \{(q_1, (m, 0)) \mid m \in \mathbb{N}\} = \{(q_1, (2^n, 0)) \mid m \in \mathbb{N}\}$ n'est pas définissable dans Presburger.

Finalement, pour certaines classes de systèmes à compteurs, l'hypothèse de platitude suffit pour obtenir le résultat du théorème 1.47, en effet :

Proposition 1.49 *Les machines à compteurs sont des systèmes à compteurs linéaires ayant un monoïde fini.*

Preuve : Soit $S = \langle Q, X, E \rangle$ une machine à n compteurs. Comme nous l'avons vu précédemment les machines à compteurs sont une sous-classe de systèmes à compteurs linéaires dont les transitions sont étiquetées par des translations gardées. Par conséquent pour chaque transition $(q, (\psi, A, \mathbf{b}), q') \in E$, on a $A = \mathcal{I}d_n$. On en déduit alors que $< S >= \{\mathcal{I}d_n\}$. □

1.4.2 Systèmes à compteurs aplatissables

Le théorème 1.47 fournit des conditions nécessaires et suffisantes sur des systèmes à compteurs linéaires pour obtenir une relation d'accessibilité définissable dans Presburger, mais la classe des systèmes vérifiant ces propriétés est assez restreinte, en particulier en ce qui concerne l'hypothèse de platitude. De plus, si le système à compteurs ne vérifie pas ces hypothèses, ce théorème ne fournit aucune piste pour l'analyser. Ainsi dans [BFLS05, Bar05], la notion d'aplatissement a été introduite. Intuitivement, la méthode proposée est la suivante : elle consiste à essayer de construire un système à compteurs plat à partir d'un système à compteurs quelconque de telle sorte que les deux systèmes aient le même ensemble d'accessibilité à partir d'une configuration (symbolique) donnée.

Dans un premier temps, nous définissons la notion d'aplatissement d'un système à compteurs.

Definition 1.50 (Aplatissement d'un système à compteurs) *[BFLS05, Bar05] Un système à compteurs $S' = \langle Q', X, E' \rangle$ est un aplatissement d'un système à compteurs $S = \langle Q, X, E \rangle$ si :*

- *S' est un système à compteurs plat, et,*
- *il existe une fonction $z : Q' \to Q$, appelée repliage, telle que pour tout $(q, \phi, q') \in E'$ alors $(z(q), \phi, z'(q')) \in E$.*

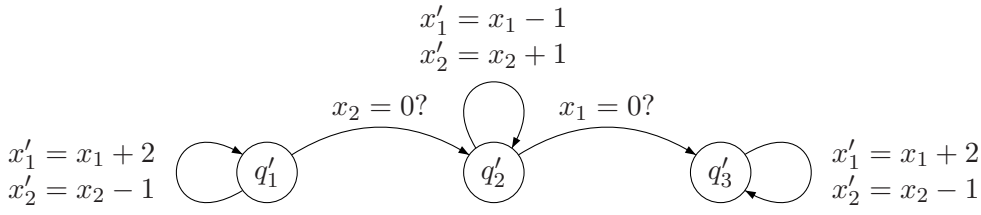


FIGURE 1.15 – Aplatissement du système à compteurs S_2

Exemple 1.51 *La figure 1.15 donne un aplatissement, parmi une infinité d'autres, du système à compteurs linéaire S_2 (figure 1.14). La fonction z de repliage correspondante est définie de la façon suivante : $z(q'_1) = z(q'_3) = q_1$ et $z(q'_2) = q_2$.*

Une méthode pour analyser un système à compteurs linéaire S consiste donc à chercher des aplatissements de S ayant le même ensemble d'accessibilité que S pour une configuration (symbolique) initiale donnée. Nous caractérisons ainsi les systèmes à compteurs linéaires pour lesquels de tels aplatissement existent et ceci en fonction du problème considéré. Dans un premier temps, nous donnons une définition pour les systèmes à compteurs munis d'une configuration initiale.

Definition 1.52 (Système à compteurs aplatissable) *[BFLS05] Soit $S = \langle Q, X, E \rangle$ un système à compteurs et $(q_0, \mathbf{v}_0) \in Q \times \mathbb{N}^X$ une configuration initiale de $TS(S)$. $(S, (q_0, \mathbf{v}_0))$ est aplatissable si il existe un aplatissement $S' = \langle Q', X, E' \rangle$ de S avec une fonction de repliage $z : Q' \to Q$ et un état de contrôle $q'_0 \in Q'$ de $TS(S')$ tels que $z(q'_0) = q_0$ et* **Reach**$(S, (q_0, \mathbf{v}_0)) = z($**Reach**$(S', (q'_0, \mathbf{v}_0)))$.

Nous étendons ensuite cette définition aux cas où le système est muni, non plus d'une simple configuration initiale, mais d'une configuration symbolique initiale.

Definition 1.53 (Système à compteurs symboliquement aplatissable) *Soit $S = \langle Q, X, E\rangle$ un système compteurs et $(q_0, \phi_0) \in Q \times$ **Presb**(X) une configuration symbolique initiale. $(S, (q_0, \phi_0))$ est symboliquement aplatissable si il existe un aplatissement $S' = \langle Q', X, E'\rangle$ de S avec une fonction de repliage $z : Q' \to Q$ et un état de contrôle $q'_0 \in Q'$ tels que $z(q'_0) = q_0$ et* **Reach**$(S, (q_0, \phi_0))$ $=$ **Reach**$(S', (q'_0, \phi_0))$.

En utilisant, le théorème 1.47 et la proposition 1.35, on en déduit la proposition suivante :

Proposition 1.54 *Soient S un système à compteurs à monoïde fini, c_0 une configuration initiale de $TS(S)$ et (q_0, ϕ_0) une configuration symbolique initiale de $TS(S)$.*

1. *Si (S, c_0) est aplatissable alors* **Reach**(S, c_0) *est définissable dans Presburger.*
2. *Si $(S, (q_0, \phi_0))$ est symboliquement aplatissable, alors* **Reach**$(S, (q_0, \phi_0))$ *est définissable dans Presburger.*

Avec les propositions 1.33 et 1.34, nous obtenons le théorème suivant caractérisant les problèmes d'accessibilité décidables selon la notion d'aplatissabilité prise en compte.

Théorème 1.55

1. *Le problème d'accessibilité symbolique est décidable pour les systèmes à compteurs à monoïde fini munis d'une configuration initiale qui sont aplatissables.*
2. *Le problème d'accessibilité symbolique généralisé est décidable pour les systèmes à compteurs à monoïde fini munis d'une configuration symbolique initiale qui sont symboliquement aplatissables.*

Cependant, une fois encore, en utilisant l'indécidabilité du problème de l'arrêt pour les machines de Minsky à deux compteurs, on peut prouver que les conditions suffisantes proposées dans le théorème précédent sont indécidables. En effet :

Théorème 1.56 *[Bar05] Soient S une machine de Minsky à deux compteurs et c_0 une configuration initiale de $TS(S)$. Savoir si (S, c_0) est aplatissable est indécidable.*

Preuve : Soient $S_1 = \langle Q_1, X, E_1\rangle$ une machine de Minsky à deux compteurs, $c_0 = (q_0, \mathbf{v}_0)$ une configuration initiale de $TS(S_1)$ et $q_f \in Q_1$. Nous reprenons la machine de Minsky à deux compteurs S_3 construite à partir de S_1 dans la preuve du théorème 1.39. D'abord, comme indiqué dans la preuve du théorème 1.39, remarquons que si q_f n'est pas accessible dans (S_3, c_0) alors, d'après la proposition 1.54, (S_3, c_0) n'est pas aplatissable, ceci car **Reach**(S_3, c_0) n'est pas définissable dans Presburger. Supposons maintenant que q_f est accessible dans (S_3, c_0). Alors il existe une exécution dans $TS(S_3)$ de la forme $(q_0, \mathbf{v}_0) \xrightarrow{e_0} (q_0, \mathbf{v}_0) \xrightarrow{e_1} \ldots (q_m, \mathbf{v}_m) \xrightarrow{e_m} (q_{m+1}, \mathbf{v}_{m+1})$ avec $e_i = (q_i, \phi_i, q_{i+1})$ pour tout $i \in [0..m]$ et $q_{m+1} = q_f$. À partir de cette observation et de la machine à compteurs $S_5 = \langle Q_5, X, E_5\rangle$ représentée à la figure 1.16, nous construisons la machine à deux compteurs $S_6 = \langle Q_6, X, E_6\rangle$ de telle sorte que $Q_6 = Q_5 \uplus Q_3 \uplus \{q'_1, \ldots, q'_m, q'_{m+1}\}$ et :

$$\begin{aligned} E_6 = \quad & E_5 \cup \{(q'_i, \phi_i, q'_{i+1}) \mid i \in [1..m]\} \\ & \cup\{(q'_{m+1}, \mathbf{inc}(x_2), p_1), (q'_{m+1}, \mathbf{inc}(x_2), p_2)\} \\ & \cup\{(p_3, \mathbf{inc}(x_1), q) \mid q \in Q_3\} \cup \{(p_3, \mathbf{ifzero}(x_1), q) \mid q \in Q_3\} \\ & \cup\{(p_4, \mathbf{inc}(x_1), q) \mid q \in Q_3\} \cup \{(p_4, \mathbf{ifzero}(x_1), q) \mid q \in Q_3\} \end{aligned}$$

Dans un premier temps remarquons que S_6 est une machine de Minsky à deux compteurs plate, les seuls cycles élémentaires existants étant représentés sur la figure 1.16. Ensuite cette machine est un aplatissement de la machine à compteurs S_3, la fonction de repliage z étant donnée par les règles suivantes :

– $\forall q \in Q_3, z(q) = q$, et,
– $\forall i \in [1..m], z(q'_i) = q_i$, et,
– $z(p_1) = z(p_2) = z(p_3) = z(p_4) = p$.

Pour rappel p est un état spécial de Q_3 (cf preuve du théorème 1.39). Finalement, nous avons que $z(\mathbf{Reach}(S_6, (q'_0, c_0))) = Q_3 \times \mathbb{N}^2$ car q'_{m+1} est accessible dans S_6 (par construction) et à partir de q'_{m+1}, le système passe dans les états de la machine à compteurs S_5 qui parcourt tous les éléments de $\mathbb{N}^2$. Or comme $\mathbf{Reach}(S_3, (q_0, c_0)) = Q_3 \times \mathbb{N}^2$ lorsque q_f est accessible dans (S_3, c_0), on en déduit que $(S_3, (q_0, c_0))$ est aplatissable si et seulement si q_f est accessible dans $(S_3, (q_0, c_0))$, c'est-à-dire si et seulement si q_f est accessible dans $(S_0, (q_0, c_0))$. □

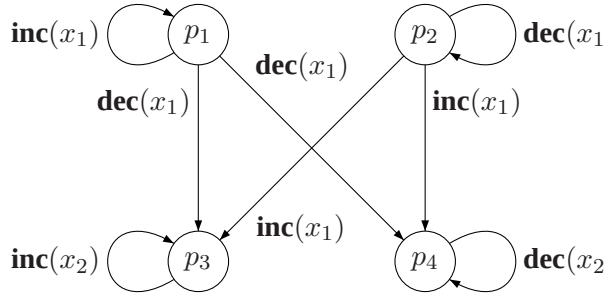


FIGURE 1.16 – Machine à compteurs plate S_5 pour construire un aplatissement de S_3

Nous avons de plus le corollaire suivant :

Corollaire 1.57 *Soient S une machine de Minsky à deux compteurs et (q_0, ϕ_0) une configuration symbolique initiale de $TS(S)$. Savoir si $(S, (q_0, \phi_0))$ est symboliquement aplatissable est indécidable.*

Malgré ces résultats d'indécidabilité, dans [LS05], les auteurs ont montré que de nombreuses sous-classes de systèmes à compteurs linéaires étaient aplatissables, prouvant ainsi la pertinence de cette notion. En particulier, ils ont montré que les machines à compteurs initialisées *reversal*-bornées (que nous définirons plus tard dans cette thèse) sont aplatissables, tout comme les SAVE à deux compteurs et d'autres sous-classes de SAVE que nous ne détaillons pas ici.

1.4.3 L'algorithme de l'outil FAST

Bien que le problème de savoir si un système à compteurs est aplatissable soit indécidable, il est toujours possible de construire un aplatissement d'un système à compteurs S et de tester si cet aplatissement a ou n'a pas le même ensemble d'accessibilité que S. Comme les relations qui étiquettent les transitions d'un système à compteurs sont des formules de Presburger, nous avons le lemme suivant :

Lemme 1.58 *Soient $S = \langle Q, X, E\rangle$ un système à n compteurs et $C \subseteq \mathbb{N}^{n+1}$ un ensemble de configurations. Si C est effectivement définissable dans Presburger alors* **Post**(S, C) *est aussi effectivement définissable dans Presburger.*

Nous donnons maintenant une proposition permettant de caractériser les aplatissements d'un système à compteurs ayant le même ensemble d'accessibilité que le système à partir duquel ils sont obtenus.

Lemme 1.59 *Soient $S = \langle Q, X, E\rangle$ un système à n compteurs et $(q_0, \phi_0) \in Q \times$* **Presb**$(X)$ *une configuration symbolique. Si $S' = \langle Q', X, E'\rangle$ est un aplatissement de S tel que pour la fonction de repliage associée $z : Q' \to Q$, il existe $q'_0 \in Q'$ vérifiant $z(q'_0) = q_0$, alors $z($***Reach**$(S', (q'_0, \phi_0))) =$* **Reach**$(S, (q_0, \phi_0))$ *si et seulement si* **Post**$(S, z($**Reach**$(S', (q'_0, \phi_0))) = z($**Reach**$(S', (q'_0, \phi_0)))$.

Preuve : Soit $TS(S) = \langle Q \times \mathbb{N}^X, E, \to\rangle$ le système de transitions associé à S. Si $z($**Reach**$(S', (q'_0, \phi_0))) =$ **Reach**$(S, (q_0, \phi_0))$ alors, comme **Post**$(S,$ **Reach**$(S, (q_0, \phi_0))) =$ **Reach**$(S, (q_0, \phi_0))$, on en déduit que **Post**$(S, z($**Reach**$(S', (q'_0, \phi_0))) = z($**Reach**$(S', (q'_0, \phi_0)))$. Supposons maintenant que **Post**$(S, z($**Reach**$(S', (q'_0, \phi_0))) = z($**Reach**$(S', (q'_0, \phi_0)))$. Remarquons que par définition d'aplatissement, on a $z($**Reach**$(S', (q'_0, \phi_0))) \subseteq$ **Reach**$(S, (q_0, \phi_0))$. Soit $(q, \mathbf{v}) \in$ **Reach**$(S, (q_0, \phi_0))$. Il existe alors dans $TS(S)$ une exécution $(q_0, \mathbf{v}_0) \to (q_1, \mathbf{v}_1) \to \ldots \to (q_m, \mathbf{v}_m)$ avec $\mathbf{v}_0 \models \phi_0$ et $(q_m, \mathbf{v}_m) = (q, \mathbf{v})$. Nous allons montrer que pour tout $i \in [0..m]$, $(q_i, \mathbf{v}_i) \in z($**Reach**$(S', (q'_0, \phi_0)))$. Nous raisonnons par induction. D'abord, remarquons que comme $(q_0, \mathbf{v}_0) = (z(q'_0), \mathbf{v}_0)$, on a $(q_0, \mathbf{v}_0) \in z($**Reach**$(S', (q'_0, \phi_0)))$. Soit $i \in [1..m]$ et supposons que $(q_{i-1}, \mathbf{v}_{i-1}) \in z($**Reach**$(S', (q'_0, \phi_0)))$. Alors $(q_i, \mathbf{v}_i) \in$ **Post**$(S, z($**Reach**$(S', (q'_0, \phi_0)))$ et par conséquent $(q_i, \mathbf{v}_i) \in z($**Reach**$(S', (q'_0, \phi_0))$. Nous en déduisons donc que $(q, \mathbf{v}) \in z($**Reach**$(S', (q'_0, \phi_0))$. □

En utilisant les propriétés énoncées par les deux lemmes précédents, on peut construire un semi-algorithme pour tester si un système à compteurs linéaire S à monoïde fini muni d'une configuration symbolique initiale (q_0, ϕ_0) est symboliquement aplatissable. Ce semi-algorithme consiste à énumérer des aplatissements de S et à vérifier si l'aplatissement courant satisfait les conditions du lemme 1.59.

Algorithme 1.60 Algorithme pour calculer **Reach**$(S, (q_0, \phi_0))$

Entrée : S un système à compteurs linéaire à monoïde fini ;
Entrée : (q_0, ϕ_0) une configuration symbolique initiale de $TS(S)$;
Sortie : C un ensemble de configurations de $TS(S)$;

1: $C = \{(q_0, \mathbf{v}_0) \mid \mathbf{v}_0 \models \phi_0\}$
2: **Tant Que Post**$(S, C) \not\subseteq C$ **Faire**
3: Choisir un aplatissement S' de S avec une fonction de repliage z
4: $C = z($**Reach**$(S', (q'_0, \phi_0))$
5: **Fin Tant Que**
6: **Retourner** C

Pour cet algorithme, nous supposons que l'action "Choisir" utilisée à la ligne 3 se fait de manière équitable, c'est-à-dire que chaque aplatissement de S est choisi infiniment souvent si l'on appelle "Choisir" infiniment souvent. Nous supposons de plus que pour chaque aplatissement choisi, la fonction z de repliage correspondante est définie en q'_0 et $z(q'_0) = q_0$. Remarquons que comme à chaque étape, le système S' est plat et à monoïde fini, d'après le théorème 1.47, nous savons que C est à chaque étape effectivement définissable dans Presburger, par conséquent d'après le lemme 1.58 tester

si **Post**$(S, C) \not\subseteq C$ peut être effectivement réalisé. Notons que cet algorithme ne termine pas toujours, en effet, grâce au lemme 1.59 et à l'hypothèse d'équité sur l'action "Choisir", nous avons la proposition suivante :

Proposition 1.61 *[Bar05] L'algorithme 1.60 termine sur l'entrée* $(S, (q_0, \phi_0))$ *si et seulement le système* $(S, (q_0, \phi_0))$ *est symboliquement aplatissable.*

Ainsi lorsque $(S, (q_0, \phi_0))$ n'est pas symboliquement aplatissable cet algorithme ne termine pas. Cependant lorsqu'il termine, il est correct, en effet :

Proposition 1.62 *[Bar05] Si l'algorithme 1.60 termine sur l'entrée* $(S, (q_0, \phi_0))$*, alors nous avons* $C =$ **Reach**$(S, (q_0, \phi_0))$.

Un point crucial pour améliorer les performances de l'algorithme 1.60 réside dans la façon dont la fonction "Choisir" est programmée. Dans [Bar05], des heuristiques sont proposées pour implanter cette fonction de façon à atteindre un point fixe de manière efficace.

L'algorithme 1.60 est à la base de l'outil FAST [BFLP03, BLP06, BFLP08] qui vérifie automatiquement des systèmes à compteurs linéaires à monoïde fini. Cet outil calcule l'ensemble des états accessibles d'un système à compteurs linéaire à partir d'une configuration symbolique initiale. FAST prend en entrée un fichier contenant une description du système à compteurs à analyser et une stratégie permettant d'influencer le calcul d'accessibilité de façon à améliorer les performances du calcul. Comme cet outil est basé sur l'algorithme 1.60, il se peut que cet outil ne termine pas, cependant il a permis de vérifier avec succès de nombreux systèmes. Dans la dernière version, il est possible de choisir la librairie qui est utilisée pour manipuler les formules de Presburger et un utilisateur peut, si il le souhaite, réimplanter sa propre librairie ; en plus de permettre une certaine flexibilité dans l'utilisation du logiciel, cette fonctionnalité peut aussi être utilisée pour tester l'efficacité d'une librairie.
L'outil FAST est disponible librement à l'adresse `http://www.lsv.ens-cachan.fr/fast/`.

Chapitre 2

Machines à compteurs *reversal*-bornées

Dans ce chapitre, nous étudions une nouvelle classe de machines à compteurs ayant un ensemble d'accessibilité effectivement définissable dans Presburger. Cette classe étend la classe des machines à compteurs *reversal*-bornées introduites par Ibarra dans [Iba78]. Les résultats présentés ici sont pour la plupart issus de [FS08].

2.1 Les machines à compteurs *reversal*-bornées d'Ibarra

2.1.1 Définition

Comme nous l'avons vu au chapitre précédent, la plupart des problèmes d'accessibilité sont indécidables pour les machines à compteurs, à cause de l'indécidabilité du problème de l'arrêt pour les machines de Minsky déterministes à deux compteurs [Min67]. Cependant il est possible de poser des restrictions que ce soit au niveau syntaxique, comme avec les SAVE ou les machines à un compteur, ou au niveau sémantique, avec les systèmes aplatissables, de façon à obtenir la décidabilité. Nous présentons ici une nouvelle restriction sur les machines à compteurs permettant d'obtenir la décidabilité pour certains problèmes d'accessibilité.

Dans [Iba78], Ibarra proposa une restriction sur les machines de Minsky en étudiant les machines pour lesquelles au cours de toute exécution, le nombre d'alternances de chaque compteur entre une phase de croissance et une phase de décroissance est borné. C'est ainsi que furent introduites les machines de Minsky dites *reversal*-bornées ("reversal-bounded" en anglais). Il s'avère que l'ensemble d'accessibilité de telles machines est un ensemble semi-linéaire qui peut effectivement être calculé. Plus tard, dans [ISD$^+$02], les auteurs ont montré qu'il est possible d'étendre cette notion aux machines à compteurs tout en conservant le résultat sur l'ensemble d'accessibilité. Ils ont également prouvé qu'il est difficile d'étendre encore la définition à d'autres systèmes à compteurs car on peut facilement simuler une machine de Minsky à deux compteurs avec une machine à compteurs *reversal*-bornée dans laquelle on autorise des tests de la forme $x_i = x_j$. En effet, chaque compteur de la première machine est encodé par deux compteurs de la deuxième machine, un qui compte les incréments et l'autre les décréments ; pour effectuer les tests à zéro, il suffit de tester si les deux compteurs ont la même valeur ; cette machine est bien *reversal*-bornée car ses compteurs ne font que croître.

Nous rappelons maintenant de façon plus formelle la définition des machines à compteurs *reversal*-bornées d'Ibarra. Soit $S = \langle Q, X, E\rangle$ une machine à n compteurs et $TS(S) = \langle Q \times \mathbb{N}^n, E, \rightarrow\rangle$ le

système de transitions associé. Nous construisons un nouveau système de transitions associé à S dans lequel nous enregistrons dans chaque configuration les informations suivantes :

1. la phase courante, croissante (notée avec le symbole $\uparrow$) ou décroissante (notée avec le symbole $\downarrow$), dans laquelle se trouve chaque compteur, et,
2. le nombre d'alternances qui ont été réalisées (entre les phases croissantes et décroissantes).

Nous définissons ainsi le système de transitions $TS_{rb}(S) = \langle Q \times \{\uparrow,\downarrow\}^n \times \mathbb{N}^n \times \mathbb{N}^n, E, \rightarrow_{rb}\rangle$. Une configuration $(q, \mathbf{m}, \mathbf{v}, \mathbf{r}) \in Q \times \{\uparrow,\downarrow\}^n \times \mathbb{N}^n \times \mathbb{N}^n$ de $TS_{rb}(S)$ donne les informations suivantes :

– q indique l'état de contrôle courant,
– $\mathbf{m}$ indique la phase dans laquelle se trouve chaque compteur,
– $\mathbf{v}$ donne la valeur de chaque compteur, et,
– $\mathbf{r}$ contient le nombre d'alternances réalisées.

Nous définissons maintenant la relation de transition $\rightarrow_{rb}$. Pour tout $(q, \mathbf{m}, \mathbf{v}, \mathbf{r}), (q', \mathbf{m}', \mathbf{v}', \mathbf{r}') \in Q \times \{\uparrow,\downarrow\}^n \times \mathbb{N}^n \times \mathbb{N}^n$ et pour tout $e \in E$, on a $(q, \mathbf{m}, \mathbf{v}, \mathbf{r}) \xrightarrow{e}_{rb} (q', \mathbf{m}', \mathbf{v}', \mathbf{r}')$ si et seulement si les conditions suivantes sont satisfaites :

1. $(q, \mathbf{v}) \xrightarrow{e} (q', \mathbf{v}')$, et,
2. pour tout $i \in [1..n]$, la relation définie par le tableau qui suit est vérifiée :

$\mathbf{v}(i) - \mathbf{v}'(i)$	$\mathbf{m}(i)$	$\mathbf{m}'(i)$	$\mathbf{r}(i)$
> 0	$\downarrow$	$\downarrow$	$\mathbf{r}(i)$
> 0	$\uparrow$	$\downarrow$	$\mathbf{r}(i) + 1$
< 0	$\uparrow$	$\uparrow$	$\mathbf{r}(i)$
< 0	$\downarrow$	$\uparrow$	$\mathbf{r}(i) + 1$
$= 0$	$\downarrow$	$\downarrow$	$\mathbf{r}(i)$
$= 0$	$\uparrow$	$\uparrow$	$\mathbf{r}(i)$

De la même façon que pour la relation de transition $\rightarrow$ de $TS(S)$, nous noterons $\rightarrow^*_{rb}$ la fermeture réflexive et transitive de $\rightarrow_{rb}$. Étant donné une configuration $c = (q, \mathbf{m}, \mathbf{v}, \mathbf{r})$ de $TS_{rb}(S)$, nous définissons également l'ensemble d'accessibilité correspondant dans $TS_{rb}(S)$, $\mathbf{Reach}_{rb}(S, c) = \{(q', \mathbf{m}', \mathbf{v}', \mathbf{r}') \in Q \times \{\uparrow,\downarrow\}^n \times \mathbb{N}^n \times \mathbb{N}^n \mid c \rightarrow^*_{rb} (q', \mathbf{m}', \mathbf{v}', \mathbf{r}')\}$. Nous étendons de plus, cette dernière définition aux configurations de $TS(S)$ en supposant qu'au début de l'exécution la phase est toujours croissante et le nombre d'alternance vaut zéro. Ainsi, si $(q, \mathbf{v}) \in Q \times \mathbb{N}^n$ est une configuration de $TS(S)$, alors $\mathbf{Reach}_{rb}(S, (q, \mathbf{v}))$ est égal à $\mathbf{Reach}_{rb}(S, (q, \uparrow, \mathbf{v}, \mathbf{0}))$ où $\uparrow$ représente ici le vecteur ayant toutes ses composantes égales à $\uparrow$. Nous disposons maintenant des notions suffisantes pour rappeler la définition des machines à compteurs *reversal*-bornées au sens d'Ibarra :

Definition 2.1 (Machine à compteurs *Ibarra-reversal*-bornée) *[Iba78] Soit $k \in \mathbb{N}$. Une machine à n compteurs (S, c_0) est k-reversal-bornée si pour tout $(q, \boldsymbol{m}, \boldsymbol{v}, \boldsymbol{r}) \in \mathbf{Reach}_{rb}(S, c_0)$, pour tout $i \in [1..n]$, nous avons $\boldsymbol{r}(i) \leq k$.*

Ibarra dit ensuite qu'une machine à compteurs (S, c_0) est *Ibarra-reversal*-bornée si il existe une constante $k \in \mathbb{N}$ telle que (S, c_0) est k-*reversal*-bornée. Notons que nous étudions la notion de *reversal*-bornée uniquement pour des machines à compteurs initialisées, mais nous ne rappelons pas à chaque fois qu'elles sont initialisée pour faciliter la lisibilité.

Exemple 2.2 *La machine à compteurs de la figure 2.1 à laquelle on associe la configuration initiale* $(q_1, (0, 0))$ *est Ibarra-reversal-bornée car elle est* 1*-reversal-bornée, en effet le compteur* x_2 *reste toujours dans la phase croissante et le compteur* x_1 *ne change qu'une seule fois de phase, la première fois que la transition bouclant en* q_2 *est franchie. Quant à la machine à compteurs de la figure 2.2 munie de la même configuration initiale, elle n'est pas Ibarra-reversal-bornée, car pour tout entier* $k \in \mathbb{N}$ *il est toujours possible de trouver une exécution pour laquelle le compteur* x_1 *alterne plus de* k *fois (entre une phase croissante et une phase décroissante).*

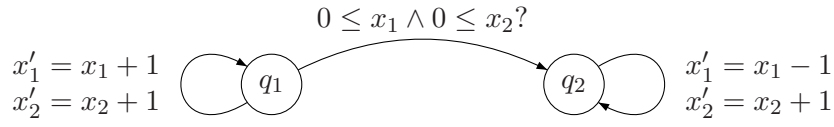


FIGURE 2.1 – Une machine à compteurs *Ibarra-reversal*-bornée avec la configuration initiale $(q_1, (0, 0))$

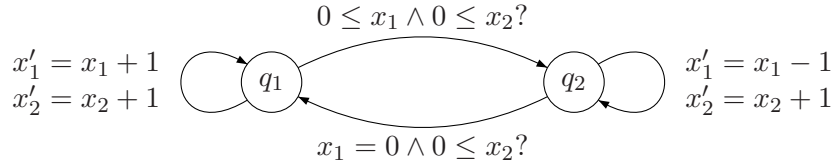


FIGURE 2.2 – Une machine non *Ibarra-reversal*-bornée avec la configuration initiale $(q_1, (0, 0))$

Par rapport à la définition de machines *Ibarra-reversal*-bornées présentée dans [Iba78], nous introduisons ici une petite différence, car nous ne munissons pas les machines à compteurs d'états acceptants. Par conséquent toutes les exécutions de la machine partant de la configuration initiale donnée sont acceptées. Nous verrons dans la section 2.3 que cette différence change certains résultats de décidabilité. Remarquons que dans les derniers travaux sur les machines à compteurs *reversal*-bornées, comme par exemple dans [ISD$^+$02], les machines à compteurs ne sont pas non plus munies d'états acceptants.

2.1.2 Propriétés

Nous montrons maintenant pourquoi la classe des machines à compteurs *Ibarra-reversal*-bornées est digne d'intérêt. Le résultat principal de [Iba78] pour les machines de Minsky, et qui fut ensuite étendu dans [ISD$^+$02], peut être énoncé de la façon suivante :

Théorème 2.3 *Si* (S, c_0) *est une machine à compteurs Ibarra-reversal-bornée, alors l'ensemble d'accessibilité* **Reach**(S, c_0) *est effectivement définissable dans Presburger.*

L'idée de la preuve de ce théorème, que l'on trouve dans [Iba78] pour les machines de Minsky, se résume de la façon suivante. Dans un premier temps, on montre qu'à partir d'une machine à compteurs k*-reversal*-bornée, on peut construire, en ajoutant un certain nombre de compteurs et d'états de

contrôle, une machine à compteurs 1-*reversal*-bornée qui simule la première machine. Ensuite, on encode cette machine à compteurs 1-*reversal*-bornée dans un automate fini et on se sert de l'image de Parikh de cet automate que l'on raffine à l'aide d'autres formules de Presburger pour obtenir l'ensemble d'accessibilité. Ainsi le fait que l'ensemble d'accessibilité soit effectivement définissable dans Presburger est prouvé en utilisant la propriété sur l'image de Parikh des langages réguliers (cf. lemme 1.2), qui est un ensemble semi-linéaire dont on peut effectivement calculer la formule de Presburger correspondante.

D'après ce théorème et en utilisant la proposition 1.33, on en déduit le théorème suivant concernant la vérification de machines à compteurs *reversal*-bornées.

Théorème 2.4 *Le problème d'accessibilité symbolique est décidable pour les machines à compteurs Ibarra-reversal-bornées.*

En revanche, on ne peut rien dire sur le problème d'accessibilité symbolique généralisé car les machines à compteurs *Ibarra-reversal*-bornées sont munies d'une configuration initiale et pas d'une configuration symbolique initiale. Il serait possible d'étendre la définition de *reversal*-bornée aux machines à compteurs munies d'une configuration symbolique initiale mais nous choisissons ici de ne pas traiter ce cas.

Plus tard, dans [LS05], en montrant que différentes sous-classes de systèmes à compteurs à monoïde fini étaient aplatissables, les auteurs ont prouvé le résultat suivant :

Théorème 2.5 *[LS05] Les machines à compteurs Ibarra-reversal-bornées sont aplatissables.*

Les machines à compteurs *Ibarra-reversal*-bornées forment ainsi une sous-classe de systèmes à compteurs aplatissables. Ce dernier résultat nous montre que si l'on donne en entrée à l'outil FAST une machine à compteurs *Ibarra-reversal*-bornée, on est sûr que l'outil va terminer son calcul d'après la proposition 1.61. Remarquons que le théorème 2.3 ne suffit pas pour montrer ce résultat, car il se peut qu'une machine à compteurs à monoïde fini ait un ensemble d'accessibilité effectivement définissable dans Presburger sans qu'elle soit pour autant aplatissable.

Finalement une dernière propriété intéressante des machines à compteurs *Ibarra=reversal*-bornées réside dans le fait qu'elles sont plus robustes que les machines aplattissables. En particulier si l'on considère une sous-machine d'une machine à compteurs *Ibarra-reversal*-bornée, elle reste *Ibarra-reversal*-bornée. Nous approfondissons un peu ce point.

Definition 2.6 (Sous-système à compteurs) *Un système à compteurs $S' = \langle Q', X, E' \rangle$ est un sous-système d'un système à compteurs $S = \langle Q, X, E \rangle$, noté $S' \leq S$, si $Q' \subseteq Q$ et $E' \subseteq E$.*

L'intérêt d'étudier un sous-système plutôt que le système lui-même apparaît lorsque l'on veut analyser des systèmes de grande taille. Nous avons la proposition suivante :

Proposition 2.7 *Soient $S = \langle Q, X, E \rangle$ un système à compteurs et $S' = \langle Q', X, E' \rangle$ un système à compteurs tel que $S' \leq S$. Si $c_0 \in Q' \times \mathbb{N}^X$, alors* **Reach**$(S', c_0) \subseteq$ **Reach**(S, c_0).

D'après cette propsosition, il peut parfois suffire de raisonner sur le système de transitions de S' plutôt que sur celui du système S, en particulier lorsque S' est "plus petit" (en nombre d'états par

exemple) que S, car si une configuration appartient à **Reach**(S', c_0), elle appartient nécessairement à **Reach**(S, c_0). Le résultat suivant, que l'on peut déduire facilement d'après la définition des machines *Ibarra-reversal*-bornées, montre que cette notion de sous-système peut en particulier servir pour résoudre "plus rapidement" parfois des problèmes d'accessibilité dans le cas de telles machines à compteurs, en effet :

Proposition 2.8 *Soient S une machine à compteurs, $S' = \langle Q', X, E' \rangle$ une machine à compteurs telle que $S' \leq S$ et $c_0 \in Q' \times \mathbb{N}^X$. Si (S, c_0) est Ibarra-reversal-bornée alors (S', c_0) est Ibarra-reversal-bornée.*

Cette propriété est d'autant plus intéressante dans le cas des machines *Ibarra-reversal*-bornées, qu'elle n'est plus valable lorsqu'il s'agit de machines à compteurs aplatissables. Par exemple si l'on considère la machine de Minsky de la preuve du théorème 1.56, remarquons que si l'on supprime l'état de contrôle p, cette machine n'est plus forcément aplatissable. Ceci est dû au fait qu'en supprimant un état de contrôle, on supprime également les chemins qui rendent l'aplatissement possible. Ainsi dans certains cas, les machines à compteurs *Ibarra-reversal*-bornées peuvent s'avérer plus simples à manipuler que les machines à compteurs aplatissables.

2.2 La généralisation des machines à compteurs *Ibarra-reversal*-bornées

2.2.1 Définition

Dans [FS08], nous avons étendu la définition des machines à compteurs *Ibarra-reversal*-bornées de façon à pouvoir englober un plus grand nombre de machines à compteurs.

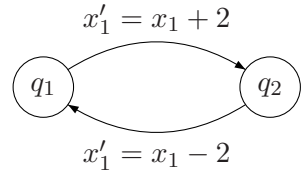


FIGURE 2.3 – Une machine à compteurs non *Ibarra-reversal*-bornée avec la configuration initiale $(q_1, 0)$

Si nous considérons la machine à compteurs représentée à la figure 2.3 munie de la configuration initiale $(q_1, 0)$, son ensemble d'accessibilité est fini et égal à $\{(q_1, 0), (q_2, 2)\}$. Par conséquent, l'ensemble d'accessibilité de cette machine à compteurs est semi-linéaire et cette machine n'est pas *Ibarra-reversal*-bornée, car les compteurs peuvent croître et décroître infiniment souvent. L'extension de la définition de *Ibarra-reversal*-bornée que nous donnons ici permet de prendre en compte de tels cas et plus généralement, elle englobe toutes les machines à compteurs bornées (c'est-à-dire ayant un ensemble fini de configurations accessibles).

Étant donné un entier $b \in \mathbb{N}$, nous allons maintenant, pour chaque compteur, compter le nombre d'alternances entre les phases croissantes et décroissantes uniquement dans les cas où ces alternances ont lieu pour une valeur du compteur strictement supérieure à b. Soit $S = \langle Q, X, E \rangle$ une machine à n compteurs et $TS(S) = \langle Q \times \mathbb{N}^n, E, \rightarrow \rangle$ le système de transitions qui lui est associé. De la même façon que nous avons construit un système de transitions $TS_{rb}(S)$ pour définir les

machines à compteurs *Ibarra-reversal*-bornées, pour chaque entier $b \in \mathbb{N}$, nous définissons un système de transitions $TS_b(S) = \langle Q \times \{\downarrow, \uparrow\}^n \times \mathbb{N}^n \times \mathbb{N}^n, E, \rightarrow_b \rangle$. Là encore, pour une configuration $(q, \mathbf{m}, \mathbf{v}, \mathbf{r}) \in Q \times \{\downarrow, \uparrow\}^n \times \mathbb{N}^n \times \mathbb{N}^n$, les vecteurs **m** et **v** ont la même signification que dans les configurations du système de transitions $TS_{rb}(S)$ défini précédemment, quant au **r**, il contient désormais pour chaque compteur le nombre d'alternances entre les phases croissantes et décroissantes réalisées au dessus de b. Formellement, la relation de transitions $\rightarrow_b$ est définie de la façon suivante, pour tout $(q, \mathbf{m}, \mathbf{v}, \mathbf{r}), (q', \mathbf{m'}, \mathbf{v'}, \mathbf{r'}) \in Q \times \{\downarrow, \uparrow\}^n \times \mathbb{N}^n \times \mathbb{N}^n$, et pour tout $e \in E$ nous avons $(q, \mathbf{m}, \mathbf{v}, \mathbf{r}) \xrightarrow{e}_b (q', \mathbf{m'}, \mathbf{v'}, \mathbf{r'})$ si et seulement si :

1. $(q, \mathbf{v}) \xrightarrow{e} (q', \mathbf{v'})$, et,
2. pour tout $i \in [1..n]$, les conditions données par le tableau suivant sont vérifiées :

$\mathbf{v}(i) - \mathbf{v'}(i)$	$\mathbf{m}(i)$	$\mathbf{m'}(i)$	$\mathbf{v}(i)$	$\mathbf{r}(i)$
≥ 0	$\downarrow$	$\downarrow$	$-$	$\mathbf{r}(i)$
> 0	$\uparrow$	$\downarrow$	$\leq b$	$\mathbf{r}(i)$
> 0	$\uparrow$	$\downarrow$	$> b$	$\mathbf{r}(i) + 1$
≤ 0	$\uparrow$	$\uparrow$	$-$	$\mathbf{r}(i)$
< 0	$\downarrow$	$\uparrow$	$\leq b$	$\mathbf{r}(i)$
< 0	$\downarrow$	$\uparrow$	$> b$	$\mathbf{r}(i) + 1$

Nous voyons bien apparaître la constante b servant à tester si l'alternance entre une phase croissante et décroissante a lieu au dessus ou en dessous de b. De la même façon que pour $TS_{rb}(S)$, nous notons $\rightarrow_b^*$ la fermeture réflexive et transitive de la relation $\rightarrow_b$ et, étant donnée une configuration $(q, \mathbf{m}, \mathbf{v}, \mathbf{r})$ de $TS_b(S)$, nous définissons l'ensemble $\mathbf{Reach}_b(S, (q, \mathbf{m}, \mathbf{v}, \mathbf{r})) = \{(q', \mathbf{m'}, \mathbf{v'}, \mathbf{r'}) \in Q \times \{\downarrow, \uparrow\}^n \times \mathbb{N}^n \times \mathbb{N}^n \mid (q, \mathbf{m}, \mathbf{v}, \mathbf{r},) \rightarrow_b^* (q', \mathbf{m'}, \mathbf{v'}, \mathbf{r'})\}$. De plus, pour une configuration $(q, \mathbf{v})$ de $TS(S)$, nous notons $\mathbf{Reach}_b(S, (q, \mathbf{v}))$ l'ensemble égal à $\mathbf{Reach}_b(S, (q, \uparrow, \mathbf{v}, \mathbf{0}))$ où $\uparrow$ représente le vecteur avec toutes ses composantes égales à $\uparrow$. Ceci nous permet de définir une nouvelle notion de machines à compteurs *reversal*-bornées.

Definition 2.9 (Machine à compteurs *reversal*-bornée) *Soit $b, k \in \mathbb{N}$. Une machines à n compteurs (S, c_0) est k-reversal-b bornée si pour tout $(q, \boldsymbol{m}, \boldsymbol{v}, \boldsymbol{r}) \in \mathbf{Reach}_b(S, c_0)$ et pour tout $i \in [1..n]$, nous avons $\boldsymbol{r}(i) \leq k$.*

Nous définissons ensuite les variantes suivantes :

1. Une machine à compteurs est dite *reversal*-bornée si il existe $k, b \in \mathbb{N}$ tels qu'elle est k-reversal-b-bornée.
2. Pour $k \in \mathbb{N}$, une machine à compteurs est dite k-*reversal*-bornée, si il existe $b \in \mathbb{N}$ tel qu'elle est k-*reversal*-b-bornée.
3. Pour $b \in \mathbb{N}$, une machine à compteurs est dite *reversal*-b-bornée, si il existe $k \in \mathbb{N}$ tel qu'elle est k-*reversal*-b-bornée.

Nous remarquons que cette définition implique la proposition suivante :

Proposition 2.10 *Une machine à compteurs est Ibarra-reversal-bornée si et seulement si elle est reversal-0-bornée.*

Pour finir, notons également que comme nous l'avons indiqué précédemment cette notion permet d'englober toutes les machines à compteurs qui ont un ensemble fini de configurations accessibles.

2.2.2 Calculer l'ensemble d'accessibilité

Nous montrons dans cette section que la généralisation que nous proposons permet de conserver les propriétés sur l'ensemble d'accessibilité d'une machine à compteurs *reversal*-bornée, à savoir qu'il est semi-linéaire. Nous présentons ici la construction permettant d'aboutir à ce résultat. L'idée principale consiste à construire, étant donnée une machine *reversal*-bornée, une machine *reversal*-0-bornée à partir de laquelle en utilisant le résultat du théorème 2.3 on peut construire l'ensemble d'accessibilité de la première machine.

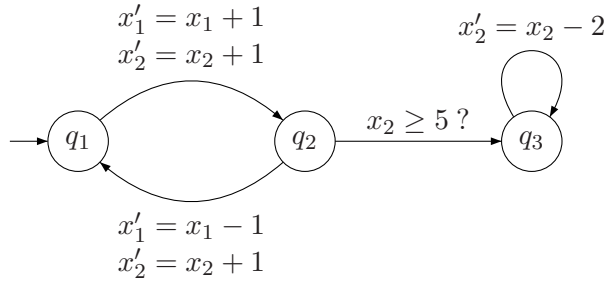


FIGURE 2.4 – Une machine $(S_1, (q_1, (0, 0))$ 1-reversal-1-bornée

Soient $k, b \in \mathbb{N}$. Dans cette section, nous considérons une machine à n compteurs $S = \langle Q, X, E \rangle$ munie d'une configuration initiale $c_0 = (q_0, \mathbf{v}_0)$ dans $Q \times \mathbb{N}^n$ telles que (S, c_0) est k-*reversal*-b-bornée. Nous construisons maintenant la machine à compteurs $S' = \langle Q', X, E' \rangle$. Avant de définir Q' et E', nous introduisons deux symboles $\perp$ et ω_b qui ne sont pas des entiers. Intuitivement ω_b nous servira à symboliser une valeur de compteurs strictement plus grande que b et $\perp$ sera utilisé lorsque nous ne saurons pas si une valeur de compteurs est plus petite ou non que b. Nous posons alors $Q' = Q \times B^n$ avec $B = [0..b] \uplus \{\omega_b, \perp\}$. La machine à compteurs S' encode les exécutions de la machine à compteurs S de la façon suivante :

1. lorsque la valeur d'un compteur dans l'exécution de $TS(S)$ est inférieure ou égale à b, alors dans $TS(S')$ elle est stockée dans l'état de contrôle et la valeur du compteur correspondant est 0,
2. lorsque la valeur d'un compteur passe strictement au-dessus de b dans $TS(S)$ alors la valeur du compteur correspondant dans $TS(S')$ devient égale à cette dernière valeur.

Avant de donner la définition formelle de la relation de transitions E', nous définissons la fonction $+_B$ de $B \times \mathbb{Z}$ dans B, satisfaisant les règles suivantes, pour tout $d \in \mathbb{Z}$:

– si $d \geq 0$, $\omega_b +_B d = \omega_b$,
– si $d \leq 0$, $\omega_b +_B d = \perp$,
– si $d \geq 0$, pour tout $e \in [0..b]$, si $d + e \leq b$ alors $d +_B e = d + e$ sinon $d +_B e = \omega_b$,
– si $d \leq 0$, pour tout $e \in [0..b]$ tel que $-e \leq d$, $e +_B d = e + d$.

En analysant la définition de ces opérations, nous voyons que lorsqu'une valeur passe strictement au dessus de b, nous la remplaçons par ω_b et si nous faisons décroître ω_b, nous introduisons le symbole $\perp$, car nous ne savons pas si la valeur obtenue à ce moment est inférieure ou égale à b ou strictement plus grande que b. Ces opérations peuvent être étendues composante par composante aux vecteurs. Nous notons B_0 l'ensemble $B \setminus \{\perp\}$. Un élément de B_0^n sera donc un vecteur de B^n n'ayant aucune composante égale à $\perp$. De même, nous notons $B_\perp^n$ l'ensemble $B^n \setminus B_0^n$ des vecteurs de B^n ayant

au moins une composante égale à $\perp$. Nous définissons maintenant la relation de transitions $E' \subseteq (Q \times B^n) \times T_n \times (Q \times B^n)$ de la machine à n compteurs S'. E' est la plus petite relation de $(Q \times B^n) \times T_n \times (Q \times B^n)$ satisfaisant les règles suivantes :

1. pour chaque $(q, (\#, \mu, \delta), q') \in E$, pour chaque $\mathbf{u} \in B_0^n$ tel que pour tout $i \in [1..n]$, si $\mathbf{u}(i) \in [0..b]$ alors $\mu(i)\#(i)\mathbf{u}(i)$ (ie $\mathbf{u}$ satisfait la garde de la transition pour les composantes différentes de ω_b), il existe $(\#', \mu', \delta') \in T_n$ et $\mathbf{u}' \in B^n$ tels que $((q, \mathbf{u}), (\#', \mu', \delta'), (q', \mathbf{u}')) \in E'$ et tels que :
   (a) $\mathbf{u}' = \mathbf{u} +_B \delta$,
   (b) pour tout $i \in [1..n]$,
      i. si $\mathbf{u}(i) \in [0..b]$ et $\mathbf{u}'(i) \in [0..b]$, alors $(\#'(i), \mu'(i), \delta'(i)) = (\leq, 0, 0)$ (sur cette composante, la transition n'a aucun effet et laisse le i-ème compteur à 0),
      ii. si $\mathbf{u}(i) \in [0..b]$ et $\mathbf{u}'(i) = \omega_b$, alors $(\#'(i), \mu'(i), \delta'(i)) = (\leq, 0, \mathbf{u}(i) + \delta(i))$ (cette transition remet la nouvelle valeur obtenue pour $\mathbf{u}$ dans le i-ème compteur, cela arrive lorsque la composante de $\mathbf{u}$ passe au dessus de b),
      iii. si $\mathbf{u}(i) = \omega_b$ alors $(\#'(i), \mu'(i), \delta'(i)) = (\#(i), \mu(i), \delta(i))$ (si la i-ème composante de $\mathbf{u}$ est déjà au-dessus de b alors la transition se comporte comme la transition de E sur les compteurs).

2. pour chaque $q \in Q$, pour chaque $\mathbf{u} \in B_\perp^n$ et pour chaque $\mathbf{u}' \in B_0^n$ tel que pour tout $i \in [1..n]$, si $\mathbf{u}(i) \neq \perp$ alors $\mathbf{u}'(i) = \mathbf{u}(i)$ (ie $\mathbf{u}$ et $\mathbf{u}'$ correspondent sur les composantes pour lesquelles $\mathbf{u}$ est différent de $\perp$), il existe $(\#', \mu', \delta') \in T_n$ tel que $((q, \mathbf{u}), (\#', \mu', \delta'), (q, \mathbf{u}')) \in E'$ et tels que, pour tout $i \in [1..n]$:
   (a) si $\mathbf{u}(i) \neq \perp$, alors $(\#'(i), \mu'(i), \delta'(i)) = (\leq, 0, 0)$ (si la i-ème composante de $\mathbf{u}$ est différente de $\perp$, on ne fait rien),
   (b) si $\mathbf{u}(i) = \perp$ alors :
      i. si $\mathbf{u}'(i) = \omega_b$, alors $\#'(i) \in \{\leq\}$, $\mu'(i) = b + 1$ et $\delta'(i) = 0$ (soit la valeur du i-ème compteur est strictement plus grande que b, et on ne fait rien sur le compteur et on a ω_b à la composante correspondante dans l'état de contrôle d'arrivée)
      ii. si $\mathbf{u}'(i) \in [0..b]$, alors $(\#'(i), \mu'(i), \delta'(i)) = (=, \mathbf{u}'(i), -\mathbf{u}'(i))$ (soit la valeur du i-ème compteur est inférieure ou égale à b et on remet ce compteur à 0 et sa valeur est enregistrée dans l'état de contrôle d'arrivée).

Lorsqu'un état de contrôle avec le symbole $\perp$ est atteint, nous testons donc la valeur du compteur correspondant pour savoir si cette valeur est comprise entre 0 et b ou si elle est strictement plus grande que b.

Exemple 2.11 *La machine à compteurs représentée à la figure 2.5 avec la configuration initiale* $((q_1, 0, 0), (0, 0))$ *encode la machine à compteurs de la figure 2.4 munie de la configuration initiale* $(q_1, (0, 0))$. *Notons juste que par souci de clarté nous n'avons pas construit entièrement la machine* S_1' *encodant* S_1 *mais seulement les états de contrôle accessibles à partir de l'état* $(q_1, 0, 0)$.

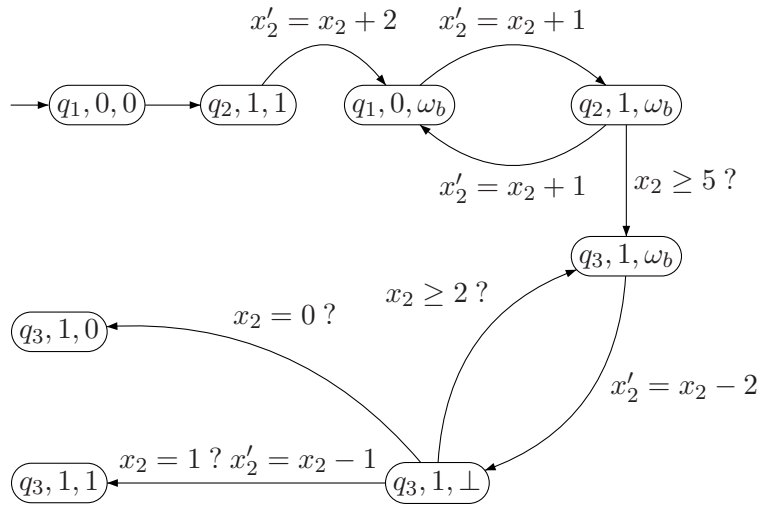


FIGURE 2.5 – Une machine à compteurs S_1' 1-*reversal*-0-bornée pour encoder S_1

Nous allons maintenant énoncer quelques lemmes récapitulant les propriétés de la machine S'. Nous définissons la relation $\sim_b \subseteq (Q \times \mathbb{N}^n) \times (Q \times B_0^n \times \mathbb{N}^n)$ permettant de faire le lien entre les configurations de $TS(S)$ et celles de $TS(S')$. Pour tout $(q, \mathbf{v}) \in Q \times \mathbb{N}^n$ et pour tout $(q', \mathbf{u}, \mathbf{v}') \in Q \times B_0^n \times \mathbb{N}^n$, nous avons $(q, \mathbf{v}) \sim_b (q', \mathbf{u}, \mathbf{v}')$ si et seulement si :

- $q = q'$, et, pour tout $i \in [1..n]$,
- si $\mathbf{u}(i) \in [0..b]$, alors $\mathbf{v}(i) = \mathbf{u}(i)$ et $\mathbf{v}'(i) = 0$, et,
- si $\mathbf{u}(i) = \omega_b$, alors $\mathbf{v}'(i) = \mathbf{v}(i)$ et $\mathbf{v}(i) > b$.

Le système de transitions associé à S est noté $TS(S) = \langle Q \times \mathbb{N}^n, E, \rightarrow \rangle$ et celui associé à S' est noté $TS(S') = \langle Q' \times \mathbb{N}^n, E', \Rightarrow \rangle$. Par construction de la machine à compteurs S', nous montrons que la machine à compteurs S' simule la machine S :

Lemme 2.12 *Soient $(q_1, \boldsymbol{v}_1) \in Q \times \mathbb{N}^n$ et $(q_1, \boldsymbol{u}_1, \boldsymbol{v}_1') \in Q \times B_0^n \times \mathbb{N}^n$ tels que $(q_1, \boldsymbol{v}_1) \sim_b (q_1, \boldsymbol{u}_1, \boldsymbol{v}_1')$. Pour tout $(q_2, \boldsymbol{v}_2) \in Q \times \mathbb{N}^n$, nous avons $(q_1, \boldsymbol{v}_1) \rightarrow (q_2, \boldsymbol{v}_2)$ si et seulement si il existe $(\boldsymbol{u}_2, \boldsymbol{v}_2') \in B_0^n \times \mathbb{N}^n$ tel que $(q_2, \boldsymbol{v}_2) \sim_b (q_2, \boldsymbol{u}_2, \boldsymbol{v}_2')$ et tel que :*

- *soit $(q_1, \boldsymbol{u}_1, \boldsymbol{v}_1') \Rightarrow (q_2, \boldsymbol{u}_2, \boldsymbol{v}_2')$,*
- *soit il existe $(\boldsymbol{u}_3, \boldsymbol{v}_3') \in Q \times B_\bot^n \times \mathbb{N}^n$ tel que $(q_1, \boldsymbol{u}_1, \boldsymbol{v}_1') \Rightarrow (q_2, \boldsymbol{u}_3, \boldsymbol{v}_3') \Rightarrow (q_2, \boldsymbol{u}_2, \boldsymbol{v}_2')$.*

Preuve : Soient $(q_1, \mathbf{v}_1) \in Q \times \mathbb{N}^n$ et $(q_1, \mathbf{u}_1, \mathbf{v}_1') \in Q \times B_0^n \times \mathbb{N}^n$ tels que $(q_1, \mathbf{v}_1) \sim_b (q_1, \mathbf{u}_1, \mathbf{v}_1')$. La preuve se fait par une étude de cas en utilisant la définition de E' et $\sim_b$.

Soit $(q_2, \mathbf{v}_2) \in Q \times \mathbb{N}^n$ tel que $(q_1, \mathbf{v}_1) \xrightarrow{e} (q_2, \mathbf{v}_2)$ avec $e = (q, (\#, \mu, \delta), q')$. Remarquons que, pour tout $i \in [1..n]$, nous avons nécessairement $\mu(i)\#(i)\mathbf{u}_1(i)$ car $\mu\#\mathbf{v}_1$ et $(q_1, \mathbf{v}_1) \sim_b (q_1, \mathbf{u}_1, \mathbf{v}_1')$. Nous posons $\mathbf{u}_3 = \mathbf{u}_1 +_B \delta$. Alors par définition de E' (cas 1.), il existe dans E' une transision $e' = ((q, \mathbf{u}_1), (\#', \mu', \delta'), (q', \mathbf{u}_3))$ vériririfiant les conditions 1.(b).(i), 1.(b)(ii), 1.(b).(iii). Montrons

qu'il existe $\mathbf{v}'_3$ tel que $(q_1, \mathbf{u}_1, \mathbf{v}'_1) \overset{e'}{\Rightarrow} (q_2, \mathbf{u}_3, \mathbf{v}'_3)$. Soit $i \in [1..n]$. Si $\mathbf{u}_1(i) \in [0..b]$, alors $\mathbf{v}'_1(i) = 0$ (par définition de $\sim_b$) et $\#'(i) \in \{\leq\}$ et $\mu'(i) = 0$ (par définition de e'), par conséquent $\mu(i)\#(i)\mathbf{v}'_1(i)$. Si $\mathbf{u}_1 = \omega_b$, alors $\mathbf{v}'_1(i) = \mathbf{v}_1(i)$ et $\#'(i) = \#(i)$ et $\mu'(i) = \mu(i)$, comme $\mu(i)\#(i)\mathbf{v}_1(i)$ et on a par conséquent $\mu'(i)\#'(i)\mathbf{v}'_1(i)$. Nous en déduisons que $\mu'\#'\mathbf{v}'_1$ et ainsi qu'il existe $\mathbf{v}'_3$ tel que $(q_1, \mathbf{u}_1, \mathbf{v}'_1) \overset{e'}{\Rightarrow} (q_2, \mathbf{u}_3, \mathbf{v}'_3)$. Se pose de nouveau deux cas, soit $\mathbf{u}_3 \in B_0^n$, soit $\mathbf{u}_3 \in B_\perp^n$.

1. Si $\mathbf{u}_3 \in B_0^n$, nous montrons que $(q_2, \mathbf{v}_2) \sim_b (q_2, \mathbf{u}_3, \mathbf{v}'_3)$. Soit $i \in [1..n]$.
   - Si $\mathbf{u}_3(i) \in [0..b]$, il faut montrer que $\mathbf{u}_3(i) = \mathbf{v}_2(i)$ et $\mathbf{v}'_3(i) = 0$. Par définition de e', $\mathbf{u}_1(i) \in [0..b]$ et $\mathbf{u}_3(i) = \mathbf{u}_1(i) + \delta(i)$. En utilisant la définition de $\sim_b$, on a $\mathbf{u}_3(i) = \mathbf{v}_1(i) + \delta(i) = \mathbf{v}_2(i)$. De plus, on est dans le cas 1.(b).(i) et comme $\mathbf{v}'_1(i) = 0$, on en déduit que $\mathbf{v}'_3(i) = 0$.
   - Si $\mathbf{u}_3(i) = \omega_b$, il faut montrer que $\mathbf{v}'_3(i) = \mathbf{v}_2(i)$ et que $\mathbf{v}_2(i) > b$. Nous avons de plus soit $\mathbf{u}_1(i) \in [0..b]$, soit $\mathbf{u}_1(i) = \omega_b$
     - Supposons que $\mathbf{u}_1(i) \in [0..b]$. On est alors dans le cas 1.(b).(ii), d'où $\delta'(i) = \mathbf{u}_1(i) + \delta(i)$. Par conséquent, $\mathbf{v}'_3(i) = \mathbf{v}'_1(i) + \delta'(i) = 0 + \mathbf{u}_1(i) + \delta(i) = \mathbf{v}_1(i) + \delta(i) = \mathbf{v}_2(i)$. De plus, comme $\mathbf{u}_3(i) = \omega_b = \mathbf{u}_1(i) +_B \delta(i)$, on a $\mathbf{u}_1(i) + \delta(i) > b$, d'où $\mathbf{v}_2(i) > b$.
     - Supposons que $\mathbf{u}_1(i) = \omega_b$. On est alors dans le cas 1.(b).(iii), d'où $\mathbf{v}'_3(i) = \mathbf{v}'_1(i) + \delta'(i) = \mathbf{v}_1(i) + \delta(i) = \mathbf{v}_2(i)$. Dans ce cas, on a aussi $\mathbf{v}_1(i) > b$ (par définition de $\sim_b$), de plus $\delta(i) > 0$, car $\omega_b +_B \delta(i) = \omega_b$, donc $\mathbf{v}_2(i) > b$.

2. Si $\mathbf{u}_3 \in B_\perp^n$. Par définition de E' (cas 2.), il existe $e'' = ((q_2, \mathbf{u}_3), (\#'', \mu'', \delta'')(q_2, \mathbf{u}_2))$ avec $\mathbf{u}_2 \in B_0^n$ tel que pour tout $j \in [1..n]$ vérifiant $\mathbf{u}_3(j) = \perp$, si $\mathbf{v}'_3(j) \in [0..b]$, alors $\#''(j) \in \{=\}$, $\mu''(j) = \mathbf{u}_2(j) = \mathbf{v}'_3(j)$ et $\delta''(i) = -\mathbf{u}_2(j)$ et si $b < \mathbf{v}'_3(j)$, alors $\mathbf{u}_2(j) = \omega_b$, $\#''(j) \in \{\leq\}$, $\mu''(j) = b+1$ et $\delta''(j) = 0$. De plus pour tout $j \in [1..n]$ tel que $\mathbf{u}_3(j) \neq \perp$, on a $\mathbf{u}_2(j) = \mathbf{u}_3(j)$, $\#''(j) \in \{\leq\}$ et $\mu''(j) = \delta''(j) = 0$. Il est alors évident que $\mu''\#''\mathbf{v}'_3$ et par conséquent il existe $\mathbf{v}'_2$ tel que $(q_2, \mathbf{u}_3, \mathbf{v}'_3) \overset{e''}{\Rightarrow} (q_2, \mathbf{u}_2, \mathbf{v}'_2)$. Nous montrons que $(q_2, \mathbf{v}_2) \sim_b (q_2, \mathbf{u}_2, \mathbf{v}'_2)$. Soit $i \in [1..n]$.
   - Si $\mathbf{u}_2(i) \in [0..b]$, il faut montrer que $\mathbf{u}_2(i) = \mathbf{v}_2(i)$ et $\mathbf{v}'_2(i) = 0$. Remarquons que par définition de e'', $\mathbf{u}_3(i) \in [0...b]$ ou $\mathbf{u}_3(i) = \perp$.
     - Si $\mathbf{u}_3(i) = \perp$. Par définition de e', nécessairement $\mathbf{u}_1(i) = \omega_b$. Donc, par définition de $\sim_b$, $\mathbf{v}'_1(i) = \mathbf{v}_1(i)$. Alors toujours en utilisant e' (cas 1.(b).(iii)), $\mathbf{v}'_3(i) = \mathbf{v}'_1(i) + \delta'(i) = \mathbf{v}_1(i) + \delta(i) = \mathbf{v}_2(i)$. Par définition de e'' (cas 2.(b).(ii)), $\mathbf{v}'_2(i) = \mathbf{v}'_3(i) - \mathbf{u}_2(i)$, de plus comme $\mathbf{u}_2(i) \in [0..b]$, nécessairement $\mathbf{u}_2(i) = \mathbf{v}'_3(i)$. D'où, $\mathbf{v}'_2(i) = 0$ et $\mathbf{u}_2(i) = \mathbf{v}_2(i)$.
     - Si $\mathbf{u}_3(i) \in [0..b]$. Par définition de e', nécessairement $\mathbf{u}_1(i) \in [0..b]$. Donc, par définition de $\sim_b$, $\mathbf{v}'_1(i) = 0$ et $\mathbf{u}_1(i) = \mathbf{v}_1(i)$. Nous sommes dans le cas 1.(b).(i). On en déduit que $\mathbf{v}'_3(i) = 0$ et $\mathbf{u}_3(i) = \mathbf{u}_1(i) + \delta(i) = \mathbf{v}_1(i) + \delta(i) = \mathbf{v}_2(i)$. En ce qui concerne e'', nous sommes au cas 2.(a), par conséquent $\mathbf{v}'_2(i) = \mathbf{v}'_3(i) = 0$ et $\mathbf{u}_2(i) = \mathbf{u}_3(i) = \mathbf{v}_2(i)$.
   - Si $\mathbf{u}_2(i) = \omega_b$, il faut montrer que $\mathbf{v}'_2(i) = \mathbf{v}_2(i)$ et que $\mathbf{v}_2(i) > b$. Remarquons que par définition de e'', $\mathbf{u}_3(i) = \omega_b$ ou $\mathbf{u}_3(i) = \perp$.
     - Si $\mathbf{u}_3(i) = \perp$. Par définition de e', nécessairement $\mathbf{u}_1(i) = \omega_b$. Donc, par définition de $\sim_b$, $\mathbf{v}'_1(i) = \mathbf{v}_1(i)$. Alors toujours en utilisant e' (cas 1.(b).(iii)), $\mathbf{v}'_3(i) = \mathbf{v}'_1(i) + \delta'(i) = \mathbf{v}_1(i) + \delta(i) = \mathbf{v}_2(i)$. Par définition de e'' (cas 2.(b).(i)), $\mathbf{v}'_2(i) = \mathbf{v}'_3(i)$ et $b + 1 \leq \mathbf{v}'_3(i)$, d'où $\mathbf{v}'_2(i) = \mathbf{v}_2(i)$ et $\mathbf{v}_2(i) > b$.
     - Si $\mathbf{u}_3(i) = \omega_b$. Alors par définition de e', on a de nouveau deux choix, soit $\mathbf{u}_1(i) \in [0..b]$, soit $\mathbf{u}_1(i) = \omega_b$.

- Si $\mathbf{u}_1(i) \in [0..b]$. Alors $\mathbf{v}'_1(i) = 0$ et $\mathbf{v}_1(i) = \mathbf{u}_1(i)$. Alors par définition de e' (cas 1.(b).(ii)), $\mathbf{v}'_3(i) = \mathbf{v}'_1(i) + \delta'(i) = 0 + \mathbf{u}_1(i) + \delta(i) = \mathbf{v}_2(i)$. De plus, par définition de e'' (cas 2.(a)), $\mathbf{v}'_2(i) = \mathbf{v}'_3(i) = \mathbf{v}_2(i)$. Comme $\mathbf{u}_1(i) +_B \delta(i) = \omega_b$, nécessairement $\mathbf{u}_1(i) +_B \delta(i) > b$, donc $\mathbf{v}_2(i) > b$.
- Si $\mathbf{u}_1(i) = \omega_b$. Alors $\mathbf{v}'_1(i) = \mathbf{v}_i(1)$. Alors par définition de e' (cas 1.(b).(iii)), $\mathbf{v}'_3(i) = \mathbf{v}'_1(i)+\delta'(i) = \mathbf{v}_1(i)+\delta(i) = \mathbf{v}_2(i)$. Par définition de e'' (cas $2.(c).(i)$), $\mathbf{v}'_2(i) = \mathbf{v}'_3(i) = \mathbf{v}_2(i)$. De plus, comme $\omega_b +_B \delta(i) = \mathbf{u}_3(i) = \omega_b$, on en déduit que $\delta(i) > 0$ et comme $\mathbf{u}_1(i) > b$ (par définition de $\sim_b$), on a bien $\mathbf{v}_2(i) > b$.

L'implication dans l'autre sens se prouve exactement de la même façon en utilisant les propriétés de E' et de $\sim_b$. □

À partir de la configuration initiale $(q_0, \mathbf{v}_0)$ de $TS(S)$, nous construisons une configuration initiale $c'_0 \in Q \times B_0^n \times \mathbb{N}^n$ de $TS(S')$ avec $c'_0 = (q_0, \mathbf{u}_0, \mathbf{v}'_0)$ et pour tout $i \in [1..n]$:

- si $\mathbf{v}_0(i) \leq b$, alors $\mathbf{u}_0(i) = \mathbf{v}_0(i)$ et $\mathbf{v}'_0(i) = 0$, et,
- si $\mathbf{v}_0(i) > b$, alors $\mathbf{u}_0(i) = \omega_b$ et $\mathbf{v}'_0(i) = \mathbf{v}_0(i)$.

On a alors $c_0 \sim_b c'_0$ de façon évidente. En utilisant le lemme précédent et les caractéristiques de S', on peut déduire le lemme suivant faisant le lien entre l'ensemble d'accessibilité de (S, c_0) et celui de (S', c'_0).

Lemme 2.13 *Nous avons* **Reach**$(S, c_0) = \{c \in Q \times \mathbb{N}^n \mid \exists c' \in$ **Reach**(S', c'_0) *tel que* $c \sim_b c'\}$.

Preuve : Soit $c \in$ **Reach**(S, c_0). Montrons qu'il existe $c' \in$ **Reach**(S', c'_0) tel que $c \sim_b c'$. Comme $c \in$ **Reach**(S, c_0), il existe une exécution dans $TS(S)$ de la forme $c_0 \to c_1 \dots c_f$ avec $f \in \mathbb{N}$ et $c_f = c$. Nous raisonnons par induction sur la taille de cette exécution, c'est-à-dire sur f, en utilisant le lemme 2.12. Si $f = 0$, nous avons directement que $c'_0 \in$ **Reach**(S', c'_0) et $c_0 \sim_b c'_0$. Supposons que $f > 0$ et que pour tout $j \in [0..f-1]$, il existe $c'_j \in$ **Reach**(S', c'_0) tel que $c_j \sim_b c'_j$. On a donc en particulier $c_{f-1} \sim_b c'_{f-1}$. Comme de plus $c_{f-1} \to c_f$, d'après le lemme 2.12 il existe une configuration c'_f de $TS(S')$ telle que $c'_{f-1} \Rightarrow^* c'_f$ et telle que $c_f \sim_b c'_f$. Comme $c'_{f-1} \Rightarrow^* c'_f$ et que $c'_{f-1} \in$ **Reach**(S', c'_0), on a bien $c'_f \in$ **Reach**(S', c'_0).

Soit $c \in Q \times \mathbb{N}^n$ et supposons qu'il existe $c' \in$ **Reach**(S', c'_0) tel que $c \sim_b c'$. Montrons que $c \in$ **Reach**(S, c_0). Comme $c' \in$ **Reach**(S', c'_0), il existe une exécution dans $TS(S')$ de la forme $c'_0 \Rightarrow c'_1 \dots c'_f$ avec $c'_f = c'$. Nous raisonnons par induction sur la taille de cette exécution, c'est-à-dire sur f, en utilisant le lemme 2.12 et la forme de la machine à compteurs S'. Si $f = 0$, alors $c = c_0$ et $c_0 \in$ **Reach**(S, c_0). Soit $f \in \mathbb{N}^*$, supposons maintenant que pour tout $j \in [0..f-1]$, si $c'_j \in Q \times B_0^n \times \mathbb{N}^n$, alors il existe $c_j \in$ **Reach**(S, c_0) tel que $c_j \sim_b c'_j$. Si $c'_{f-1} \in Q \times B_0^n \times \mathbb{N}^n$ alors par hypothèse d'induction $c_{f-1} \sim_b c'_{f-1}$ et par le lemme 2.12, nous avons $c_{f-1} \to c$ et comme $c_{f-1} \in$ **Reach**(S, c_0), on en déduit que $c \in$ **Reach**(S, c_0). Supposons maintenant que $c'_{f-1} \in Q \times B_\perp^n \times \mathbb{N}^n$, alors par construction de E', nous avons nécessairement $c'_{f-2} \in Q \times B_0^n \times \mathbb{N}^n$ et par hypothèse d'induction $c_{f-2} \sim_b c'_{f-2}$, comme de plus $c'_{f-2} \Rightarrow c'_{f-1} \Rightarrow c'_f$, alors en utilisant le lemme 2.12, on obtient $c_{f-1} \to c$ et comme $c_{f-1} \in$ **Reach**(s, c_0), on conclut que $c \in$ **Reach**(S, c_0). □

Ce lemme qui caractérise l'ensemble d'accessibilité **Reach**(S, c_0) en fonction de **Reach**(S', c'_0) nous apporte une autre information capitale pour pouvoir montrer le résultat voulu sur l'ensemble d'accessibilité des machines à compteurs *reversal*-bornées. En effet, la propriété qui suit est vérifiée :

Lemme 2.14 *Si* **Reach**(S', c'_0) *est effectivement définissable dans Presburger alors* **Reach**(S, c_0) *l'est aussi.*

Preuve : D'après le lemme 2.13, on a **Reach**$(S, c_0) = \{c \in Q \times \mathbb{N}^n \mid \exists c' \in$ **Reach**(S', c'_0) tel que $c \sim_b c'\}$. Supposons qu'il existe une formule de Presburger $\phi' \in$ **Presb**(X'_0) telle que **Reach**$(S', c'_0) = [\![\phi']\!]_{X'_0}$. Nous construisons alors la formule suivante $\phi \in$ **Presb**(X_0), tel que :

$$\phi = \exists x'_0 \dots \exists x'_n . \phi' \wedge \bigvee_{(q,\mathbf{u}) \in Q \times B_0^n} \big(x'_0 = (q, \mathbf{u}) \wedge x_0 = q \bigwedge_{i \in [1..n]} (x'_i = 0 \wedge x_i = \mathbf{u}(i)) \vee (x'_i > b \wedge x_i = x'_i)\big)$$

Par définition de $\sim_b$, nous en déduisons que **Reach**$(S, c_0) = [\![\phi]\!]_{X_0}$.

□

Pour pouvoir utiliser l'ensemble d'accessibilité de (S', c'_0) afin de déduire celui de (S, c_0), encore faut-il montrer que **Reach**(S', c'_0) est effectivement définissable dans Presburger. Nous allons maintenant nous attacher à démontrer que comme la machine à compteurs (S, c_0) est k-*reversal*-b-bornée, la machine à compteurs (S', c'_0) est k-*reversal*-0-bornée. Ceci est dû au fait que, grâce au codage utilisé, chaque compteur x_i réalise dans (S', c'_0) le même nombre d'alternances que celles réalisées strictement au-dessus de b par x_i dans (S, c_0).

Nous définissons une nouvelle relation $\approx_b \subseteq (Q \times \{\downarrow, \uparrow\}^n \times \mathbb{N}^n \times \mathbb{N}^n) \times (Q \times B_0^n \times \{\downarrow, \uparrow\}^n \times \mathbb{N}^n \times \mathbb{N}^n)$ entre les configurations du système de transitions $TS_b(S)$ (où l'on prend en compte les alternances qui ont lieu au dessus de b) et celles de $TS_0(S')$ (où l'on prend en compte les alternances qui ont lieu au dessus de 0). Nous définissons cette relation de la façon suivante, $(q, \mathbf{m}, \mathbf{v}, \mathbf{r}) \approx_b ((q, \mathbf{u}), \mathbf{m}', \mathbf{v}', \mathbf{r}')$ si et seulement si :

- $(q, \mathbf{v}) \sim_b (q, \mathbf{u}, \mathbf{v}')$, et,
- pour tout $i \in [1..n]$, si $\mathbf{v}(i) > b$ alors $\mathbf{m}(i) = \mathbf{m}'(i)$ et $\mathbf{r}(i) = \mathbf{r}'(i)$.

Nous notons les systèmes de transitions considérés ainsi $TS_b(S) = \langle Q \times \{\downarrow, \uparrow\}^n \times \mathbb{N}^n \times \mathbb{N}^n, E, \rightarrow_b \rangle$ et $TS_0(S') = \langle Q' \times \{\downarrow, \uparrow\}^n \times \mathbb{N}^n \times \mathbb{N}^n, E', \Rightarrow_0 \rangle$.Nous avons alors le pendant du lemme 2.12 en ce qui concerne les systèmes de transitions associés à S et S' dans lesquels on compte le nombre d'alternances entre les phases de croissance et de décroissance.

Lemme 2.15 *Soient* $c_1 \in Q \times \{\downarrow, \uparrow\}^n \times \mathbb{N}^n \times \mathbb{N}^n$ *et* $c'_1 \in Q \times B_0^n \times \{\downarrow, \uparrow\}^n \times \mathbb{N}^n \times \mathbb{N}^n$ *tels que* $c_1 \approx_b c'_1$. *Pour tout* $c_2 \in Q \times \{\downarrow, \uparrow\}^n \times \mathbb{N}^n \times \mathbb{N}^n$, *nous avons* $c_1 \rightarrow_b c_2$ *si et seulement si il existe* c'_2 *tel que* $c_2 \approx_b c'_2$ *et tel que :*

- *soit* $c'_1 \Rightarrow_0 c'_2$,
- *soit il existe* $c'_3 \in Q \times B_\perp^n \times \{\downarrow, \uparrow\}^n \times \mathbb{N}^n \times \mathbb{N}^n$ *tel que* $c'_1 \Rightarrow_0 c'_3 \Rightarrow_0 c'_2$.

Preuve : Supposons que $c_1 \approx_b c'_1$ et soit $c_2 \in Q \times \{\downarrow, \uparrow\}^n \times \mathbb{N}^n \times \mathbb{N}^n$ tel que $c_1 \rightarrow_b c_2$. Posons $c_1 = (q_1, \mathbf{m}_1, \mathbf{v}_1, \mathbf{r}_1)$, $c'_1 = (q_1, \mathbf{u}_1, \mathbf{m}'_1, \mathbf{v}'_1, \mathbf{r}'_1)$ et $c_2 = (q_2, \mathbf{m}_2, \mathbf{v}_2, \mathbf{r}_2)$. Comme $c_1 \approx_b c'_1$, par définition de la relation de transitions $\rightarrow_b$, on a $(q_1, \mathbf{v}_1) \rightarrow (q_2, \mathbf{v}_2)$. De plus comme $c_1 \approx_b c'_1$, par définition de $\approx_b$, on en déduit que $(q_1, \mathbf{v}_1) \sim_b (q_1, \mathbf{v}'_1)$. On utilise alors le lemme 2.12 qui nous permet de déduire qu'il existe $(q_2, \mathbf{u}_2, \mathbf{v}'_2) \in Q \times B_0^n \times \mathbb{N}^n$ tel que $(q_2, \mathbf{v}_2) \sim_b (q_2, \mathbf{u}_2, \mathbf{v}'_2)$ et :

- soit $(q_1, \mathbf{u}_1, \mathbf{v}'_1) \Rightarrow (q_2, \mathbf{u}_2, \mathbf{v}'_2)$,
- soit il existe $(q_3, \mathbf{u}_3, \mathbf{v}'_3) \in Q \times B_\perp^n \times \mathbb{N}^n$ vérifiant $(q_1, \mathbf{u}_1, \mathbf{v}'_1) \Rightarrow (q_3, \mathbf{u}_3, \mathbf{v}'_3) \Rightarrow (q_2, \mathbf{u}_2, \mathbf{v}'_2)$.

Supposons que nous sommes dans le premier cas, c'est-à-dire que $(q_1, \mathbf{u}_1, \mathbf{v}'_1) \Rightarrow (q_2, \mathbf{u}_2, \mathbf{v}'_2)$. Nous construisons alors $\mathbf{m}'_2$ et $\mathbf{r}'_2$ de la façon suivante, pour tout $i \in [1..n]$:

– si $\mathbf{u}_2(i) \in [0..b]$, $\mathbf{m}'_2(i) = \mathbf{m}'_1(i)$ et $\mathbf{r}'_2(i) = \mathbf{r}'_1(i)$,
– si $\mathbf{u}_2(i) = \omega_b$, $\mathbf{m}'_2(i) = \mathbf{m}_2(i)$ et $\mathbf{r}'_2(i) = \mathbf{r}_2(i)$.

En appliquant la définition de $\approx_b$, nous avons bien $(q_2, \mathbf{m}_2, \mathbf{v}_2, \mathbf{r}_2) \approx_b (q_2, \mathbf{u}_2, \mathbf{m}'_2, \mathbf{v}'_2, \mathbf{r}'_2)$. Nous pouvons de plus montrer que $(q_1, \mathbf{m}'_1, \mathbf{v}'_1, \mathbf{r}'_1) \Rightarrow_0 (q_2, \mathbf{m}'_2, \mathbf{v}'_2, \mathbf{r}'_2)$ par une étude de cas en utilisant la façon dont la relation E' est construite et la définition de $\Rightarrow_0$.
Supposons maintenant que nous sommes dans le cas où il existe $(q_3, \mathbf{u}_3, \mathbf{v}'_3) \in Q \times B_\perp^n \times \mathbb{N}^n$ vérifiant $(q_1, \mathbf{u}_1, \mathbf{v}'_1) \Rightarrow (q_3, \mathbf{u}_3, \mathbf{v}'_3) \Rightarrow (q_2, \mathbf{u}_2, \mathbf{v'}_2)$. Nous construisons alors $\mathbf{m}'_3, \mathbf{r}'_3, \mathbf{m}'_2$ et $\mathbf{r}'_2$ de la façon suivante, pour tout $i \in [1..n]$:

– si $\mathbf{u}_3(i) \in [0..b]$, alors $\mathbf{m}'_3(i) = \mathbf{m}'_1(i)$ et $\mathbf{r}'_3(i) = \mathbf{r}'_1(i)$,
– si $\mathbf{u}_3(i) = \omega_b$, $\mathbf{m}'_3(i) = \mathbf{m}_2(i)$ et $\mathbf{r}'_3(i) = \mathbf{r}_2(i)$,
– si $\mathbf{u}_3(i) = \perp$, $\mathbf{m}'_3(i) = \downarrow$ et $\mathbf{r}'_3(i) = \mathbf{r}_2(i)$,
– $\mathbf{m}'_2(i) = \mathbf{m}'_3(i)$ et $\mathbf{r}'_2(i) = \mathbf{r}'_3(i)$.

Montrons que nous avons alors $(q_2, \mathbf{m}_2, \mathbf{v}_2, \mathbf{r}_2) \approx_b (q_2, \mathbf{u}_2, \mathbf{m}'_2, \mathbf{v}'_2, \mathbf{r}'_2)$. Comme $(q_2, \mathbf{v}_2) \sim_b (q_2, \mathbf{u}_2, \mathbf{v}'_2)$, il nous suffit de montrer que pour tout $i \in [1..n]$, si $\mathbf{v}'_2(i) > b$ alors $\mathbf{m}'_2(i) = \mathbf{m}_2(i)$ et $\mathbf{r}'_2(i) = \mathbf{r}_2(i)$. Soit $i \in [1..n]$ tel que $\mathbf{v}'_2(i) > b$. Par définition de $\sim_b$, nous avons nécessairement $\mathbf{u}_2(i) = \omega_b$. De la façon dont la relation E' est construite, nous en déduisons que $\mathbf{u}_3(i) = \omega_b$ ou $\mathbf{u}_3(i) = \perp$. Si $\mathbf{u}_3(i) = \omega_b$, alors $\mathbf{m}'_3(i) = \mathbf{m}_2(i)$ et $\mathbf{r}'_3(i) = \mathbf{r}'_2(i)$ et par définition de E', $\mathbf{v}'_2(i) = \mathbf{v}'_3(i)$, en utilisant la définition de $\Rightarrow_0$, on en déduit que $\mathbf{m}'_2(i) = \mathbf{m}'_3(i) = \mathbf{m}_2(i)$ et $\mathbf{r}'_2(i) = \mathbf{r}'_3(i) = \mathbf{r}_2(i)$. Si $\mathbf{u}_3(i) = \perp$, alors $\mathbf{m}'_3(i) = \downarrow$ et $\mathbf{r}'_3(i) = \mathbf{r}_2(i)$. Par définition de E', on a nécessairement $\mathbf{u}_1(i) = \omega_b$ et par conséquent $\mathbf{u}_1(i) > b$. Toujours par définition de E', on a forcément $\mathbf{u}_2(i) < \mathbf{u}_1(i)$, d'où $\mathbf{m}_2(i) = \downarrow$. De plus comme $\mathbf{m}'_3(i) = \mathbf{m}'_2(i)$, on a bien $\mathbf{m}'_2(i) = \mathbf{m}_2(i)$. De plus, on a $\mathbf{r}'_2(i) = \mathbf{r}'_3(i) = \mathbf{r}_2(i)$. Par une étude de cas utilisant les définitions de E' et de $\Rightarrow_0$, nous pouvons de plus montrer que $(q_1, \mathbf{m}'_1, \mathbf{v}'_1, \mathbf{r}'_1) \Rightarrow_0 (q_3, \mathbf{m}'_3, \mathbf{v}'_3, \mathbf{r}'_3) \Rightarrow_0 (q_2, \mathbf{m}'_2, \mathbf{v}'_2, \mathbf{r}'_2)$.

La preuve de l'implication dans l'autre sens se fait exactement de la même façon. □

Ce lemme nous permet d'établir le lien qui existe entre les ensembles d'accessibilité $\mathbf{Reach}_b(S, c_0)$ et $\mathbf{Reach}_0(S', c'_0)$.

Lemme 2.16 *Nous avons* $\mathbf{Reach}_b(S, c_0) = \{c \in Q \times \{\downarrow, \uparrow\}^n \times \mathbb{N}^n \times \mathbb{N}^n \mid \exists c' \in \mathbf{Reach}_0(S', c'_0)$ *tel que* $c \approx_b c'\}$.

Idée de la preuve : Nous avons $\mathbf{Reach}_b(S, c_0) = \mathbf{Reach}_b(S, (q_0, \uparrow, \mathbf{v}_0, \mathbf{0}))$ et $\mathbf{Reach}_0(S', c'_0) = \mathbf{Reach}_0(S', (q_0, \mathbf{u}_0, \uparrow, \mathbf{v}'_0, \mathbf{0}))$. Remarquons que comme $c_0 \sim_b c'_0$, nous avons bien $(q_0, \uparrow, \mathbf{v}_0, \mathbf{0}) \approx_b (q_0, \mathbf{u}_0, \uparrow, \mathbf{v}'_0, \mathbf{0})$. La preuve se fait alors exactement de la même manière que celle du lemme 2.13, excepté que l'on considère ici la relation $\approx_b$ et le résultat du lemme 2.15. □

De ce dernier lemme, on arrive facilement à la propriété souhaitée pour la machine à compteurs (S', c'_0) sachant que (S, c_0) est k-*reversal*-b-bornée, en effet :

Lemme 2.17 (S', c'_0) *est* k-*reversal*-0 *bornée.*

Preuve : Nous raisonnons par contradiction et supposons que la machine à compteurs (S', c'_0) n'est pas k-*reversal*-0-bornée. Par conséquent, il existe $k' \in \mathbb{N}$, $j \in [1..n]$ et $(q, \mathbf{u}, \mathbf{m'}, \mathbf{v'}, \mathbf{r'}) \in \mathbf{Reach}_0(S', c'_0)$ tel que $k' > k$ et $\mathbf{r}(j) = k'$. Deux cas se posent alors, soit $\mathbf{u} \in B_0^n$ soit $\mathbf{u} \in B_\perp^n$. Supposons que $\mathbf{u} \in B_0^n$. Alors il existe $(q, \mathbf{m}, \mathbf{v}, \mathbf{r}) \in Q \times \{\downarrow, \uparrow\}^n \times \mathbb{N}^n \times \mathbb{N}^n$ tel que $(q, \mathbf{m}, \mathbf{v}, \mathbf{r}) \approx_b (q, \mathbf{u}, \mathbf{m'}, \mathbf{v'}, \mathbf{r'})$. Par le lemme 2.16, on en déduit que $(q, \mathbf{m}, \mathbf{v}, \mathbf{r}) \in \mathbf{Reach}_b(S, c_0)$, et comme par définition de $\approx_b$, $\mathbf{r}(i) = \mathbf{r'}(i) = k'$, (S, c_0) n'est pas k-*reversal*-b-bornée, ce qui constitue une contradiction. Supposons maintenant que $\mathbf{u} \in B_\perp^n$. Par définition de E' et de $\Rightarrow_0$, il existe nécessairement $\mathbf{u'} \in B_0^n$ et $\mathbf{v''} \in \mathbb{N}^n$ tel que $(q, \mathbf{u}, \mathbf{m'}, \mathbf{v'}, \mathbf{r'}) \Rightarrow_0 (q, \mathbf{u}, \mathbf{m'}, \mathbf{v''}, \mathbf{r'})$ et nous reprenons alors le raisonnement du cas précédent. □

Si nous récapitulons, étant donnée une machine à compteurs (S, c_0) k-*reversal*-b-bornée, nous avons construit une machine à compteurs (S', c'_0) vérifiant les deux propriétés suivantes :

– (S', c'_0) est k-*reversal*-0-bornée,
– si $\mathbf{Reach}(S', c'_0)$ est effectivement définissable dans Presburger alors $\mathbf{Reach}(S, c_0)$ l'est aussi.

En utilisant la proposition 2.10 et le résultat d'Ibarra énoncé au théorème 2.3, on en déduit le théorème suivant :

Théorème 2.18 *L'ensemble d'accessibilité d'une machines à compteurs reversal-bornée est effectivement définissable dans Presburger.*

Nous obtenons, comme nous l'avons fait précédemment, le lien avec les problématiques de vérification :

Corollaire 2.19 *Le problème d'accessibilité symbolique est décidable pour les machines à compteurs reversal-bornées.*

2.2.3 Propriétés

Nous nous intéressons maintenant aux autres propriétés des machines à compteurs *reversal*-bornées et nous montrons qu'elles sont aussi préservées par la nouvelle définition que nous proposons.

En ce qui concerne la propriété sur les sous-systèmes décrite par la proposition 2.8, de la même façon que cette proposition était évidente dans le cas des machines à compteurs *Ibarra-reversal*-bornées, elle reste évidente dans notre cas.

Proposition 2.20 *Soit S une machine à compteurs, $S' = \langle Q', X, E \rangle$ une machine à compteurs telle que $S' \leq S$ et $c_0 \in Q' \times \mathbb{N}^X$. Si (S, c_0) est reversal-bornée alors (S', c_0) est reversal-bornée.*

Comme nous l'avons dit cette propriété est particulièrement utile lorsque l'on veut analyser des systèmes de grande taille, on sait ainsi que dans le cas de machines à compteurs *reversal*-bornées, l'analyse de sous-systèmes est également possible.

Nous allons maintenant montrer qu'avec la nouvelle définition proposée, les machines à compteurs *reversal*-bornées sont toujours aplatissables et par conséquent, si l'on donne une telle machine à compteurs en entrée de l'outil FAST , on sait que l'algorithme implanté dans cet outil termine. Nous ajoutons ainsi une nouvelle classe à la classification réalisée dans [LS05] des systèmes à compteurs aplatissables. Nous nous attachons maintenant à prouver cette propriété.

Soit $k, b \in \mathbb{N}$. Nous considérons une machine à n compteurs $S = \langle Q, X, E\rangle$ munie d'une configuration $c_0 = (q_0, \mathbf{v}_0)$ dans $Q \times \mathbb{N}^n$ de telle sorte que (S, c_0) est k-*reversal*-b-bornée. Nous reprenons la machine à n compteurs initialisée k-*reversal*-0-bornée (S', c'_0) construite dans la section 2.2.2 pour montrer que (S, c_0) a un ensemble d'accessibilité effectivement définissable dans Presburger. Nous rappelons que $S' = \langle Q', X, E'\rangle$. D'après le théorème 2.5, la machine à compteurs (S', c'_0) (qui est *Ibarra-reversal*-bornée) est aplatissable. Par conséquent, il existe une machine à compteurs plate $S'_P = \langle Q'_P, X, E'_P\rangle$ munie d'une configuration initiale $c''_0 = (q''_0, \mathbf{v}_0)$ et une fonction de repliage $z' : Q'_P \to Q'$ tels que $\mathbf{Reach}(S', c'_0) = z'(\mathbf{Reach}(S'_P, c''_0))$. À partir de S'_P, nous allons construire un aplatissement S_p de S qui nous servira pour montrer que S est aplatissable. La principale difficulté dans la construction de S_P vient du fait que les actions étiquetant les transitions de S' et par conséquent aussi celles de S'_P ne sont pas exactement celles de S, il nous faut modifier la relation de transition E'_P de façon à incorporer les actions étiquetant les transitions de S tout en gardant la platitude. Nous construisons un système $S_p = \langle Q_p, X, E_p\rangle$ dont nous associons les états de contrôle aux états de contrôle de S'_P qui sont liées par la fonction de repliage z' aux états de S' ne contenant pas le symbole $\perp$.

Ainsi $S_p = \langle Q_p, X, E_p\rangle$ est définie de telle sorte qu'il existe une fonction $f : Q_p \to Q'_p$ vérifiant les propriétés suivantes :

- pour tout $q \in Q_p$, $z'(f(q)) \in Q \times B_0^n$,
- pour tout $q' \in Q'_p$, tel que $z'(q') \in Q \times B_0^n$, il existe un unique $q \in Q_p$ tel que $f(q) = q'$,
- pour tout $q_1, q_2 \in Q_p$, pour toute translation gardée $(\#, \mu, \delta) \in T_n$, $(q_1, (\#, \mu, \delta), q_2) \in E_p$ si et seulement si une des deux conditions suivantes est respectée :

  1. Soit il existe une translation gardée $t_1 \in T_n$ tel que $(f(q_1), t_1, f(q_2)) \in E'_p$ et si on pose $z(f(q_1)) = (q'_1, \mathbf{u}_1)$, $z(f(q_2)) = (q'_2, \mathbf{u}_2)$ et $t_1 = (\#', \mu', \delta')$ alors les propriétés suivantes sont vérifiées :

     (a) $(q'_1, (\#, \mu, \delta), q'_2) \in E$, et,

     (b) pour tout $i \in [1..n]$ tel que $\mathbf{u}_1(i) \in [1..b]$, $\mu(i)\#(i)\mathbf{u}_1(i)$, et,

     (c) $\mathbf{u}_2 = \mathbf{u}_1 +_B \delta$, et,

     (d) pour tout $i \in [1..n]$, si $\mathbf{u}_1(i) = \omega_b$ alors $\#(i) = \#'(i)$ et $\mu(i) = \mu'(i)$ et $\delta(i) = \delta'(i)$.

  2. Soit il existe $q_3 \in Q'_P$ et deux translations gardées $t_1, t_2 \in T_n$ tels que $(f(q_1), t_1, q_3) \in E'_p$ et $(q_3, t_2, f(q_2)) \in E'_p$ et $z'(q_3) \in Q \times B_\perp^n$ et si on pose $z'(f(q_1)) = (q'_1, \mathbf{u}_1)$ et $z'(q_3) = (q'_3, \mathbf{u}_3)$ alors on a les propriétés suivantes :

     (a) $(q'_1, (\#, \mu, \delta), q'_3) \in E$, et,

     (b) pour tout $i \in [1..n]$ tel que $\mathbf{u}_1(i) \in [1..b]$, $\mu(i)\#(i)\mathbf{u}_1(i)$, et,

     (c) $\mathbf{u}_3 = \mathbf{u}_1 +_B \delta$, et,

     (d) pour tout $i \in [1..n]$, si $\mathbf{u}_1(i) = \omega_b$ alors $\#(i) = \#'(i)$ et $\mu(i) = \mu'(i)$ et $\delta(i) = \delta'(i)$.

- Pour tout état $q \in Q_p$, le nombre d'éléments dans $\{(q', t, q) \in E_p\}$ est égal au nombre d'éléments $\{(q', t, f(q)) \in E'_p\}$.

Remarquons que par définition de E' et de E'_p, pour le cas 1. si $(f(q_1), t_1, f(q_2)) \in E'_p$ alors il existe nécessairement une transition dans E vérifiant les conditions 1.(b), 1.(c) et 1.(d). De la même façon,

pour le cas 2. si $(f(q_1), t_1, q_3) \in E'$ et $(q_3, t_2, f(q_2)) \in E'$ alors il existe nécessairement une transition dans E vérifiant les conditions 2.(b), 2.(c) et 2.(d). La dernière condition nous dit que pour chaque état de contrôle dans Q_p, il y a autant de transitions qui arrivent dans cet état que dans l'état de Q'_p auquel il est associé. Ceci nous permet entre autre d'assurer la platitude.

Comme S'_p est un aplatissement de S', par définition de S_p, nous pouvons déduire le lemme qui suit.

Lemme 2.21 *S_p est un aplatissement de S.*

Preuve : Pour prouver que S_p est un aplatissement de S, il faut montrer que S_p est plat et qu'il existe une fonction de repliage $z : Q_p \to Q$. Comme S'_p est plat, nous en déduisons facilement que S_p est plat. En effet, si S_p n'était pas plat, il existerait une état de contrôle $q \in Q_p$ appartenant à deux cycles élémentaires dans S_p, et par construction de S_p, l'état correspondant $f(q)$ de Q'_p appartiendrait également à deux cycles élémentaires dans S'_p ce qui constitue une contradiction. Nous considérons maintenant la fonction $z : Q_p \to Q$ définie de la façon suivante, pour tout $q \in Q_p$ et $q' \in Q$, $z(q) = q'$ si et seulement si $z'(f(q)) = (q', \mathbf{u})$ avec $\mathbf{u} \in B_0^n$. Ainsi par construction de E_p, on a bien pour tout $(q_1, t, q_2) \in E_p$, $(z(q_1), t, z(q_2)) \in E$. On en déduit que S_p est bien un aplatissement de S. □

Nous utilisons maintenant la fonction de repliage $z : Q_p \to Q$ définie dans la preuve précédente. Nous avons alors le lemme suivant concernant les systèmes de transitions $TS(S_P) = \langle Q_p \times \mathbb{N}^n, E_p, \to_p \rangle$ et $TS(S'_p) = \langle Q_p \times \mathbb{N}^n, E'_p, \Rightarrow_p \rangle$. La preuve de ce lemme se fait par étude de cas comme pour la preuve du lemme 2.12 en utilisant les façons dont les machines à compteurs S', S'_p et S_p sont construites.

Lemme 2.22 *Soient $c_1 \in Q_p \times \mathbb{N}^n$ une configuration de $TS(S_p)$ et $c'_1 \in Q'_p \times \mathbb{N}^n$ une configuration de $TS(S'_p)$ telles que $z(c_1) \sim_b z'(c'_1)$. Pour tout $c_2 \in Q_p \times \mathbb{N}^n$, nous avons $c_1 \to_p c_2$ si et seulement si il existe $c'_2 \in Q'_p \times \mathbb{N}^n$ tel que $z(c_2) \sim_b z'(c'_2)$ et tel que :*

- *soit $c'_1 \Rightarrow_p c'_2$,*
- *soit il existe c'_3 tel que $z'(c'_3) \in Q \times B_\perp^n \times \mathbb{N}^n$ tel que $c'_1 \Rightarrow_p c'_3 \Rightarrow_p c'_2$*

Pour pouvoir conclure que (S, c_0) est aplatissable, il nous reste à montrer le lemme qui suit où $q'''_0 \in Q_p$ est l'état tel que $z(q'''_0) = q''_0$ et $c'''_0 = (q'''_0, \mathbf{v}_0)$. Notons que par définition de f et z' que cet état $q'''_0 \in Q_p$ existe. Nous avons alors :

Lemme 2.23 $z(\mathbf{Reach}(S_p, c'''_0)) = \mathbf{Reach}(S, c_0)$.

Preuve : Comme S_p est un aplatissement de S, nous avons $z(\mathbf{Reach}(S_p, c'''_0)) \subseteq \mathbf{Reach}(S, c_0)$. Nous nous attachons maintenant à montrer que $\mathbf{Reach}(S, c_0) \subseteq z(\mathbf{Reach}(S_p, c'''_0))$. D'après le lemme 2.13, nous avons $\mathbf{Reach}(S, c_0) = \{c \in Q \times \mathbb{N}^n | \exists c' \in \mathbf{Reach}(S', c'_0)$ tel que $c \sim_b c'\}$. Comme $\mathbf{Reach}(S', c'_0) = z'(\mathbf{Reach}(S'_p, c''_0))$:

$$\mathbf{Reach}(S, c_0) = \{c \in Q \times \mathbb{N}^n | \exists c' \in \mathbf{Reach}(S'_p, c''_0) \text{ tel que } c \sim_b z'(c')\}$$

Soit $c \in \mathbf{Reach}(S, c_0)$. Alors il existe $c' \in \mathbf{Reach}(S'_p, c''_0)$ tel que $c \sim_b z'(c')$. On pose $c' = (q', \mathbf{v})$, comme $c \sim_b z'(c')$, alors par définition de $\sim_b$, on a $z'(q') \in Q \times B_0^n$. Donc il existe $q \in Q_p$ tel que $f(q) = q'$. Si on définit la configuration $c'' = (q, \mathbf{v})$, on a par ailleurs $z(c'') = c$ par définition de z et de plus $z(c'') \sim_b z'(c')$. Il nous reste plus qu'à montrer que $c'' \in \mathbf{Reach}(S_p, c'''_0)$. Or, en utilisant le lemme 2.22, de la même façon que pour le lemme 2.13, nous avons $\mathbf{Reach}(S_p, c'''_0) = \{c \in Q_p \times \mathbb{N}^n \mid$

$\exists c' \in \mathbf{Reach}(S'_p, c''_0)$ tel que $z(c) \sim_b z'(c')\}$. Comme $c' \in \mathbf{Reach}(S'_p, c''_0)$ et comme $z(c'') \sim_b z'(c')$, on a donc bien $c'' \in \mathbf{Reach}(S_p, c'''_0)$. □

À partir de la machine à compteurs initialisée (S, c_0), nous avons donc construit un aplatissement S_p de S avec une fonction de repliage z et une configuration initiale c'''_0 de $TS(S_p)$ vérifiant $z(c'''_0) = c_0$ et $z(\mathbf{Reach}(S_p, c'''_0)) = \mathbf{Reach}(S, c_0)$. La seule hypothèse que nous avons utilisée pour réaliser cette construction est que la machine à compteurs (S, c_0) est *reversal*-bornée. Nous pouvons ainsi conclure par le théorème suivant :

Théorème 2.24 *Les machines à compteurs reversal-bornées sont aplatissables.*

Nous pouvons donc utiliser l'outil FAST pour analyser ces systèmes en étant sûr que l'algorithme utilisé terminera.

2.3 Décider si une machine à compteurs est *reversal*-bornée

Nous allons maintenant nous intéresser à la décidabilité de la notion de *reversal*-bornée, c'est-à-dire savoir si étant donnée une machine à compteurs, on peut décider si elle est *reversal*-bornée. Nous étudions ici différentes variantes de ce problème selon les paramètres donnés pour la notion de *reversal*-bornée.

2.3.1 Indécidabilité dans le cas général

Dans [Iba78], Ibarra montre qu'il n'est pas possible de décider si une machine à compteurs initialisée est *reversal*-0-bornée. En utilisant une preuve similaire, nous prouvons le résultat suivant :

Théorème 2.25 *Vérifier si une machine à compteurs est reversal-bornée est un problème indécidable.*

Preuve : Nous réduisons le problème de l'arrêt pour les machines de Minksy déterministes à 2 compteurs initialisées qui est indécidable (cf. théorème 1.37). Soit $S = \langle Q, X, E\rangle$ une machine de Minsky déterministe à deux compteurs munie d'une configuration intiale $c_0 = (q_0, (a_0, a_1))$. À partir de S, nous construisons une machine à compteurs $S' = \langle Q', X', E'\rangle$ travaillant sur trois compteurs x_1, x_2 et x_3 de la façon suivante :

- $Q' = Q \cup \{q'_e, q''_e \mid e \in E\}$,
- $E' = \{(q, t_1, q'_e), (q'_e, t_2, q''_e), (q''_e, t', q') \mid e \in E$ et $e = (q, t, q')\}$.

où t_1 est une translation gardée qui modifie uniquement la valeur du compteur x_3 avec l'action $x'_3 = x_3 + 2$, t_2 est une translation gardée qui modifie uniquement la valeur du compteur x_3 avec l'action $x'_3 = x_3 - 1$ et t' ne modifie pas la valeur du compteur x_3 et modifie les valeurs des compteurs x_1 et x_2 comme t. Remarquons que la machine à compteurs S' est aussi déterministe. De plus remarquons que la machine à compteurs S' munie de la configuration $(q_0, (a_0, a_1, 0))$ est *reversal*-bornée si et seulement si son unique exécution est finie. En effet, si cette exécution est infinie, le compteur x_3 alterne entre une phase croissante et décroissante toujours au-dessus d'une nouvelle valeur. Comme par construction, S' partant de l'entrée $(q_0, (a_0, a_1, 0))$ a une exécution finie si et seulement si la machine à compteurs initialisée (S, c_0) s'arrête, nous en déduisons le résultat d'indécidabilité énoncé. □

2.3.2 Cas où un des deux paramètres est fixé

Nous regardons maintenant ce qui se passe lorsque l'un des deux paramètres est donné, à savoir si l'on peut décider étant donné $k \in \mathbb{N}$ (respectivement $b \in \mathbb{N}$) si une machine à compteurs initialisée est k-*reversal*-bornée (respectivement *reversal*-b-bornée). Nous obtenons malheureusement encore une fois des résultats d'indécidabilité. Notons que dans [Iba78], Ibarra montre que savoir si une machine à compteurs est *reversal*-0-borné est un problème indécidable. Nous généralisons ici ce résultat :

Théorème 2.26 *Étant donné $b \in \mathbb{N}$, savoir si une machine à compteurs est reversal-b-bornée est un problème indécidable.*

Preuve : Pour tout $b \in \mathbb{N}$, nous pouvons réutiliser la preuve du théorème 2.25. En effet, la machine à compteurs S' construite dans cette preuve munie de la configuration initiale $(q_0, (a_0, a_1, 0))$ est *reversal*-b-bornée si et seulement si elle s'arrête. Or cette machine s'arrête si et seulement si la machine de Minsky à deux compteurs S munie de la configuration initiale c_0 s'arrête. □

Nous avons le même résultat d'indécidabilité si c'est le paramètre comptant le nombre d'alternances qui est fixé plutôt que la borne, en effet :

Théorème 2.27 *Étant donné $k \in \mathbb{N}$, savoir si une machine à compteurs est k-reversal-bornée est un problème indécidable.*

Preuve : Pour montrer ce résultat, nous utilisons encore une fois la preuve du théorème 2.25 et la machine à 3 compteurs $S' = \langle Q', X', E' \rangle$ munie de la configuration initiale $(q_0, (a_0, a_1, 0))$. Soit $k \in \mathbb{N}$, nous construisons alors une autre machine à 3 compteurs $S_k = \langle Q_k, X', E_k \rangle$ de la façon suivante :

- $Q_k = Q' \cup \{q_i, q'_i \mid i \in [1..k]\}$,
- $E_k = E \cup \{(q_i, \textbf{inc}(x_3), q_i), (q_i, \textbf{inc}(x_3), q'_i), (q'_i, \textbf{dec}(x_3), q'_i) \mid i \in [1..k]\} \cup \{(q'_i, \textbf{ifzero}(x_3), q_{i+1}) \mid i \in [1..k-1]\} \cup \{(q'_k, \textbf{ifzero}(x_3), q_0)\}$.

Nous assurons ainsi que pour chaque borne $b \in \mathbb{N}$, il existe une exécution de la machine à compteurs S_k partant de la configuration $(q_1, (a_0, a_1, 0))$ pour laquelle le compteur x_3 réalise k alternances au dessus de b entre une phase croissante et une phase décroissante avant d'arriver dans l'état q_0. De plus, pour tout $b \in \mathbb{N}$ il n'existe pas d'exécution réalisant plus de k alternances au dessus de b entre une phase croissante et une phase décroissante avant d'arriver dans l'état q_0. On en déduit que l'unique exécution de la machine à compteurs S' partant de la configuration $(q_0, (a_0, a_1, 0))$ est finie si et seulement si $(S_k, (q_1, (a_0, a_1, 0)))$ est k-*reversal*-bornée. Il suffit de prendre comme borne b, la valeur maximale prise par tout compteur dans cette dernière exécution. Comme l'unique exécution de la machine à compteurs S' partant de la configuration $(q_0, (a_0, a_1, 0))$ est finie si et seulement si la machine de Minsky à 2 compteurs initialisée (S, c_0) s'arrête, on obtient le résultat du théorème. □

2.3.3 Cas où les deux paramètres sont fixés

Dans [Iba78], Ibarra montre que étant donné un entier $k \in \mathbb{N}$, le problème de savoir si une machine à compteurs initialisée est k-*reversal*-0-bornée est indécidable. Cependant, comme nous l'avons déjà mentionné, dans son article Ibarra considère des machines à compteurs munies d'états acceptants et il prend en compte uniquement les exécutions terminant dans ces états acceptants. Le fait que nous acceptons toutes les exécutions d'une machine à compteurs change ce résultat d'indécidabilité. Nous

montrons en effet qu'étant donnés deux entiers $k, b \in \mathbb{N}$, on peut décider si une machine à compteurs est k-*reversal*-b-bornée.

Nous considérons une machine à n compteurs $S = \langle Q, X, E\rangle$ munie d'une configuration initiale $c_0 = (q_0, \mathbf{v}_0)$ dans $Q \times \mathbb{N}^n$. Soient $k, b \in \mathbb{N}$. Nous construisons dans cette section une machine à compteurs $S_{k,b} = \langle Q_{k,b}, X, E_{k,b}\rangle$ qui va encoder les exécutions de S au cours desquelles chaque compteur effectue au plus $k+1$ alternances au-dessus de b entre les phases croissantes et les phases décroissantes. Pour réaliser cela, nous encodons dans les états de contrôle de $S_{k,b}$ la phase, croissante ($\uparrow$) ou décroissante ($\downarrow$), dans laquelle se trouve chaque compteur et le nombre d'alternances réalisées au dessus de b pour arriver à cet état de contrôle. L'idée principal de la machine à compteurs $S_{k,b}$ munie d'une configuration adéquate c'_0 calculée à partir de c_0 est que cette machine va être $k+1$-*reversal*-b-bornée, et qu'elle atteindra un état spécial q_{err} si et seulement si (S, c_0) n'est pas k-*reversal*-b-bornée. Nous formalisons maintenant cette idée.

Les états de contrôle de la machine à compteurs $S_{k,b} = \langle Q_{k,b}, X, E_{k,b}\rangle$ sont construits de la façon suivante $Q_{k,b} = Q_k \uplus Q_b \uplus \{q_{err}\}$ avec :

- $Q_k = Q \times \{\uparrow, \downarrow\}^n \times [0..k]^n$ représentent les états dans lesquels on stocke l'état de contrôle de S, la phase dans laquelle se trouve chaque compteur ainsi que le nombre d'alternances réalisées,
- $Q_b = Q_k \times \{b_{\leq}, b_{>}\}^n$ sont des états intermédiaires dont on se sert pour tester si une valeur de compteurs est plus petite (noté $b_{\leq}$) ou strictement plus grande (noté $b_{>}$) que l'entier b,
- q_{err} est un état spécial qui ne sera accessible à partir d'une configuration donnée c_0 que si (S, c_0) n'est pas k-*reversal*-b-bornée.

En ce qui concerne la relation de transitions $E_{k,b}$, elle est quant à elle définie par ces règles :

- $E_{k,b} \subseteq (Q_k \times Q_b) \cup (Q_b \times Q_k) \cup (Q_b \times \{q_{err}\})$,
- pour tout $(q', \mathbf{m}', \mathbf{r}') \in Q_k$, $(q, \mathbf{m}, \mathbf{r}, \mathbf{u}) \in Q_b$ et toute translation gardée $(\#, \mu, \delta) \in T_n$:

  1. $((q, \mathbf{m}, \mathbf{r}, \mathbf{u}), (\#, \mu, \delta), (q', \mathbf{m}', \mathbf{r}')) \in E_{k,b}$ si et seulement si :

     (a) $(q, (\#, \mu, \delta), q') \in E$, et,

     (b) pour tout $i \in [1..n]$ les conditions exprimées dans le tableau suivant sont vérifiées :

$\delta(i)$	$\mathbf{m}(i)$	$\mathbf{u}(i)$	$\mathbf{m}'(i)$	$\mathbf{r}(i)$	$\mathbf{r}'(i)$
≤ 0	$\downarrow$	$-$	$\downarrow$	$-$	$\mathbf{r}(i)$
< 0	$\uparrow$	$b_{\leq}$	$\downarrow$	$-$	$\mathbf{r}(i)$
< 0	$\uparrow$	$b_{>}$	$\downarrow$	$< k$	$\mathbf{r}(i)+1$
≥ 0	$\uparrow$	$-$	$\uparrow$	$-$	$\mathbf{r}(i)$
> 0	$\downarrow$	$b_{\leq}$	$\uparrow$	$-$	$\mathbf{r}(i)$
> 0	$\downarrow$	$b_{>}$	$\uparrow$	$< k$	$\mathbf{r}(i)+1$

  2. $((q', \mathbf{m}', \mathbf{r}'), (\#, \mu, \delta), (q, \mathbf{m}, \mathbf{r}, \mathbf{u})) \in E_{k,b}$ si et seulement si :

     (a) $\delta = \mathbf{0}$, et,

     (b) $(q', \mathbf{m}', \mathbf{r}') = (q, \mathbf{m}, \mathbf{r})$, et pour tout $i \in [1..n]$,

     (c) si $\mathbf{u}(i) = b_{\leq}$ alors $\#(i) \in \{\geq\}$ et $\mu(i) = b$,et,

     (d) si $\mathbf{u}(i) = b_{>}$ alors $\#(i) \in \{\leq\}$ et $\mu(i) = b+1$.

3. $((q, \mathbf{m}, \mathbf{r}, \mathbf{u}), (\#, \mu, \delta), q_{err}) \in E_{k,b}$ si et seulement si il existe $q' \in Q$ tel que $(q, (\#, \mu, \delta), q') \in E$ et il existe $i \in [1..n]$ tel que :
   - $\mathbf{m}(i) = \downarrow$ et $\delta(i) > 0$ et $\mathbf{u}(i) = b_>$ et $\mathbf{r}(i) = k$, ou,
   - $\mathbf{m}(i) = \uparrow$ et $\delta(i) < 0$ et $\mathbf{u}(i) = b_>$ et $\mathbf{r}(i) = k$.

Notons que pour le point 2.(c), nous ne sommes pas autorisés normalement à avoir le symbole $\geq$ dans le vecteur μ, mais en fait ceci est un raccourci mis pour représenter $b + 1$ transitions réalisant un test d'égalité avec chaque constante comprise entre 0 et b.
Nous munissons ensuite la machine à compteurs $S_{k,b}$ d'une configuration initiale $c'_0 \in Q_{k,b} \times \mathbb{N}^n$ obtenue à partir de la configuration initiale $c_0 = (q_0, \mathbf{v}_0)$ de $TS(S)$ de telle façon que $c'_0 = (q'_0, \mathbf{v}_0)$ avec $q'_0 = (q_0, \uparrow, \mathbf{0})$ où $\uparrow$ désigne ici le vecteur de $\{\uparrow, \downarrow\}^n$ ayant toutes ses composantes égales à $\uparrow$. Nous supposons également que $TS(S_{k,b}) = \langle Q_{k,b} \times \mathbb{N}^n, E_{k,b}, \Rightarrow \rangle$ et que $TS_b(S) = \langle Q \times \mathbb{N}^n, E, \rightarrow_b \rangle$. Par construction de $S_{k,b}$ et par définition du système de transitions associé $TS_b(S)$, nous obtenons alors le lemme suivant :

Lemme 2.28 *Pour tout $(q, \boldsymbol{m}, \boldsymbol{r}, \boldsymbol{u}, \boldsymbol{v}) \in Q \times \{\uparrow, \downarrow\}^n \times [0..k]^n \times \{b_\leq, b_>\}^n \times \mathbb{N}^n$, nous avons $(q, \boldsymbol{m}, \boldsymbol{r}, \boldsymbol{u}, \boldsymbol{v}) \in$ **Reach**$(S_{k,b}, c'_0)$ si et seulement si les conditions suivantes sont vérifiées :*

1. $(q, \boldsymbol{m}, \boldsymbol{v}, \boldsymbol{r}) \in$ **Reach**$_b(S, c_0)$*, et,*
2. *pour tout $i \in [1..n]$:*
   - $\boldsymbol{u}(i) = b_>$ *si et seulement si* $\boldsymbol{v}(i) > b$*, et,*
   - $\boldsymbol{u}(i) = b_\leq$ *si et seulement si* $\boldsymbol{v}(i) \leq b$*.*

Preuve : Supposons qu'il existe $c' = (q, \mathbf{m}, \mathbf{r}, \mathbf{u}, \mathbf{v})$ dans **Reach**$(S_{k,b}, c'_0)$. Alors il existe une exécution $c'_0 \stackrel{e'_0}{\Rightarrow} c'_1 \dots c'_f$ avec $c' = c'_f$. Pour montrer que les conditions sont vérifiées, nous raisonnons par induction sur la taille de cette exécution. Si $f = 0$, nous avons $c'_0 \stackrel{e'_0}{\Rightarrow} c'$ avec $c'_0 = ((q, \uparrow, \mathbf{0}), \mathbf{v}_0)$. Par définition de $E_{k,b}$, nous sommes alors dans le cas (2), nous avons $q = q_0$, $\mathbf{m} = \uparrow$, $\mathbf{v} = \mathbf{v}_0$ et $\mathbf{r} = \mathbf{0}$. Or par définition de **Reach**$_b(S, c_0)$, nous avons bien $(q, \mathbf{m} \uparrow, \mathbf{v}, \mathbf{0}) \in$ **Reach**$_b(S, c_0)$. De plus pour tout $i \in [1..n]$, nous avons bien $\mathbf{u}(i) = b_>$ si et seulement si $\mathbf{v}(i) > b$, et $\mathbf{u}(i) = b_\leq$ si et seulement si $\mathbf{v}(i) \leq b$, car par définition e'_0 teste chaque valeur de compteur pour savoir si elle est au-dessus de b ou en dessous. Nous supposons maintenant que pour tout $j \in [1..f]$ si $c'_j \in Q \times \{\uparrow, \downarrow\}^n \times [0..k]^n \times \{b_\leq, b_>\}^n \times \mathbb{N}^n$ alors les conditions du lemme sont vérifiées. Par définition de $E_{k,b}$, nous avons nécessairement $c'_{f-1} \in Q \times \{\uparrow, \downarrow\}^n \times [0..k]^n \times \{b_\leq, b_>\}^n \times \mathbb{N}^n$ et $c'_f \in Q \times \{\uparrow, \downarrow\}^n \times [0..k]^n \times \mathbb{N}^n$. On pose $c_{f-1} = (q_{f-1}, \mathbf{m}_{f-1}, \mathbf{r}_{f-1}, \mathbf{u}_{f-1}, \mathbf{v}_{f-1})$ et $c_f = (q_f, \mathbf{m}_f, \mathbf{r}_f, \mathbf{v}_f)$. Par définition de $E_{k,b}$ (cas 1.) et de $\rightarrow_b$, nous avons nécessairement $(q_{f-1}, \mathbf{m}_{f-1}, \mathbf{v}_{f-1}, \mathbf{r}_{f-1}) \rightarrow_b (q_f, \mathbf{m}_f, \mathbf{v}_f, \mathbf{r}_f)$ dans $TS_b(S)$ (il suffit en effet de constater que les conditions établies dans le tableau du cas 1. de la définition de $E_{k,b}$ sont les mêmes que celles figurant dans la définition de $\rightarrow_b$). De plus par hypothèse de récurrence, nous avons $(q_{f-1}, \mathbf{m}_{f-1}, \mathbf{v}_{f-1}, \mathbf{r}_{f-1}) \in$ **Reach**$_b(S, c_0)$, d'où $(q_f, \mathbf{m}_f, \mathbf{v}_f, \mathbf{r}_f) \in$ **Reach**$_b(S, c_0)$. Par définition de $E_{k,b}$, nous avons de plus nécessairement $(q, \mathbf{m}, \mathbf{r}) = (q_f, \mathbf{m}_f, \mathbf{r}_f)$ et $\mathbf{v} = \mathbf{v}_f$, et comme la transition e'_f ne fait que tester les valeur de $\mathbf{v}_f$ pour les faire correspondre avec $\mathbf{u}$, nous en déduisons que la condition 2. du lemme est aussi vérifiée par $\mathbf{v}$ et $\mathbf{u}$.
L'implication dans l'autre sens se prouve de la même façon en utilisant la définition de $E_{k,b}$ et de $\rightarrow_b$. □

À partir de ce dernier lemme et compte tenu de la façon dont l'état de contrôle q_{err} est connecté dans $E_{k,b}$ nous en déduisons le lemme suivant :

Lemme 2.29 *(S, c_0) est k-reversal-b-bornée si et seulement si il n'existe pas de $\mathbf{v} \in \mathbb{N}^n$ tel que $(q_{err}, \mathbf{v}) \in$ **Reach**$(S_{k,b}, c'_0)$.*

Preuve : Supposons qu'il existe $\mathbf{v}' \in \mathbb{N}^n$ tel que $(q_{err}, \mathbf{v}') \in$ **Reach**$(S_{k,b}, c'_0)$. Alors par définition de $E_{k,b}$ (cas 3.) il existe $c = (q, \mathbf{m}, \mathbf{r}, \mathbf{u}, \mathbf{v})$ dans **Reach**$(S_{k,b}, c'_0)$ et $e = ((q, \mathbf{m}, \mathbf{r}, \mathbf{u}), (\#, \mu, \delta), q_{err})$ dans $E_{k,b}$ tels que $c \stackrel{e}{\Rightarrow} (q_{err}, \mathbf{v}')$ et il existe $j \in [1..n]$ tels que $\mathbf{m}(j) = \downarrow$ et $\delta(j) > 0$ et $\mathbf{u}(j) = b_>$ et $\mathbf{r}(j) = k$, ou, $\mathbf{m}(j) = \uparrow$ et $\delta(j) < 0$ et $\mathbf{u}(j) = b_>$ et $\mathbf{r}(j) = k$. Alors par le lemme 2.28, nous déduisons que $(q, \mathbf{m}, \mathbf{v}, \mathbf{r}) \in$ **Reach**$_b(S, c_0)$. Par définition de $E_{k,b}$ nous avons qu'il existe $q' \in Q$ tel que $(q, (\#, \mu, \delta), q') \in E$. Nous obtenons alors qu'il existe $\mathbf{m}' \in \{\uparrow, \downarrow\}$ et $\mathbf{v}', \mathbf{r}' \in \mathbb{N}^n$ tels que $(q, \mathbf{m}, \mathbf{v}, \mathbf{r}) \rightarrow_b (q', \mathbf{m}', \mathbf{v}', \mathbf{r}')$ et que de plus $\mathbf{r}'(j) = k + 1$. Comme $(q', \mathbf{m}', \mathbf{v}', \mathbf{r}')$ appartient aussi à **Reach**$_b(S, c_0)$, nous concluons que la machine à compteurs initialisée (S, c_0) n'est pas k-*reversal*-b-bornée.
La preuve de l'implication dans l'autre sens se fait alors de la même façon en utilisant le lemme 2.28 et la définition de $E_{k,b}$. □

Pour pouvoir conclure, il nous reste à montrer que l'on peut décider si l'état de contrôle q_{err} est accessible dans $(S_{k,b}, c_0)$ ou non. Pour cela, nous utilisons le lemme énoncé ci-après :

Lemme 2.30 *La machine à compteurs $(S_{k,b}, c'_0)$ est $(k+1)$-reversal-b-bornée.*

Preuve : Cette propriété est immédiate par la façon dont nous construisons la machine à compteurs $S_{k,b}$, nous encodons déjà dans chaque état la phase dans laquelle se trouve chaque compteur et le nombre d'alternances entre chaque phase réalisées au dessus de b pour atteindre cet état, et ceci pour tous les états atteints dans des exécutions réalisant moins de k alternances. Toujours par construction, nous remarquons qu'une exécution peut faire plus de k alternances au-dessus de b entre les phases de croissance et de décroissance, elle peut en faire en fait exactement $k + 1$, mais elle atteint alors l'état q_{err} à partir duquel aucune transition ne part, c'est pour cela que $(S_{k,b}, c'_0)$ est effectivement $k + 1$-*reversal*-b-bornée. □

Par le lemme 2.29, nous savons que (S, c_0) est k-*reversal*-b-bornée si et seulement si l'état de contrôle q_{err} n'est pas accessible dans $(S_{k,b}, c'_0)$ et comme par le lemme 2.30, nous savons que $(S_{k,b}, c'_0)$ est $(k + 1)$-*reversal*-b-bornée, le corollaire 2.19 nous permet alors d'énoncer le résultat que nous annoncions au début de cette section à savoir :

Théorème 2.31 *Étant donnés deux entiers $k, b \in \mathbb{N}$, savoir si une machine à compteurs est k-reversal-b-bornée est un problème décidable.*

Exemple 2.32 *La machine à compteurs $S_{1,1}$ construite à la figure 2.7 munie de la configuration initiale $((q_1, \uparrow, 0), 0)$ permet de décider si la machine à compteurs S de la figure 2.6 munie de la configuration initiale $(q_1, 0)$ est* 1-*reversal*-1-*bornée. On s'aperçoit que ce n'est pas le cas car à partir de la configuration $((q_1, \uparrow, 0), 0)$ la machine à compteurs $S_{1,1}$ peut atteindre l'état de contrôle q_{err}.*

Notons de plus que la méthode proposée pour prouver le théorème 2.31 fournit également un semi-algorithme (c'est-à-dire un algorithme qui ne termine pas toujours) pour tester si une machine à compteurs est *reversal*-bornée. En effet, il suffit d'énumérer de façon exhaustive les paires $(k, b) \in \mathbb{N}^2$

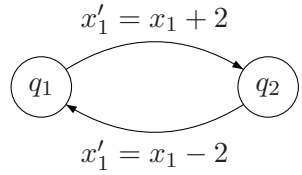


FIGURE 2.6 – Une machine à compteurs S

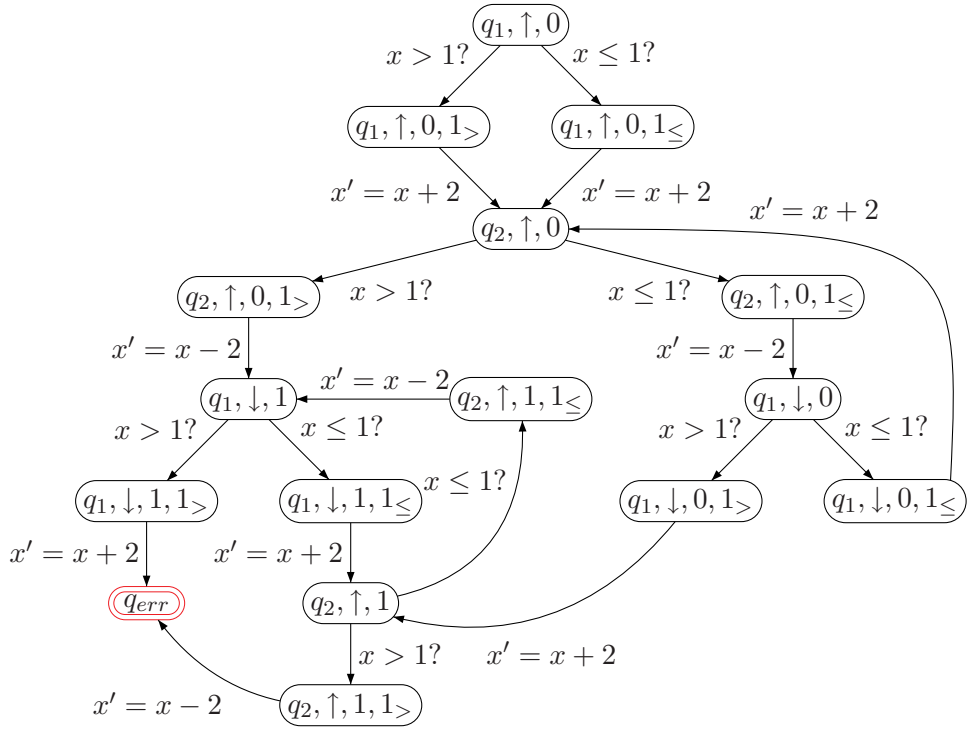


FIGURE 2.7 – La machine à compteurs $S_{1,1}$ pour décider S est 1-*reversal*-1-bornée

et tester si la machine à compteurs initialisée est k-*reversal*-b-bornée. Si la machine est effectivement *reversal*-bornée, on finira forcément par trouver une paire qui fonctionne, en revanche si elle n'est pas *reversal*-bornée, ce semi-algorithme ne terminera jamais. De plus, chaque machine à compteurs $(S_{k,b}, c'_0)$ ainsi construite fournit des informations sur l'ensemble d'accessibilité de la machine à compteurs (S, c_0) et permet de déduire une sous-approximation de son ensemble d'accessibilité en utilisant le résultat énoncé dans le lemme 2.28.

2.3.4 Calculer les k et b pour lesquels un système est k-*reversal*-b-borné

Lorsqu'une machine à compteurs est *reversal*-bornée, il peut-être utile de pouvoir caractériser les paires d'entiers $(k, b) \in \mathbb{N}^2$ pour lesquels cette machine est k-*reversal*-b-bornée, tout d'abord parce que cela donne des informations sur le comportement de la machine à compteurs considérée mais aussi car ces paramètres sont impliqués dans la façon dont l'ensemble d'accessibilité est calculée comme on peut le voir dans la section 2.2.2 et dans [Iba78].

Étant donnée une machine à compteurs (S, c_0), nous définissons l'ensemble suivant pour décrire les paramètres impliquées dans la définition des machines *reversal*-bornées :

$$\mathbf{RB}(S, c_0) = \{(k, b) \in \mathbb{N}^2 \mid (S, c_0) \text{ est } k\text{-reversal-}b\text{-bornée}\}$$

Par définition $\mathbf{RB}(S, c_0) \neq \emptyset$ si et seulement si (S, c_0) est *reversal*-bornée, par conséquent d'après le théorème 2.25 le problème du vide pour $\mathbf{RB}(S, c_0)$ est indécidable en général. En revanche, cet ensemble est récursif, car étant donnée une paire $(k, b) \in \mathbb{N}^2$, on peut décider si cette paire appartient à $\mathbf{RB}(S, c_0)$ par le théorème 2.31. Nous avons de plus la propriété suivante qui est vérifiée :

Lemme 2.33 *Soit (S, c_0) une machine à compteurs initialisée. Si il existe $(k, b) \in \mathbf{RB}(S, c_0)$ et $(k', b') \in \mathbb{N}^2$ tel que $(k, b) \leq (k', b')$ alors $(k', b') \in \mathbf{RB}(S, c_0)$.*

Preuve : Soit (S, c_0) une machine à n compteurs initialisée. Supposons qu'il existe (k, b) dans $\mathbf{RB}(S, c_0)$ et $(k', b') \in \mathbb{N}^2$ tel que $(k, b) \leq (k', b')$. Alors (S, c_0) est k-*reversal*-b-bornée. Par conséquent pour tout $(q, \mathbf{m}, \mathbf{v}, \mathbf{r}) \in \mathbf{Reach}_b(S, c_0)$ et pour tout $i \in [1..n]$, $\mathbf{r}(i) \leq k$. Comme $k \leq k'$, on en déduit que (S, c_0) est k'-*reversal*-b-bornée. De plus, comme $b \leq b'$, par définition de $\mathbf{Reach}_b(S, c_0)$ et de $\mathbf{Reach}_{b'}(S, c_0)$, on a $(q, \mathbf{m}, \mathbf{v}, \mathbf{r}') \in \mathbf{Reach}_{b'}(S, c_0)$ si et seulement si il existe $(q, \mathbf{m}, \mathbf{v}, \mathbf{r}) \in \mathbf{Reach}_b(S, c_0)$ et pour tout $i \in [1..n]$, $\mathbf{r}'(i) \leq \mathbf{r}(i)$. Si (S, c_0) n'était pas k'-*reversal*-b'-bornée, il existerait $(q, \mathbf{m}, \mathbf{v}, \mathbf{r}') \in \mathbf{Reach}_{b'}(S, c_0)$ et $j \in [1..n]$ tel que $k' < \mathbf{r}'(j)$, du coup par la remarque précédente il existerait $\mathbf{r} \in \mathbb{N}^n$ tel que $(q, \mathbf{m}, \mathbf{v}, \mathbf{r}) \in \mathbf{Reach}_b(S, c_0)$ et $k' < \mathbf{r}'(j) \leq \mathbf{r}(j)$, par conséquent (S, c_0) ne serait pas k'-*reversal*-b-bornée, ce qui constitue une contradiction. On en déduit que $(k', b') \in \mathbf{Reach}(S, c_0)$. □

Grâce à ce dernier lemme, nous pouvons déduire que l'ensemble $\mathbf{RB}(S, c_0)$ est fermé par le haut. Nous nous attachons maintenant à montrer que l'on peut calculer les éléments minimaux de cet ensemble dans le cas où (S, c_0) est une machine à compteurs *reversal*-bornée. Dans un premier temps, nous donnons ce lemme intermédiaire qui nous servira par la suite.

Lemme 2.34 *Étant donné $k \in \mathbb{N}$, savoir si une machine à compteurs reversal-bornée est k-reversal-bornée est un problème décidable.*

Preuve : Soit (S, c_0) une machine à n compteurs *reversal*-bornée. Il est donc possible de trouver une paire (k_0, b_0) tel que (S, c_0) est k_0-*reversal*-b_0-bornée, en énumérant des paires de façon exhaustive et en utilisant le résultat du théorème 2.31.
Soit maintenant $k \in \mathbb{N}$. Alors plusieurs cas se posent. D'abord si $k_0 \leq k$ alors forcément (S, c_0) est k-*reversal*-b_0-bornée. Supposons maintenant que $k < k_0$. On considère la machine à compteurs initialisée (S_{k_0,b_0}, c'_0) construite dans la section 2.3.3 pour montrer que l'on peut décider si une machine à compteurs est k'-*reversal*-b'-bornée lorsque les entiers k' et b' sont donnés. Nous avons vu par le lemme 2.30 que cette machine à compteurs initialisée est *reversal*-bornée, par conséquent son ensemble d'accessibilité $\mathbf{Reach}(S_{k_0,b_0}, c'_0)$ est effectivement définissable dans Presburger. Nous transformons alors S_{k_0,b_0} dans une machine S' à $n.(k_0 + 1)$ compteurs avec les mêmes états de contrôle mais en ajoutant des actions sur les transitions. En fait, on a $S' = \langle Q_{k_0,b_0}, X', E' \rangle$ avec $X' = \{x_1, \ldots, x_n\} \cup \{y_{i,1}, \ldots, y_{i,k_0} \mid i \in [1..n]\}$. Les n premiers compteurs $x_1, \ldots, x_n$ gardent le même comportement que les n compteurs de S_{k_0,b_0}, et pour chaque compteurs x_i , les compteurs $y_{i,j}$ avec $j \in [1..k_0]$ servent à mémoriser la valeur du compteur x_i au moment où le nombre d'alternances pour ce compteur est passé de $i - 1$ à i. Nous ne donnons pas le détail de E', mais à partir de S_{k_0,b_0} il est facile de construire une machine à compteurs tels que les compteurs ont le comportement décrit car on connaît pour chaque compteur le moment exact où son nombre d'alternances change (ce nombre d'alternances étant stocké dans les états de contrôle). Ainsi le compteur $y_{i,j}$ a le même comportement que le compteur x_i jusqu'à la j-ème alternance au dessus de b après laquelle sa valeur n'est plus modifiée. Cette machine à compteurs S' munie de la configuration initiale c''_0, obtenue à partir de c_0 en mettant les valeurs des compteurs $y_{i,j}$ égales à celles des compteurs x_i, est encore *reversal*-bornée, car les compteurs que nous avons rajoutés ne font pas plus d'alternances au-dessus de b_0 en partant de c''_0 que les compteurs de S_{k_0,b_0} en partant de c_0. Pour tout $i \in [1..n]$ et tout $l \in [0..k_0]$, nous nous définissons l'ensemble $R(i, l) = \{((q, \mathbf{m}, \mathbf{r}), \mathbf{v}) \in \mathbf{Reach}(S', c''_0) \mid \mathbf{r}(i) = l\}$ des configurations accessibles dans (S', c''_0) pour lesquels le compteur x_i a réalisé l alternances au dessus de b_0. En effet, par le lemme 2.28, nous savons que dans S_{k_0,b_0} le nombre d'alternances réalisées par chaque compteur pour arriver à une configuration $((q, \mathbf{m}, \mathbf{r}), \mathbf{v})$ est donné par le vecteur $\mathbf{r}$, par conséquent c'est aussi le cas dans S'. Remarquons que comme $\mathbf{Reach}(S', c''_0)$ est effectivement définissable dans Presburger, pour tout $i \in [1..n]$ et tout $l \in [0..k_0]$, $R(i, l)$ l'est aussi. De plus pour tester si (S, c_0) est effectivement k-*reversal*-bornée, il suffit de tester que la propriété qui suit est vérifiée : il existe $b \in \mathbb{N}$, tel que pour toute paire $(l, i) \in \mathbb{N}^2$ vérifiant $l \in [(k + 1)..k_0]$ et $i \in [1..n]$ et pour toute configuration $((q, \mathbf{m}, \mathbf{r}), \mathbf{v}) \in R(i, l)$, il n'existe pas $k + 1$ différents indices $j_0, \ldots, j_k \in [1..l]$ vérifiant $\mathbf{v}(y_{i,j_0}) > b$, …, $\mathbf{v}(y_{i,j_k}) > b$. Cette dernière propriété peut effectivement être écrite sous la forme d'une formule de Presburger que nous ne détaillons pas ici. De plus, si (S, c_0) est k-*reversal*-bornée alors il existe un entier $b \in \mathbb{N}$ tel que (S, c_0) est k-*reversal*-b-bornée, et comme les compteurs $y_{i,j}$ mémorisent la valeur du compteur x_i au moment des différentes alternances au-dessus de b_0, on en déduit qu'il ne peut pas y avoir une exécution réalisant plus de k alternances au dessus de b. Et pour les mêmes raisons si (S, c_0) n'est pas k-*reversal*-bornée alors la propriété précédente n'est pas vérifiée. □

Ce lemme nous permet alors d'énoncer le résultat principal caractérisant l'ensemble $\mathbf{RB}(S, c_0)$ dans les cas où la machine à compteurs (S, c_0) est *reversal*-bornée.

Théorème 2.35 *Soit (S, c_0) une machine à compteurs reversal-bornée. L'ensemble $\mathbf{RB}(S, c_0)$ est fermé par le haut et il a un nombre fini d'éléments minimaux que l'on peut effectivement calculer.*

Preuve : Soit (S, c_0) une machine à compteurs *reversal*-bornée. Le fait que $\mathbf{RB}(S, c) \subseteq \mathbb{N}^2$ est un ensemble fermé par le haut est une conséquence directe du lemme 2.33. De plus, par le lemme 1.1,

nous savons que comme $\mathbf{RB}(S, c_0)$ est fermé par le haut, il a un nombre fini d'élément minimaux. Il nous reste à montrer que l'on peut effectivement les calculer.
Nous supposons que (S, c_0) est k-*reversal*-b-bornée avec $(k, b) \in \mathbb{N}^2$, comme nous l'avons dit dans la preuve du lemme 2.3.4, (S, c_0) étant *reversal*-bornée, il est possible de trouver une telle paire (k, b) en utilisant le résultat énoncé par le théorème 2.31. Nous pouvons alors calculer $b_0 \in \mathbb{N}$ le plus petit entier $b' \in \mathbb{N}$ tel que (S, c_0) est *reversal*-b'-bornée. En effet, pour cela, nous nous inspirons de la machine à compteurs initialisée $(S_{k,b}, c'_0)$ construite dans la section 2.3.3, et nous construisons une machine à compteurs *reversal*-bornée dans laquelle nous ajoutons, pour chaque compteur de S, b nouveaux compteurs dont le rôle est de compter pour chaque b' plus petit que b le nombre d'alternances entre les phases croissantes et décroissantes réalisé au dessus de b' par le compteur considéré. Comme ces compteurs que nous ajoutons ne font que croître, cette nouvelle machine est bien *reversal*-bornée et b_0 est alors le plus petit des b' pour lesquels le nombre d'alternances est borné, ce que l'on peut décider en utilisant le fait que l'ensemble d'accessibilité d'une machine à compteurs *reversal*-bornée est effectivement définissable dans Presburger. Cette méthode nous fournit de plus la constante k_0 pour laquelle (S, c_0) est k_0-*reversal*-b_0-bornée et pas $(k_0 - 1)$-*reversal*-b_0-bornée (il s'agit en fait de la constante bornant les compteurs calculant le nombre d'alternances effectuées au dessus de b_0). Nous avons alors que $(k_0, b_0) \in \mathbf{RB}(S, c_0)$ et de plus (k_0, b_0) est un élément minimal de $\mathbf{RB}(S, c_0)$. De plus, si (k', b') est un élément minimal de $\mathbf{RB}(S, c_0)$ différent de (k_0, b_0), nous avons nécessairement $k' < k_0$ et $b' > b_0$ par définition de k_0 et de b_0. Il nous reste alors pour chaque $k' \in [0..(k_0 - 1)]$ à tester en utilisant le lemme 2.3.4 si (S, c_0) est k'-*reversal*-bornée et si c'est le cas on peut calculer le plus petit b' tel que (S, c_0) est k'-*reversal*-b'-bornée, et ensuite déduire les éléments minimaux de $\mathbf{RB}(S, c_0)$. Cette méthode fonctionne car il n'y a qu'un nombre fini d'éléments dans $[0..(k_0 - 1)]$.□

Ce théorème nous donne donc une manière pratique de caractériser les éléments de $\mathbf{RB}(S, c_0)$ lorsque (S, c_0) est *reversal*-bornée, car comme $\mathbf{RB}(S, c_0)$ est fermé par le haut, pour tester qu'une paire d'entiers appartient à cet ensemble il suffit de tester si elle est plus grande qu'un des éléments minimaux calculés.

2.4 Les SAVE *reversal*-bornés

Dans cette section nous allons nous intéresser au cas particulier des SAVE *reversal*-bornés et nous allons montrer que nous pouvons décider si un SAVE est *reversal*-borné.

2.4.1 Graphe de couverture pour les SAVE

Nous rappelons que les SAVE sont des machines à compteurs dans lesquels il n'est pas autorisé de tester si un compteur à une valeur égale à une constante mais seulement de tester si une valeur de compteur est supérieure ou égale à une constante. Les SAVE constituent un modèle très répandu, en particulier car le problème d'accessibilité d'une configuration est décidable pour ce modèle (cf. théorème 1.42).

Dans [KM69], les auteurs proposent une méthode pour construire à partir d'un SAVE un arbre étiqueté appelé communément l'arbre de Karp et Miller. L'idée principale de cette construction est de "couvrir" avec un nombre fini d'éléments les configurations accessibles en utilisant le symbole ω de $\mathbb{N}_\omega$, lorsqu'un compteur n'est pas borné. On parle également d'arbre de couverture. Plus formellement si $S = \langle Q, X, E \rangle$ est un SAVE à n compteurs muni d'une configuration initiale $c_0 \in Q \times \mathbb{N}^n$ l'arbre

de Karp et Miller $\mathbf{KM}(S, c_0)$ associé à (S, c_0) est un arbre étiqueté $\langle P, E, R, l\rangle$ pour lequel :

– P est un ensemble fini de noeuds,
– $l : P \to Q \times \mathbb{N}_\omega^n$ est une fonction d'étiquetage qui associe à chaque noeud une étiquette,
– $R \subseteq P \times E \times P$ est la relation de transition.

Notation 2.36 *Pour représenter un noeud p de l'arbre de Karp et Miller étiqueté avec $l(p) = (q, \mathbf{w})$ avec $q \in Q$ et $\mathbf{w} \in \mathbb{N}_\omega^n$, nous écrirons parfois $p(q, w)$.*

Avant de présenter l'algorithme permettant de construire l'arbre de Karp et Miller, nous définissons la fonction **Acceleration** : $\mathbb{N}_\omega^n \times \mathbb{N}_\omega^n \to \mathbb{N}_\omega^n$ qui à tous vecteurs $\mathbf{w}, \mathbf{w}' \in \mathbb{N}_\omega^n$ tels que $\mathbf{w} \leq \mathbf{w}'$ associe le vecteur $\mathbf{w}'' = \mathbf{Acceleration}(\mathbf{w}, \mathbf{w}')$ défini de la façon suivante, pour tout $i \in [1..n]$, nous avons :

– si $\mathbf{w}(i) = \mathbf{w}'(i)$ alors $\mathbf{w}''(i) = \mathbf{w}(i)$, et,
– si $\mathbf{w}(i) \leq \mathbf{w}'(i)$ alors $\mathbf{w}''(i) = \omega$.

L'algorithme 2.37 nous montre alors comment construire l'arbre de Karp et Miller à partir d'un SAVE.

Algorithme 2.37 $T = \mathbf{KM}(\langle Q, X, E\rangle, c_0)$

Entrée : $(\langle Q, X, E\rangle, c_0)$ un SAVE ;
Sortie : $T = \langle P, E, R, l\rangle$ l'arbre de Karp et Miller associé à $(\langle Q, X, E\rangle, c_0)$;

1: $P = \{p_0\}$, $R = \emptyset$, $l(p_0) = c$
2: $ATraiter = \{p_0\}$
3: **Tant Que** $ATraiter \neq \emptyset$ **Faire**
4: Choisir $p(q, \mathbf{w}) \in ATraiter$
5: **Si** il n'y a pas de prédécesseur $p'(q, \mathbf{w})$ de p dans T **Alors**
6: **Pour** chaque $e = (q, (\#, \mu, \delta), q')$ dans E **Faire**
7: **Si** $\mu \leq \mathbf{w}$ **Alors**
8: soit $\mathbf{w}' = \mathbf{w} + \delta$
9: **Si** il existe un prédécesseur $p'(q', \mathbf{w}'')$ de p dans T tel que $\mathbf{w}' > \mathbf{w}''$ **Alors**
10: soit $\mathbf{w}' = \mathbf{Acceleration}(\mathbf{w}', \mathbf{w}'')$
11: **Fin Si**
12: Ajouter un nouveau noeud p' à P vérifiant $l(p') = (q', \mathbf{w}')$
13: Ajouter (p, e, p') à R
14: Ajouter p' à $ATraiter$
15: **Fin Si**
16: **Fin Pour**
17: **Fin Si**
18: Enlever p de $ATraiter$
19: **Fin Tant Que**

Dans [KM69], les auteurs ont montré que l'algorithme 2.37 terminait toujours et que l'arbre de Karp et Miller ainsi obtenu avait de bonnes propriétés, en particulier il permet de décider si les valeurs d'un compteur du SAVE sont bornées. Ce résultat peut s'exprimer ainsi :

Proposition 2.38 *[KM69] Soient $S = \langle Q, X, E\rangle$ un SAVE à n compteurs muni de la configuration initiale c_0 et $\mathbf{KM}(S, c_0) = \langle P, E, R, l\rangle$ l'arbre de Karp et Miller qui lui est associé. Il existe $b \in \mathbb{N}$,*

tel que pour toute configuration $(q, \boldsymbol{v}) \in \mathbf{Reach}(S, c_0)$ *et pour tout* $i \in [1..n]$, $\boldsymbol{v}(i) \leq b$ *si et seulement si pour tout noeud* $p \in P$ *avec* $l(p) = (q', \boldsymbol{w})$ *et pour tout* $i \in [1..n]$, $\boldsymbol{w}(i) \neq \omega$.

Ceci permet donc de résoudre le problème de bornitude d'un SAVE, qui consiste à savoir si il existe une constante bornant toutes les valeurs prises par les compteurs lors des différentes exécutions. Cependant notons que la construction de l'arbre de Karp et Miller nécessite un espace non-primitif récursif (plus de détails à ce sujet peuvent être trouvés dans [Jan87]). De plus, dans [Rac78], Rackoff montra qu'il n'était pas nécessaire de construire tout l'arbre de Karp et Miller pour résoudre le problème de bornitude, et il prouva en particulier que ce problème est EXPSPACE-complet.

Dans [VVN81], Valk et Vidal-Nacquet proposent une version modifiée de l'arbre de Karp et Miller pour pouvoir décider si le langage des transitions franchies lors de toutes les exécutions d'un SAVE initialisé est régulier ou non. La construction qu'ils utilisent alors est connue sous le nom de graphe de couverture. Pour l'obtenir, il suffit tout simplement de regrouper ensemble dans l'arbre de Karp et Miller les noeuds ayant les mêmes étiquettes. Formellement si (S, c_0) est un SAVE initialisé à n compteurs avec $S = \langle Q, X, E \rangle$, nous notons $\mathbf{CG}(S, c_0)$ son graphe de couverture égal au triplet $\langle N, E, \Delta \rangle$ où :

– $N \subseteq Q \times \mathbb{N}_\omega^n$ est un ensemble fini de noeuds, et,
– $\Delta \subseteq N \times E \times N$ est l'ensemble des transitions.

Exemple 2.39 *La figure 2.9 montre le graphe de couverture du SAVE* S *représenté à la figure 2.8 muni de la configuration initiale* $(q_1, 0)$. *La présence de* ω *dans le graphe de couverture nous permet de déduire que le compteur* x_1 *n'est pas borné dans* $TS(S, (q_1, 0))$.

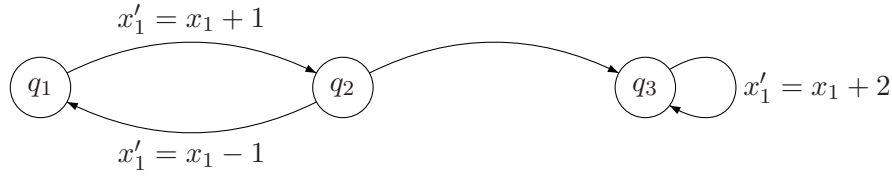


FIGURE 2.8 – Un SAVE S à 1 compteur

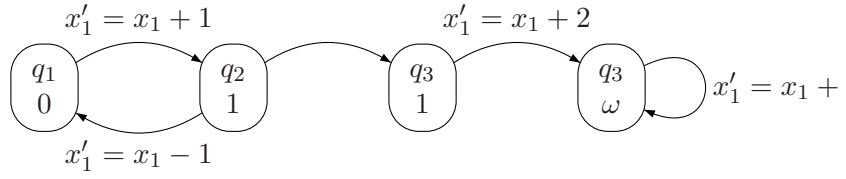


FIGURE 2.9 – Le graphe de couverture $\mathbf{CG}(S, (q_1, 0))$

Avant de donner les propriétés vérifiées par le graphe de couverture, nous introduisons quelques notions utiles. Comme pour les systèmes à compteurs (cf. section 1.4) nous parlons de cycles élémentaires pour le graphe de couverture d'un SAVE. Nous rappelons qu'un cycle élémentaire est un chemin

dans lequel tous les noeuds sont différents exceptés le noeud de départ et le noeud d'arrivée qui sont les mêmes. De plus, pour un vecteur $\mathbf{w} \in \mathbb{N}_\omega^n$, nous notons $\mathbf{Inf}(\mathbf{w})$ l'ensemble $\{i \in [1..n] \mid \mathbf{w}(i) = \omega\}$ et $\mathbf{Fin}(\mathbf{w}) = [1..n] \setminus \mathbf{Inf}(\mathbf{w})$. Finalement, si $S = \langle Q, X, E\rangle$ est un SAVE à n compteurs, pour tout $e = (q, (\#, \mu, \delta), q')$ appartenant à E, nous définissons le vecteur de déplacement associé à e, $\mathbf{D}_e \in \mathbb{Z}^n$ de telle façon que pour tout $i \in [1..n]$, $\mathbf{D}_e(i) = \delta(i)$. Nous étendons cette dernière définition aux mots dans E^* en disant que si $\sigma \in E^*$ et $e \in E$, nous avons $\mathbf{D}_{e.\sigma} = \mathbf{D}_e + \mathbf{D}_\sigma$ et par convention lorsque σ est le mot vide ϵ, nous posons $\mathbf{D}_\epsilon = \mathbf{0}$.

Soit (S, c_0) un SAVE tel que $S = \langle Q, X, E\rangle$. Le système de transitions qui lui est associé est noté $TS(S) = \langle Q \times \mathbb{N}^n, E, \rightarrow\rangle$ et son graphe de couverture $\mathbf{CG}(S, c_0) = \langle N, E, \Delta\rangle$. Nous avons alors les résultats suivants :

Théorème 2.40 *[KM69, VVN81]*

1. *Si $(q, \boldsymbol{w})$ est un noeud de $\mathbf{CG}(S, c_0)$, alors pour tout $k \in \mathbb{N}$, il existe $(q, \boldsymbol{v}) \in \mathbf{Reach}(S, c_0)$ tel que pour tout $i \in \mathbf{Inf}(\boldsymbol{w})$, $k \leq \boldsymbol{v}(i)$ et pour tout $i \in \mathbf{Fin}(\boldsymbol{w})$, $\boldsymbol{w}(i) = \boldsymbol{v}(i)$.*
2. *Pour $\sigma \in E^*$, si $c_0 \xrightarrow{\sigma} (q, \boldsymbol{v})$ alors il y a un unique chemin dans $\mathbf{CG}(S, c_0)$ étiqueté par σ et partant de c_0 vers un noeud $(q, \boldsymbol{w})$ tel que pour tout $i \in \mathbf{Fin}(\boldsymbol{w})$, $\boldsymbol{v}(i) = \boldsymbol{w}(i)$.*
3. *Si $\sigma \in E^*$ est un mot étiquetant un cycle dans G et $(q, \boldsymbol{w})$ est le noeud initial de ce cycle, alors il existe $(q, \boldsymbol{v}) \in \mathbf{Reach}(S, c)$ et $(q', \boldsymbol{v}')$ tels que $(q, \boldsymbol{v}) \xrightarrow{\sigma} (q, \boldsymbol{v}')$ et pour tout $i \in \mathbf{Fin}(\boldsymbol{w})$, $\boldsymbol{w}(i) = \boldsymbol{v}(i) = \boldsymbol{v}'(i)$.*

De ce théorème, nous déduisons le lemme suivant que nous utiliserons par la suite pour montrer que l'on peut décider si un SAVE est *reversal*-borné :

Lemme 2.41 *Si il existe un cycle élémentaire dans $\mathbf{CG}(S, c_0)$ de la forme :*

$$((q_1, \boldsymbol{w}_1), e_1, (q_2, \boldsymbol{w}_2))((q_2, \boldsymbol{w}_2), e_2, (q_3, \boldsymbol{w}_3)) \dots ((q_f, \boldsymbol{w}_f), e_f, (q_1, \boldsymbol{w}_1))$$

alors pour tout $k, l \in \mathbb{N}$, il existe $\boldsymbol{v}_1, \dots, \boldsymbol{v}_l \in \mathbb{N}^n$ tels que :

(i) $c \rightarrow^* (q_1, \boldsymbol{v}_1) \xrightarrow{\sigma} (q_1, \boldsymbol{v}_2) \xrightarrow{\sigma} \dots \xrightarrow{\sigma} (q_1, \boldsymbol{v}_l)$ *in $TS(S)$ avec $\sigma = e_1 \dots e_f$, et,*

(ii) *pour tout $j \in [1..l]$, pour tout $i \in \mathbf{Inf}(\boldsymbol{w}_1)$, $k \leq \boldsymbol{v}_j(i)$ et pour tout $i \in \mathbf{Fin}(\boldsymbol{w}_1)$, $\boldsymbol{w}_1(i) = \boldsymbol{v}_j(i)$.*

Preuve : Nous fixons $k, l \in \mathbb{N}$. Nous définissons $\mathbf{D}_{min} = \mathbf{Min}\{\mathbf{D}_\sigma(i) \mid i \in [1..n]\}$. Comme σ^{l-1} est un mot étiquetant un cycle de $\mathbf{CG}(S, c_0)$, par le point 3 du théorème 2.40, nous en déduisons qu'il existe $\mathbf{v}'_1, \dots, \mathbf{v}'_l \in \mathbb{N}^n$ tels que $c \rightarrow^* (q_1, \mathbf{v}'_1) \xrightarrow{\sigma} (q_1, \mathbf{v}'_2) \xrightarrow{\sigma} \dots \xrightarrow{\sigma} (q_1, \mathbf{v}'_l)$ dans $TS(S)$ et tels que pour tout $j \in [1..l]$, $\forall i \in \mathbf{Fin}(\mathbf{w}_1)$, $\mathbf{v}'_j(i) = \mathbf{w}_1(i)$. Nous considérons l'entier k' défini comme suit $k' = \mathbf{Max}(k + l.|\mathbf{D}_{min}|, \mathbf{Max}\{\mathbf{v}'_1(i) \mid i \in \mathbf{Inf}(\mathbf{w}_1)\})$ (où $|\mathbf{D}_{min}|$ désigne la valeur absolue de $\mathbf{D}_{min}$). D'après le point 1 du théorème 2.40, nous déduisons qu'il existe un $\mathbf{v}_1 \in \mathbb{N}^n$ tel que $(q_1, \mathbf{v}_1) \in \mathbf{Reach}(S, c)$ et $\forall i \in \mathbf{Fin}(\mathbf{w}_1)$, $\mathbf{v}_1(i) = \mathbf{w}_1(i)$ et $\forall i \in \mathbf{Inf}(\mathbf{w}_1)$, $k' \leq \mathbf{v}_1(i)$. Par définition de k', nous avons $\mathbf{v}'_1 \leq \mathbf{v}_1$, et par conséquent il existe $\mathbf{v}_2, \dots, \mathbf{v}_l$ tels que $(q_1, \mathbf{v}_1) \xrightarrow{\sigma} (q_1, \mathbf{v}_2) \xrightarrow{\sigma} \dots \xrightarrow{\sigma} (q_1.\mathbf{v}_l)$ en utilisant la propriété de monotonie des SAVE (cf. la proposition 1.25). De plus, pour tout $j \in [1..l]$, $\forall i \in \mathbf{Fin}(\mathbf{w}_1)$, $\mathbf{v}_j(i) = \mathbf{w}_1(i)$ (ceci est en effet vrai pour $\mathbf{v}_1$, et peut être déduit en utilisant le fait que σ est cycle commençant au noeud $(q_1, \mathbf{w}_1)$). Par propriété de k', nous avons aussi que pour tout $j \in [1..l]$, $\forall i \in \mathbf{Inf}(\mathbf{w}_1)$, $k \leq \mathbf{v}_j(i)$. □

Intuitivement ce dernier lemme nous dit que si on a un cycle élémentaire dans le graphe de couverture, il va exister une exécution dans le SAVE correspondant qui va passer autant de fois que l'on souhaite à travers ce cycle et les compteurs dont la composante est égale à ω dans le cycle vont pouvoir prendre des valeurs au-dessus d'un k donné à chaque tour du cycle et ceci pour tout $k \in \mathbb{N}$.

2.4.2 Décider si un SAVE est *reversal*-b-borné

Dans cette section nous allons montrer qu'étant donné un entier $b \in \mathbb{N}$, il est possible de décider si un SAVE initialisé est *reversal*-b-borné.

Soient $S = \langle Q, X, E\rangle$ une machine à n compteurs munie d'une configuration initiale $c_0 = (q, \mathbf{v})$ dans $Q \times \mathbb{N}^n$ et un entier $b \in \mathbb{N}$. Nous construisons une machine à $3n$ compteurs $S_b = \langle Q_b, X', E_b\rangle$ en ajoutant pour chaque compteur dans X, deux compteurs, un nous servira à tester si la valeur du compteur auquel il est associé est strictement plus grande que b, l'autre nous servira à compter le nombre d'alternances réalisées au dessus de b entre les phases de croissance et les phases de décroissance. Formellement, nous posons $Q_b = Q \times \{\uparrow, \downarrow\}^n$, $X' = \{x_1, \ldots, x_{3n}\}$ et E_b est définie comme suit, $((q, \mathbf{m}), (\#', \mu', \delta'), (q', \mathbf{m}')) \in E_b$ si et seulement si les conditions suivantes sont vérifiées :

1. il existe $(q, (\#, \mu, \delta), q') \in E$ tel que pour tout $i \in [1..n]$, $\#'(i) = \#(i)$, $\mu(i) = \mu'(i)$ et $\delta'(i) = \delta'(n+i) = \delta(i)$, et,
2. pour tout $i \in [(2n+1)..3n]$, $\#'(i) \in \{\leq\}$ et $\mu'(i) = 0$, et,
3. $\#'$,μ',δ',$\mathbf{m}$ et $\mathbf{m}'$ vérifient les conditions suivantes pour tout $i \in [1..n]$:

$\delta'(i)$	$\mathbf{m}(i)$	$\mathbf{m}'(i)$	$\mu'(n+i)$	$\#'(n+i)$	$\delta'(2n+i)$
$=0$	$-$	$\mathbf{m}(i)$	0	$\leq$	0
>0	$-$	$\uparrow$	0	$\leq$	0
>0	$\downarrow$	$\uparrow$	$b+1$	$\leq$	1
<0	$-$	$\downarrow$	0	$\leq$	0
>0	$\uparrow$	$\downarrow$	$b+1$	$\leq$	1

Nous voyons dans le tableau que les compteurs $x_1, \ldots, x_n$ ont le même comportement que les n compteurs de S, tout comme les compteurs $x_{2n+1}, \ldots, x_{2n}$ qui servent en plus à tester si la valeur du compteur est plus grande que b, et finalement les compteurs $x_{2n+1}, \ldots, x_{3n}$ comptent le nombre d'alternances réalisées au-dessus de b par les compteurs. Remarquosn que nous aurions pu prendre un codage avec uniquement $2n$ compteurs en ajoutant des états de contrôle pour tester si une valeur de compteur est strictement plus grande que b. Nous avons alors le lemme suivant :

Lemme 2.42 *Si S est un SAVE alors S_b est aussi un SAVE.*

Preuve : Ce résultat est directement déduit de la façon dont S_b est défini. □

Pour avoir la propriété énoncée par ce dernier lemme, dans les actions que nous ajoutons nous ne testons jamais si la valeur d'un compteur est égal à une constante, ceci a pour conséquence que nous ne testons pas si la valeur d'un compteur est plus petite que b, il se peut donc que dans certaines exécutions le nombre d'alternances que nous comptons soit inférieur à celui vraiment fait, mais il existe de toute façon une exécution où il est exact. Pour prouver cela, nous définissons la relation $\simeq \subseteq (Q \times \{\uparrow, \downarrow\}^n \times \mathbb{N}^n \times \mathbb{N}^n) \times (Q \times \{\uparrow, \downarrow\}^n \times \mathbb{N}^{3n})$ entre les configurations de $TS_b(S)$ et celles de $TS(S_b)$ de la façon suivante, pour tout $(q, \mathbf{m}, \mathbf{v}, \mathbf{r}) \in Q \times \{\uparrow, \downarrow\}^n \times \mathbb{N}^n \times \mathbb{N}^n$ et pour tout $(q', \mathbf{m}', \mathbf{v}') \in Q \times \{\uparrow, \downarrow\}^n \times \mathbb{N}^{3n}$, nous avons $(q, \mathbf{m}, \mathbf{v}, \mathbf{r}) \simeq (q', \mathbf{m}', \mathbf{v}')$ si et seulement si :

– $q = q'$,
– $\mathbf{m} = \mathbf{m}'$,
– pour tout $i \in [1..n]$, $\mathbf{v}(i) = \mathbf{v}'(i) = \mathbf{v}'(n+i)$,

– pour tout $i \in [1..n]$, $\mathbf{v}(2n+i) \leq \mathbf{r}(i)$.

Nous notons $TS_b(S) = \langle Q \times \{\uparrow,\downarrow\}^n \times \mathbb{N}^n \times \mathbb{N}^n, \rightarrow_b \rangle$ et $TS(S_b) = \langle Q \times \{\uparrow,\downarrow\}^n \times \mathbb{N}^{3n}, \Rightarrow \rangle$. Nous avons alors le lemme suivant :

Lemme 2.43 *Soient $c_1 = (q_1, \boldsymbol{m}_1, \boldsymbol{v}_1, \boldsymbol{r}_1)$ dans $Q \times \{\uparrow,\downarrow\}^n \times \mathbb{N}^n \times \mathbb{N}^n$ et $c'_1 = (q_1, \boldsymbol{m}_1, \boldsymbol{v}'_1)$ dans $Q \times \{\uparrow,\downarrow\}^n \times \mathbb{N}^{3n}$ tels que $c_1 \simeq c'_1$. Nous avons les propriétés suivantes :*

– *Pour tout $c_2 = (q_2, \boldsymbol{m}_2, \boldsymbol{v}_2, \boldsymbol{r}_2)$ dans $Q \times \{\uparrow,\downarrow\}^n \times \mathbb{N}^n \times \mathbb{N}^n$ tel que $c_1 \rightarrow_b c_2$, il existe $c'_2 = (q_2, \boldsymbol{m}_2, \boldsymbol{v}'_2)$ dans $Q \times \{\uparrow,\downarrow\}^n \times \mathbb{N}^{3n}$ tel que :*

  1. *$c'_1 \Rightarrow c'_2$,*
  2. *$c_2 \simeq c'_2$,*
  3. *pour tout $i \in [1..n]$, $\boldsymbol{v}'_2(2n+i) - \boldsymbol{v}'_1(2n+i) = \boldsymbol{r}_2(i) - \boldsymbol{r}_1(i)$.*

– *pour tout $c'_2 = (q_2, \boldsymbol{m}_2, \boldsymbol{v}'_2)$ dans $Q \times \{\uparrow,\downarrow\}^n \times \mathbb{N}^{3n}$ tel que $c'_1 \Rightarrow c'_2$, il existe $c_2 = (q_2, \boldsymbol{m}_2, \boldsymbol{v}_2, \boldsymbol{r}_2)$ dans $Q \times \{\uparrow,\downarrow\}^n \times \mathbb{N}^n \times \mathbb{N}^n$ tel que :*

  1. *$c_1 \rightarrow_b c_2$,*
  2. *$c_2 \simeq c'_2$,*
  3. *pour tout $i \in [1..n]$, $\boldsymbol{v}'_2(2n+i) - \boldsymbol{v}'_1(2n+i) \leq \boldsymbol{r}_2(i) - \boldsymbol{r}_1(i)$.*

Preuve : La preuve de ce théorème est immédiate par la façon dont nous construisons E_b et définissons la relation $\simeq$ et se fait par une étude de cas similaire à celle que nous avons faite pour lemme 2.15 par exemple. □

Le cas le plus intéressant de ce lemme est le deuxième, car nous avons pour tout $i \in [1..n]$, $\mathbf{v}'_1(2n+i) - \mathbf{v}'_2(2n+i) \leq \mathbf{r}_2(i) - \mathbf{r}_1(i)$ et non pas pour tout $i \in [1..n]$, $\mathbf{v}'_1(2n+i) - \mathbf{v}'_2(2n+i) = \mathbf{r}_2(i) - \mathbf{r}_1(i)$, ceci est dû au fait qu'il peut y avoir une transition dans E_b pour laquelle le i-ème compteur alterne au dessus de b entre une phase de croissance et de décroissance, mais pour laquelle nous n'augmentons pas le $(2n+i)$-ème compteur. Ceci car nous ne testons pas si la valeur d'un compteur est plus petite que b, mais seulement si elle est plus grande.

Nous définissons alors à partir de la configuration initiale $c_0 = (q_0, \mathbf{v}_0)$ de $TS(S)$, une configuration initiale de $TS(S_b)$ de la façon suivante, $c'_0 = (q_0, \uparrow, \mathbf{v}'_0)$ où $\uparrow$ désigne le vecteur dont toutes les composantes sont égales à $\uparrow$ et pour tout $i \in [1..n]$, $\mathbf{v}'_0(i) = \mathbf{v}'_0(n+i) = \mathbf{v}_0(i)$ et $\mathbf{v}'_0(2n+1) = 0$. Il est alors clair que $(q_0, \uparrow, \mathbf{v}_0, \mathbf{0}) \simeq c'_0$. À partir de cette remarque et du résultat du lemme précédent, nous obtenons ce lemme :

Lemme 2.44

1. *Pour tout $(q, \boldsymbol{m}, \boldsymbol{v}, \boldsymbol{r}) \in \mathbf{Reach}_b(S, c_0)$, il existe $(q, \boldsymbol{m}, \boldsymbol{v}') \in \mathbf{Reach}(S_b, c'_0)$ tel que $(q, \boldsymbol{m}, \boldsymbol{v}, \boldsymbol{r}) \simeq (q, \boldsymbol{m}, \boldsymbol{v}')$ et pour tout $i \in [1..n]$, $\boldsymbol{r}(i) = \boldsymbol{v}'(2n+i)$.*
2. *Pour tout $(q, \boldsymbol{m}, \boldsymbol{v}') \in \mathbf{Reach}(S_b, c'_0)$, il existe $(q, \boldsymbol{m}, \boldsymbol{v}, \boldsymbol{r}) \in \mathbf{Reach}_b(S, c'_0)$ tel que $(q, \boldsymbol{m}, \boldsymbol{v}, \boldsymbol{r}) \simeq (q, \boldsymbol{m}, \boldsymbol{v})$.*

Preuve : Nous montrons d'abord le point 1. de ce lemme. Supposons qu'il existe $(q, \mathbf{m}, \mathbf{v}, \mathbf{r}) \in \mathbf{Reach}_b(S, c_0)$. Alors il existe une exécution dans $TS_b(S)$ tel que $(q_0, \uparrow, \mathbf{v}_0, \mathbf{0}) \rightarrow_b (q_1, \mathbf{m}_1, \mathbf{v}_1, \mathbf{r}_1) \ldots \rightarrow_b (q_f, \mathbf{m}_f, \mathbf{v}_f, \mathbf{r}_f)$ avec $(q_f, \mathbf{m}_f, \mathbf{v}_f, \mathbf{r}_f) = (q, \mathbf{m}, \mathbf{v}, \mathbf{r})$. Nous raisonnons alors par induction sur la taille de cette exécution. Pour $f = 0$, nous avons bien $(q_0, \uparrow, \mathbf{v}_0, \mathbf{0}) \simeq c'_0$ et $c'_0 \in \mathbf{Reach}(S_b, c'_0)$ et pour tout $i \in [1..n]$, $\mathbf{v}'_0(2n+i) = 0$. Nous supposons maintenant que pour tout $j \in [0..f-1]$, le lemme est vrai pour $(q_j, \mathbf{m}_j, \mathbf{v}_j, \mathbf{r}_j)$. Par hypothèse d'induction, il existe $(q_{f-1}, \mathbf{m}_{f-1}, \mathbf{v}'_{f-1}) \in \mathbf{Reach}(S, c'_0)$ tel que $(q_{f-1}, \mathbf{m}_{f-1}, \mathbf{v}_{f-1}, \mathbf{r}_{f-1}) \simeq (q_{f-1}, \mathbf{m}_{f-1}, \mathbf{v}'_{f-1})$ et pour tout $i \in [1..n]$, $\mathbf{r}_{f-1}(i) = \mathbf{v}'_{f-1}(2n+i)$. Nous avons de plus $(q_{f-1}, \mathbf{m}_{f-1}, \mathbf{v}_{f-1}, \mathbf{r}_{f-1}) \rightarrow_b (q, \mathbf{m}, \mathbf{v}, \mathbf{r})$, donc par le lemme 2.43 il existe $(q, \mathbf{m}, \mathbf{v}')$ tel que :

- $(q, \mathbf{m}, \mathbf{v}, \mathbf{r}) \simeq (q, \mathbf{m}, \mathbf{v}')$, et,
- $(q_{f-1}, \mathbf{m}_{f-1}, \mathbf{v}'_{f-1}) \Rightarrow (q, \mathbf{m}, \mathbf{v}')$, et,
- pour tout $i \in [1..n]$, $\mathbf{v}'(2n+i) - \mathbf{v}'_{f-1}(2n+i) = \mathbf{r}(i) - \mathbf{r}_{f-1}(i)$.

Par conséquent, $(q, \mathbf{m}, \mathbf{v}') \in \mathbf{Reach}(s_b, c'_0)$ et pour tout $i \in [1..n]$, $\mathbf{v}'(2n+i) = \mathbf{r}(i) - \mathbf{r}_f(i) + \mathbf{v}'_f(i) = \mathbf{r}(i)$.
Le point 2. se prouve de la même façon par induction en utilisant toujours le lemme 2.43. □

Finalement de ce lemme, nous déduisons le résultat liant le fait que (S, c_0) soit *reversal-b*-bornée et les valeurs de compteurs présentes dans $\mathbf{Reach}(S_b, c'_0)$.

Lemme 2.45 *La machine à compteurs (S, c_0) est reversal-b-bornée si et seulement si il existe $k \in \mathbb{N}$ tel que pour tout $(q, \boldsymbol{m}, \mathbf{v}') \in \mathbf{Reach}(S_b, c'_0)$, pour tout $i \in [1..n]$, $\mathbf{v}'(2n+i) \leq k$.*

Preuve : Supposons que (S, c_0) est *reversal-b*-bornée alors il existe $k \in \mathbb{N}$ tel que (S, c_0) est *k-reversal-b*-bornée. Pour tout $(q, \mathbf{v}, \mathbf{m}, \mathbf{r}) \in \mathbf{Reach}_b(S, c_0)$, nous avons pour tout $i \in [1..n]$, $\mathbf{r}(i) \leq k$. Soit $(q, \mathbf{m}, \mathbf{v}') \in \mathbf{Reach}(S_b, c'_0)$. D'après le lemme 2.44, il existe $(q, \mathbf{m}, \mathbf{v}, \mathbf{r}) \in \mathbf{Reach}_b(S, c_0)$ tel que $(q, \mathbf{m}, \mathbf{v}, \mathbf{r}) \simeq (q, \mathbf{m}, \mathbf{v}')$. Par définition de $\simeq$, nous avons pour tout $i \in [1..n]$, $\mathbf{v}'(2n+i) \leq \mathbf{r}(i) \leq k$.

Supposons qu'il existe $k \in \mathbb{N}$ tel que pour tout $(q, \mathbf{m}, \mathbf{v}') \in \mathbf{Reach}(S_b, c'_0)$, pour tout $i \in [1..n]$, $\mathbf{v}'(2n+i) \leq k$. Soit $(q, \mathbf{m}, \mathbf{v}, \mathbf{r}) \in \mathbf{Reach}_b(S, c_0)$. D'après le lemme 2.44, il existe $(q, \mathbf{m}, \mathbf{v}') \in \mathbf{Reach}(S_b, c'_0)$ tel que pour tout $i \in [1..n]$, $\mathbf{r}(i) = \mathbf{v}'(2n+i)$ et comme pour tout $i \in [1..n]$, on a $\mathbf{r}(2n+i) \leq k$, on en déduit que pour tout $i \in [1..n]$ $\mathbf{r}(i) \leq k$. Par conséquent (S, c_0) est *k-reversal-b*-bornée. □

Du fait que l'on puisse décider grâce au graphe de couverture, si dans un SAVE les valeurs d'un compteur sont bornées et que si S est un SAVE, S_b est également un SAVE, on obtient le résultat principal de cette section, c'est-à-dire :

Théorème 2.46 *Étant donné $b \in \mathbb{N}$, le problème de savoir si un SAVE est reversal-b-borné est décidable.*

Nous rappelons que ce résultat n'est plus vrai lorsque l'on considère des machines à compteurs à la place des SAVE. Nous allons de plus voir que le SAVE S_0 (c'est-à-dire le SAVE S_b avec $b = 0$) construit à partir du SAVE S peut même nous servir à décider si (S, c_0) est *reversal*-borné.

Exemple 2.47 *La figure 2.10 représente le SAVE à 3 compteurs S_0 construit à partir du SAVE S de la figure 2.8. Dans ce SAVE, on voit que le compteur x_2 a le même comportement que le compteur x_1 et*

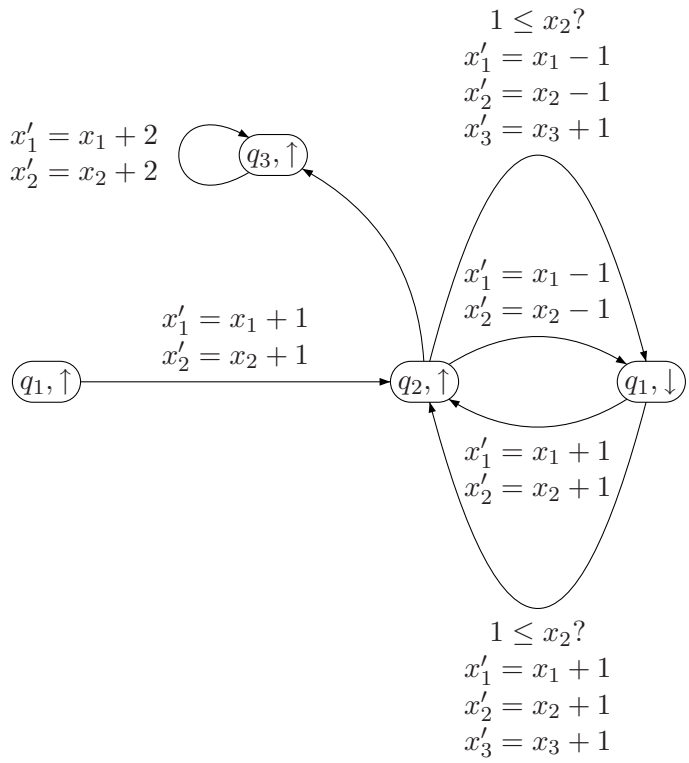


FIGURE 2.10 – Le SAVE S_0

que l'on s'en sert uniquement pour tester si la valeur de ce compteur est strictement plus grande que 0*. Quant au compteur* x_3*, il nous sert à compter le nombre d'alternances entre les phases de croissance et de décroissance réalisées par le compteur* x_1 *au-dessus de* 0*. Finalement, la figure 2.11 représente le graphe de couverture associé au SAVE* S_0 *munie de la configuration initiale* $c'_0 = ((q_0, \uparrow), (0, 0, 0))$*. On voit grâce à ce graphe de couverture ques les valeurs du compteur* x_3 *ne sont pas bornées dans* **Reach**(S_0, c'_0)*, car un* ω *est associé à ce compteur dans certains noeuds du graphe. En utilisant le théorème 2.46, nous pouvons déduire que le SAVE initialisé* $(S, (q_0, 0))$ *n'est pas reversal-*0*-borné.*

2.4.3 Décider si un SAVE est *reversal*-borné

Dans cette section, nous allons finir par montrer le résultat principal concernant les SAVE, à savoir que l'on peut décider si un SAVE initialisé (S, c_0) est *reversal*-borné. Pour cela, nous proposons une condition nécessaire et suffisante sur le graphe de couverture du SAVE (S_0, c'_0) dont nous avons décrit la construction et les propriétés dans la section précédente. Remarquons que le résultat que nous voulons prouver ici n'est pas une conséquence directe du théorème 2.46, car il n'est pas possible de tester pour chaque $b \in \mathbb{N}$ si (S, c_0) est *reversal*-b-borné, cette méthode ne terminant pas lorsque le SAVE initialisé (S, c_0) n'est pas *reversal*-borné.

Nous considérons un SAVE à n compteurs S muni d'une configuration initiale $c_0 = (q_0, \mathbf{v}_0)$. Nous notons $TS(S) = \langle Q, E, \rightarrow \rangle$. Pour tout $b \in \mathbb{N}$, nous supposons que $TS_b(S)$ est égal à $\langle Q \times \{\uparrow, \downarrow\}^n \times \mathbb{N}^n \times \mathbb{N}^n, E, \rightarrow_b \rangle$. Finalement nous utiliserons également le SAVE à $3n$ compteurs $S_0 = \langle Q_0, X', E_0 \rangle$ munie de la configuration initiale c'_0 dont la construction est fournie dans la section précédente. Nous

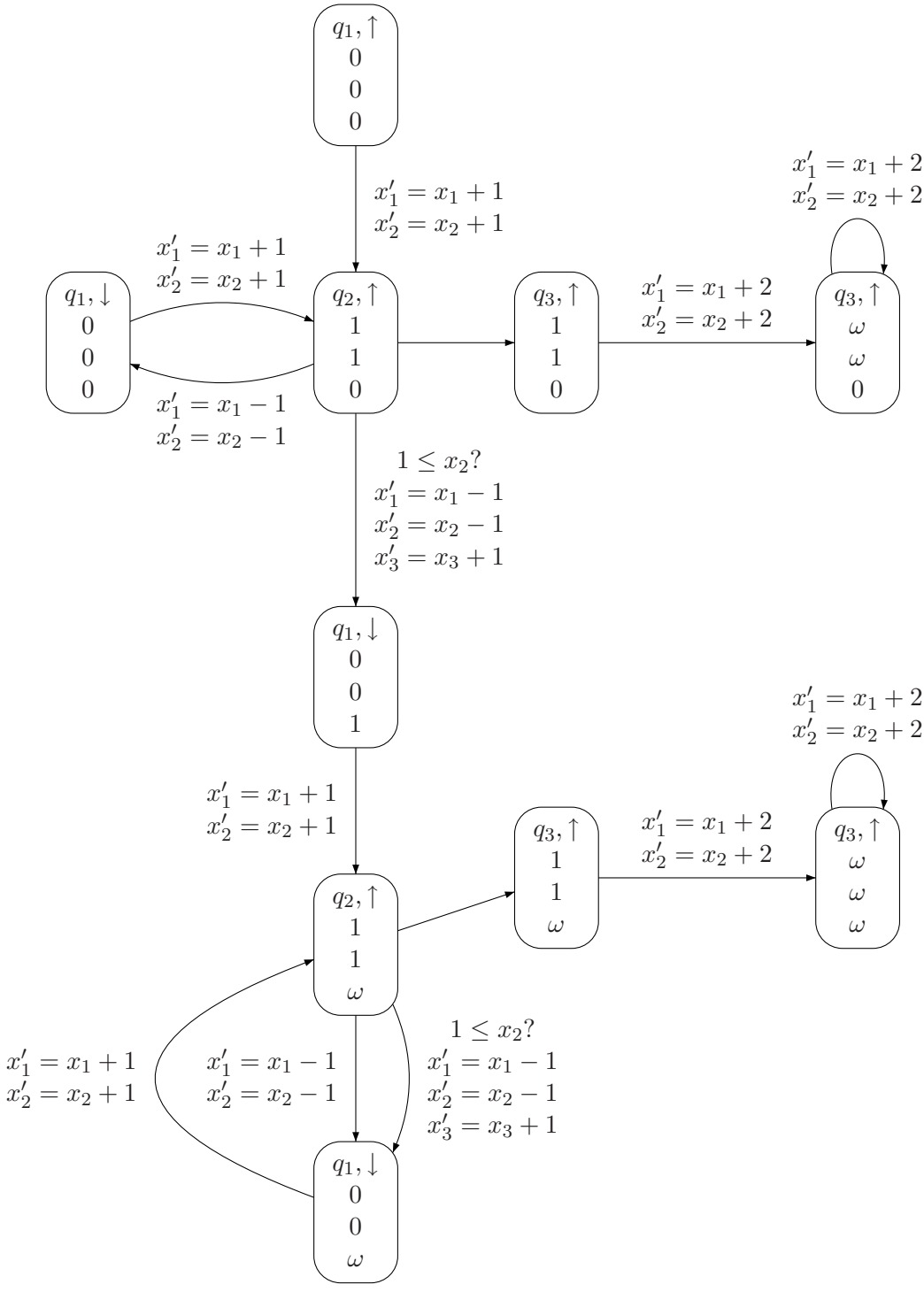


FIGURE 2.11 – Le graphe de couverture $\mathbf{CG}(S_0, c'_0)$

avons aussi $TS(S_0) = \langle Q_0 \times \mathbb{N}^{3n}, E_0, \Rightarrow \rangle$.

Afin de montrer le résultat souhaité, nous montrons d'abord le lien entre les systèmes de transition $TS_0(S)$ où l'on compte les alternances qui ont lieu au dessus de 0 et $TS_b(S)$ où l'on compte les alternances qui ont lieu au dessus de b. Ceci nous permettra de faire le lien entre le fait que (S, c_0) soit *reversal-b*-borné et la forme des exécutions dans $TS(S_0)$. Finalement, nous obtiendrons, en utilisant la relation $\simeq$ introduite dans la section précédente, une condition nécessaire et suffisante sur le graphe de couverture $\mathbf{CG}(S_0, c_0')$ pour décider si (S, c_0) est *reversal*-borné ou non.

Tout d'abord, nous faisons la connexion entre les exécutions dans $TS_0(S)$ celle dans $TS_b(S)$.

Lemme 2.48 *Soit $b, i \in \mathbb{N}$. Si il existe $c_1, c_2, c_3 \in Q \times \{\downarrow, \uparrow\}^n \times \mathbb{N}^n \times \mathbb{N}^n$ et $\sigma_1, \sigma_2, \sigma_3 \in E^*$ tels que :*

(i) $(q_0, \uparrow, \boldsymbol{v}_0, \boldsymbol{0}) \xrightarrow{\sigma_1}_0 c_1 \xrightarrow{\sigma_2}_0 c_2 \xrightarrow{\sigma_3}_0 c_3$ *dans $TS_0(S)$, et,*

(ii) *pour tout $j \in [1..3]$, $b < \boldsymbol{v}_j(i)$ et $\boldsymbol{r}_1(i) < \boldsymbol{r}_2(i) < \boldsymbol{r}_3(i)$ (avec $c_j = (q_j, \boldsymbol{m}_j, \boldsymbol{v}_j, \boldsymbol{r}_j)$)*

alors il existe $c_1', c_2', c_3' \in Q \times \{\downarrow, \uparrow\}^n \times \mathbb{N}^n \times \mathbb{N}^n$ tels que :

$$(q_0, \uparrow, \boldsymbol{v}_0, \boldsymbol{0}) \xrightarrow{\sigma_1}_b c_1' \xrightarrow{\sigma_2}_b c_2' \xrightarrow{\sigma_3}_b c_3' \text{ dans } TS_b(S) \text{ et } \boldsymbol{r}_1'(i) < \boldsymbol{r}_3'(i)$$

Preuve : Par définition de $\rightarrow_0$ et$\rightarrow_b$, si nous considérons c_1, c_2 et c_3 tels que $(q_0, \uparrow, \mathbf{v}_0, \mathbf{0}) \xrightarrow{\sigma_1}_0 c_1 \xrightarrow{\sigma_2}_0 c_2 \xrightarrow{\sigma_3}_0 c_3$, alors il existe nécessairement $c_1', c_2', c_3' \in Q \times \{\downarrow, \uparrow\}^n \times \mathbb{N}^n \times \mathbb{N}^n$ tels que $(q_0, \uparrow, \mathbf{v}_0, \mathbf{0}) \xrightarrow{\sigma_1}_b c_1' \xrightarrow{\sigma_2}_b c_2' \xrightarrow{\sigma_3}_b c_3'$. Remarquons que nous avons pour tout $j \in [1..3]$, $q_j = q_j'$, $\mathbf{m}_j = \mathbf{m}_j'$ et $\mathbf{v}_j = \mathbf{v}_j'$. De plus, comme $\mathbf{r}_1(i) < \mathbf{r}_2(i) < \mathbf{r}_3(i)$, nous en déduisons que le i-ème compteur réalise au moins deux alternances entre des phases de croissance et de décroissance lors de la séquence $\sigma_1\sigma_2\sigma_3$, et comme $\mathbf{v}_1(i) > b$, $\mathbf{v}_2(i) > b$ et $\mathbf{v}_3(i) > b$, nous en déduisons qu'au moins une de ces alternances a été effectuée quand la valeur du i-ème compteur était strictement supérieure à b, par conséquent nous avons $\mathbf{r}_1'(i) < \mathbf{r}_3'(i)$. □

Nous obtenons à partir de ce lemme une caractérisation des exécutions de $TS_0(S)$ qui si elle est vérifiée implique que (S, c_0) n'est pas k-*reversal-b*-bornée.

Lemme 2.49 *Soient $k, b \in \mathbb{N}$. Si il existe $c_1, c_2, \ldots, c_{2k+1} \in Q \times \{\downarrow, \uparrow\}^n \times \mathbb{N}^n \times \mathbb{N}^n$ et $\sigma_1, \ldots, \sigma_{2k+1} \in E^*$ tels que :*

$$(q_0, \uparrow, \boldsymbol{v}_0, \boldsymbol{0}) \xrightarrow{\sigma_1}_0 c_1 \xrightarrow{\sigma_2}_0 \ldots \xrightarrow{\sigma_{2k+1}}_0 c_{2k+1}$$

et si il existe $i \in [1..n]$ tel que pour tout $j \in [1..(2k+1)]$, $b < \boldsymbol{v}_j(i)$ et pour tout $j \in [1..2k]$, $\boldsymbol{r}_j(i) < \boldsymbol{r}_{j+1}(i)$, alors $(S, (q, \boldsymbol{v}))$ n'est pas k-reversal-b-bornée.

Preuve : En utilisant le lemme 2.48, nous pouvons en effet construire une exécution dans $TS_b(S)$ partant de $(q, \uparrow, \mathbf{v}, \mathbf{0})$, qui ne satisfait pas la propriété des exécutions dans les systèmes k-*reversal-b*-bornés, c'est-à-dire qui réalise plus de k alternances au dessus de b. □

Nous donnons maintenant un autre lemme un peu technique, qui peut être vu comme le pendant du lemme précédent, car il fournit une description nécessaire sur les exécutions de $TS_0(S)$ lorsque (S, c_0) n'est pas *reversal-b*-bornée.

Lemme 2.50 *Soit $b \in \mathbb{N}$. Si (S, c_0) n'est pas reversal-b-borné, alors il existe $i \in [1..n]$ tel que pour tout $k \in \mathbb{N}$, il existe $c_1, \ldots, c_k \in Q \times \{\downarrow, \uparrow\}^n \times \mathbb{N}^n \times \mathbb{N}^n$ qui vérifient :*

(i) *$(q_0, \uparrow, \boldsymbol{v}_0, \boldsymbol{0}) \rightarrow_0^* c_1 \rightarrow_0^* \ldots \rightarrow_0^* c_k$ est une exécution dans $TS_0(S)$, et,*

(ii) *pour tout $j \in [1..k]$, $b < \boldsymbol{v}_j(i)$ et pour tout $j \in [1..k-1]$, $\boldsymbol{r}_j(i) < \boldsymbol{r}_{j+1}(i)$ (avec $\boldsymbol{c}_j = (q_j, \boldsymbol{m}_j, \boldsymbol{v}_j, \boldsymbol{r}_j)$).*

Preuve : Soit $k \in \mathbb{N}$. Nous supposons que (S, c_0) n'est pas *reversal-b*-borné, en particulier (S, c_0) n'est pas $(k+1)$-*reversal-b*-borné. Par conséquent, il existe $(q', \mathbf{m}', \mathbf{v}', \mathbf{r}') \in \mathbf{Reach}_b(S, c_0)$ et $i \in \mathbb{N}$ tel que $\mathbf{r}'(i) \geq k+1$. Comme les compteurs qui comptent les alternances entre les phase de croissance et de décroissance dans $TS_b(S)$ ne font que croître et à chaque pas au plus d'une valeur valant 1 et seulement lorsque le compteur associé est strictement plus grand que b, on en déduit qu'il existe $c'_1, \ldots, c'_k \in Q \times \{\downarrow, \uparrow\}^n \times \mathbb{N}^n \times \mathbb{N}^n$ tels que $(q_0, \uparrow, \mathbf{v}_0, \mathbf{0}) \rightarrow_b^* c'_1 \rightarrow_b^* \ldots \rightarrow_b^* c'_k$ et tels que pour tout $j \in [1..k]$, $c'_j = (q'_j, \mathbf{m}'_j, \mathbf{v}'_j, \mathbf{r}'_j)$ avec $b < \mathbf{v}'_j(i)$ et $\mathbf{r}'_j(i) = j$. De plus, par définition de $\rightarrow_b$ et de $\rightarrow_0$, nous avons qu'il existe $c_1, ..., c_k \in Q \times \{\downarrow, \uparrow\}^n \times \mathbb{N}^n \times \mathbb{N}^n$ qui satisfont la propriété donnée par le lemme. □

Les lemmes techniques que nous avons donnés précédemment vont nous être maintenant utiles pour donner une caractérisation nécessaire et suffisante sur le graphe de couverture $CG(S_0, c'_0)$ permettant de dire si le SAVE (S, c_0) est *reversal-b*-borné ou non pour $b \in \mathbb{N}$. En effet :

Lemme 2.51 *Soit $b \in \mathbb{N}$. (S, c_0) est reversal-b-borné si et seulement si pour tout $i \in [1..n]$, tout noeud $(q, \boldsymbol{w})$ appartenant à un cycle élémentaire de $\mathbf{CG}(S_0, c'_0)$ étiqueté par un mot $\sigma \in E^*$ tel que $\mathbf{D}_\sigma(2n+i) > 0$ vérifie $\boldsymbol{w}(i) \leq b$.*

Preuve : D'abord, nous supposons que le SAVE initialisé (S, c_0) est *reversal-b*-borné. Il existe donc $k \in \mathbb{N}$ tel que (S, c_0) est k-*reversal-b*-borné. Nous raisonnons par l'absurde. Supposons qu'il existe $i \in \mathbb{N}$ et un noeud $(q, \mathbf{w})$ qui appartient à un cycle élémentaire de la forme $((q_1, \mathbf{w}_1), e_1(q_2, \mathbf{w}_2))$ $((q_2, \mathbf{w}_2), e_2, (q_3, \mathbf{w}_3)) \ldots ((q_f, \mathbf{w}_f), e_f, (q_1, \mathbf{w}_1))$ de $\mathbf{CG}(S_0, c'_0)$ avec $D_{e_1 \ldots e_f}(2n+i) > 0$, et tel que $\mathbf{w}(i) > b$. Nous remarquons que comme $\mathbf{w}(i) > b$, nous avons soit $\mathbf{w}(i) = b' > b$ soit $\mathbf{w}(i) = \omega$. Nous notons σ le mot $e_1.e_2 \ldots e_f$. Par le lemme 2.41, nous déduisons qu'il existe $\mathbf{v}_1, \ldots, \mathbf{v}_{2k+1} \in \mathbb{N}^{2n}$ tels que nous avons dans $TS(S_0)$:

$$c_0 \Rightarrow^* (q', \mathbf{v}_1) \stackrel{\sigma}{\Rightarrow} \ldots \stackrel{\sigma}{\Rightarrow} (q', \mathbf{v}_{2k+1})$$

avec pour tout $j \in [1..2k+1]$, $b < \mathbf{v}_j(i)$. De plus comme $D_\sigma(2n+i) > 0$, nous déduisons que pour tout $j \in [1..2k]$, $\mathbf{v}_j(2n+i) < \mathbf{v}_{j+1}(2n+i)$. En utilisant le lemme 2.43 caractérisant la relation $\simeq$ entre les configurations de $TS_0(S)$ et de $TS(S_0)$ et par le lemme 2.49,nous déduisons que (S, c_0) n'est pas k-*reversal-b*-borné, ce qui constitue une contradiction.

Nous supposons maintenant que pour tout $i \in [1..n]$, pour tout noeud $(q, \mathbf{w})$ appartenant à un cycle élémentaire $((q_1, \mathbf{w}_1), e_1, (q_2, \mathbf{w}_2))((q_2, \mathbf{w}_2), e_2, (q_3, \mathbf{w}_3)) \ldots ((q_f, \mathbf{w}_f), e_f, (q_1, \mathbf{w}_1))$ de $\mathbf{CG}(S_0, c'_0)$ avec $D_{e_1 \ldots e_f}(2n+i) > 0$, nous avons $\mathbf{w}(i) \leq b$. Encore une fois, nous raisonnons par l'absurde. Supposons que (S, c_0) n'est pas *reversal-b*-borné. Soit N le nombre de noeuds dans $\mathbf{CG}(S_0, c'_0)$. En utilisant le lemme 2.50 et la caractérisation donnée par le lemme 2.43 sur la relation $\simeq$ entre les configurations de $TS(S_0)$ et de $TS_0(S)$, nous déduisons qu'il existe $i \in \mathbb{N}$ et $(q_1, \mathbf{v}_1), \ldots, (q_{N+1}, \mathbf{v}_{N+1}) \in Q \times \mathbb{N}^{3n}$ et $\sigma_1, \ldots \sigma_N$ tels que dans $TS(S_0)$ nous avons $c_0 \Rightarrow^* (q_1, \mathbf{v}_1) \stackrel{\sigma_1}{\Rightarrow} \ldots \stackrel{\sigma_N}{\Rightarrow} (q_{N+1}, \mathbf{v}_{N+1})$ et

pour tout $j \in [1..N+1]$, $b < \mathbf{v}_j(i)$ et pour tout $j \in [1..N]$, $\mathbf{v}_j(2n+i) < \mathbf{v}_{j+1}(2n+i)$. Premièrement, de ce dernier point, nous obtenons immédiatement par définition de $\mathbf{D}$ que pour tout $1 \leq j \leq N$, $\mathbf{D}_{\sigma_j}(2n+i) > 0$. Deuxièmement, d'après le théorème 2.40, nous pouvons dire qu'il existe $(q_1.\mathbf{w}_1), \ldots, (q_{N+1}, \mathbf{w}_{N+1})$ noeuds dans $\mathbf{CG}(S_0, c'_0)$ tels que dans le graphe de couverture nous avons le chemin (noté schématiquement avec des flèches) $(q_1, \mathbf{w}_1) \xrightarrow{\sigma_1} \ldots \xrightarrow{\sigma_N} (q_{N+1}, \mathbf{w}_{N+1})$ et pour $j \in [1..N+1]$,$\mathbf{v}_j \leq \mathbf{w}_j$. N étant le nombre de noeuds dans $\mathbf{CG}(S_0, c'_0)$, nous en déduisons qu'il existe $(q'_1.\mathbf{w}'_1), \ldots, (q'_f, \mathbf{w}'_f)$ noeuds dans $\mathbf{CG}(S_0, c'_0)$ et $e_1, \ldots, e_f \in E_0$ tels que : $((q'_1, \mathbf{w}'_1), e_1, (q'_2, \mathbf{w}'_2))((q'_2, \mathbf{w}'_2), e_2, (q'_2, \mathbf{w}'_2)) \ldots ((q'_f, \mathbf{w}'_f), e_f, (q'_1, \mathbf{w}'_1))$ est un cycle dans $\mathbf{CG}(S_0, c'_0)$ et $D_{e_1 \ldots e_f}(2n+i) > 0$ et il existe $j \in [1..N]$ tel que $(q'_1, \mathbf{w}'_1) = (q_j, \mathbf{w}_j)$. Nous rappelons que pour toute transition $e \in E_0$ du SAVE à $3n$ compteurs S_0, pour tout $i \in [1..n]$, $D_e(2n+i) \geq 0$ (par construction de S_0). Nous prenons le plus grand r et le plus petit s tels que $1 \leq s \leq r \leq f$ et $D_{e_r}(2n+i) > 0$ et $D_{e_s}(2n+i) > 0$. Comme $\mathbf{D}_{e_1 \ldots e_f}(2n+i) > 0$ et en utilisant le rappel précédent, ces deux entiers r et s existent nécessairement. Nous avons alors dans $\mathbf{CG}(S_0, c'_0)$ le cycle $((q'_r, \mathbf{w}'_r) \xrightarrow{e_r} (q', \mathbf{w}') \xrightarrow{\sigma'_1} (q'_1.\mathbf{w}'_1) \xrightarrow{\sigma'_2} (q'_s.\mathbf{w}'_s) \xrightarrow{e_s} (q'', \mathbf{w}'') \xrightarrow{\sigma'_3} (q'_r, \mathbf{w}'_r))$, où $(q', \mathbf{w}'), (q'', \mathbf{w}'') \in \{(q'_1, \mathbf{w}'_1), \ldots, (q'_f, \mathbf{w}'_f)\}$ et $\sigma'_1, \sigma'_2, \sigma'_3 \in E_n^*$. Par définition de r et s, nous avons que $\mathbf{D}_{\sigma'_1}(2n+i) = \mathbf{D}_{\sigma'_2}(2n+i) = 0$. Comme $(q'_r, \mathbf{w}'_r), e_r, (q', \mathbf{w}'))$ appartient à un cycle, il appartient à un cycle élémentaire, et comme $\mathbf{D}_{e_r}(2n+i) > 0$, par hypothèse nous avons $\mathbf{w}'_r(i) \leq b$ et $\mathbf{w}'(i) \leq b$. Pour les mêmes raisons, nous avons aussi $\mathbf{w}'_s(i) \leq b$ and $\mathbf{w}''(i) \leq b$. Par définition de $\mathbf{w}'_1$, il existe $j \in [1..N]$ tel que $\mathbf{w}'_1 = \mathbf{w}_j$ et comme nous l'avons fait remarquer, nous avons $b < \mathbf{v}_j(i) \leq \mathbf{w}_j(i)$, par conséquent $b < \mathbf{w}'_1(i)$. Si nous résumons, nous avons dans $\mathbf{CG}(S_0, c'_0)$, un cycle $(q'_r, \mathbf{w}'_r) \xrightarrow{e_r} (q', \mathbf{w}') \xrightarrow{\sigma'_1} (q'_1.\mathbf{w}'_1) \xrightarrow{\sigma'_2} (q'_s.\mathbf{w}'_s) \xrightarrow{e_s} (q'', \mathbf{w}'') \xrightarrow{\sigma'_3} (q'_r, \mathbf{w}'_r)$ tel que :

(i) $\mathbf{w}'(i) \leq b$, $\mathbf{w}'_s(i) \leq b$, $b < \mathbf{w}'_1(i)$, et,

(ii) $D_{\sigma'_1}(2n+i) = D_{\sigma'_2}(2n+i) = 0$.

Nous rappelons que dans S_0 le $(2n+i)$-ème compteur compte les alternances réalisées au dessus de 0 par le i-ème compteur entre les phases de croissance et de décroissance. Nous en déduisons qu'il serait possible de construire une exécution de S_0 partant de c'_0 qui passent par une configuration dans laquelle le i-ème compteur est plus petite que b et ensuite par une configuration dans laquelle le i-ème compteur est strictement plus grand que b pour arriver de nouveau dans une configuration où le i-ème compteur est plus petite que b, sans que la valeur du $(2n+i)$-ème compteur ne change, ce qui constitue une contradiction. Par conséquent (S, c_0) est *reversal-b*-borné. □.

En d'autres termes, ce dernier lemme nous dit que le SAVE (S, c_0) est *reversal-b*-borné pour un $b \in \mathbb{N}$ si et seulement si pour tout $i \in [1..n]$, il n'y a pas de cycle élémentaire dans le graphe de couverture $\mathbf{CG}(S_0, c'_0)$ qui fait strictement croître le $(2n+i)$-ème compteur et dans lequel on trouve un noeud dont la i-ème composante est strictement plus grande que b ou égal à ω. En effet, en utilisant le lemme 2.41, si un tel circuit existait, nous pourrions construire une exécution du SAVE (S, c_0) qui ne respecte pas la définition de *reversal-b*-borné.

Ainsi, le lemme 2.51, nous fournit une condition nécessaire et suffisante sur le graphe de couverture $\mathbf{CG}(S_0, c'_0)$ pour vérifier si le SAVE initialisé (S, c_0) est *reversal*-borné. Il suffit en effet de tester pour chaque $i \in [1..n]$ que, dans chaque cycle élémentaire de $\mathbf{CG}(S_0, c'_0)$ faisant strictement croître la valeur du $2n+i$-ème compteur, il n'y a pas de noeud dont la i-ème composante vaut ω. Comme le graphe de couverture est fini, si cette condition est vérifiée, on trouvera un b tel que la condition du lemme 2.51 est vérifiée et donc (S, c_0) sera *reversal-b*-borné. Et si cela n'est pas vrai alors la condition

du lemme n'est vérifiée pour aucun b, et donc (S, c_0) ne peut pas être *reversal*-bornée. Notons de plus que c'est la finitude du graphe de couverture qui permet de pouvoir tester cette condition. Nous énonçons alors le résultat obtenu :

Théorème 2.52 *Vérifier si un SAVE est reversal-borné est un problème décidable.*

Malheureusement, l'algorithme de décision que nous proposons nécessite de construire le graphe de couverture d'un SAVE dans son intégralité et comme nous l'avons dit précédemment cette construction peut prendre un espace de taille non primitive récursive.

Exemple 2.53 *Si nous considérons le graphe de couverture* $\mathbf{CG}(S_0, c'_0)$ *de la figure 2.11 associé au SAVE* $(S, (q_1, 0))$ *de la figure 2.8, nous voyons que même si dans certains noeuds la composante associé au premier compteur vaut* ω*, ces noeuds n'appartiennent jamais à un cycle élémentaire faisant strictement croître les valeurs du* 3*-ème compteur, par conséquent nous en déduisons que le SAVE initialisé* $(S, (q_1, 0))$ *est reversal-borné. Nous nous apercevons en effet que pour une valeur de* $b = 1$*, les conditions du lemme 2.51 sont vérifiées, par conséquent ce SAVE est même reversal-*1*-borné.*

Les SAVE *reversal*-bornés constituent ainsi une nouvelle classe récursive de SAVE ayant un ensemble d'accessibilité semi-linéaire. Nous utilisons le mot récursif ici dans le sens que l'on peut décider si un SAVE appartient ou non à cette classe. Dans le survey sur les réseaux de Petri [EN94], les auteurs donnent d'autres classes de SAVE ayant un ensemble d'accessibilité semi-linéaire, comme par exemple les réseaux BPP [Esp97], les réseaux de Petri cycliques [AK77], les réseaux de Petri persistants [LR78, May81], les réseaux de Petri réguliers [VVN81], ou encore les SAVE de dimension 2 [HP79]. Nous ne donnons pas ici le détail de ces différentes classes, il semblerait cependant que ces classes soient incomparables avec les SAVE *reversal*-bornés, pour certaines cela est évident et pour d'autres il faudrait faire une étude un peu plus poussée.

Conclusion

Dans ce chapitre, nous avons étendu la définition des machines à compteurs *Ibarra-reversal*-bornées de la façon suivante : une machine à compteurs est k*-reversal-*b-bornée si dans toute exécution chaque compteur réalise au dessus de b au plus k alternances entre les phases de croissance et les phases de décroissance. Nous avons ensuite prouvé que ces nouvelles machines *reversal*-bornées avaient encore un ensemble d'accessibilité effectivement définissable dans Presburger et qu'elles étaient aplatissables. Nous avons également étudié la décidabilité de cette notion et vu que l'on pouvait décider si une machine à compteurs initialisée était *reversal*-bornée uniquement lorsque les deux paramètres k et b étaient donnés. Nous avons fini par étudier les SAVE *reversal*-bornés et nous avons prouvé que le problème de savoir si un SAVE initialisé est *reversal*-borné est un problème décidable. Ce résultat est d'autant plus intéressant que l'on ne sait pas si l'on peut décider si un SAVE est aplatissable ou non. En revanche, le problème de savoir si un SAVE a un ensemble d'accessibilité semi-linéaire serait décidable comme cela est indiqué dans le survey [EN94].

Chapitre 3

Des logiques pour la vérification de systèmes à compteurs

Après avoir présenté et étudié des problèmes d'accessibilité pour les systèmes à compteurs, nous nous attaquons dans ce chapitre à l'étude de problèmes de model-checking en considérant différentes logiques. Dans un premier temps nous nous intéressons à une extension de la logique arborescente CTL* , obtenue en ajoutant des formules de l'arithmétique de Presburger à l'ensemble des propositions atomiques. Dans un deuxième temps, nous considérons des logiques permettant d'exprimer des propriétés sur des mots de données, en particuliers la logique LTL avec registres et la logique du premier ordre sur les mots de données.

3.1 Model-checking avec une logique temporelle sur domaine concret

3.1.1 La logique FOCTL*(Pr)

Précédemment, dans le chapitre 1, les principaux problèmes de vérification que nous avons présentés étaient des problèmes dits d'accessibilité. Ils consistaient à savoir si une configuration ou un ensemble de configurations étaient accessibles à partir d'un ensemble de configurations initiales. Pour assurer le bon fonctionnement d'un système, il est parfois nécessaire d'avoir des informations plus complètes sur son comportement. En effet, on pourrait vouloir savoir comment se comportent les exécutions du système dans le temps. Voici quelques exemples de questions qui peuvent être soulevées lors de l'analyse d'un système :

– Le système va-t-il passer infiniment souvent par un ensemble donné de configurations ?
– Existe-t-il il une exécution du système qui va passer d'abord par une configuraion donnée et ensuite par une autre ?
– Existe-t-il une exécution du système qui ne passe jamais deux fois par la même configuration ?

Afin d'exprimer de telles questions de façon formelle, des logiques, appelées logiques temporelles, ont été introduites. Ainsi dans [Pnu77], Pnueli définit la logique temporelle linéaire LTL pour spécifier des propriétés sur une exécution particulière d'un système. La spécificité de cette logique sont les opérateurs temporels `X` (*next)* et `U` (*until*), le premier est utilisé pour dire que le système va arriver dans un état vérifiant une certaine propriété au pas suivant de l'exécution, et l'autre qu'une propriété est vérifiée jusqu'à ce qu'une autre propriété soit vérifiée. On voit ainsi apparaître les spécifications temporelles. Par la suite, dans [CE82], les auteurs présentèrent une autre logique temporelle appelée

logique temporelle arborescente qui permet d'exprimer les différentes possibilités d'évoluer qu'a une exécution à un moment donné. Ainsi, contrairement à LTL , où l'on fixe une exécution pour ensuite l'analyser et donc où il n'y a au plus qu'un état suivant, avec les logiques arborescentes, on peut considérer les différents états suivants. La logique introduite dans [CE82] s'appelle CTL , et de façon à pouvoir parler des différentes exécutions possibles à un instant donné, elle utilise des quantificateurs (existentiels et universels) sur les exécutions. Il s'avère que ces deux logiques LTL et CTL sont incomparables, une autre logique temporelle arborescente, appelée CTL* a donc été définie dans [EH83], et LTL et CTL sont des fragments de cette logique.

LTL , CTL et CTL* ont été introduites dans un premier temps pour résoudre des problèmes de model-checking dans des structures de Kripke, c'est-à-dire dans des automates dont les états sont associés à un nombre fini de variables propositionnelles. Dans le cas de LTL , le problème du model-checking consiste à savoir si étant donnés une structure de Kripke, une formule ϕ de LTL et un état initial de la structure, il existe une exécution (ou un chemin) partant de l'état initial qui satisfait la formule ϕ. Dans le cas de CTL et CTL* , le problème consiste à savoir si étant donnés un état initial de la structure de Kripke et une formule ϕ de CTL ou de CTL* , cet état satisfait la formule ϕ. Pour parler des états du système, on utilise donc des variables propositionnelles avec lesquelles chaque état est étiqueté. Dans la plupart des problèmes regardés, les logiques introduites alors ne permettent pas de parler de configurations d'un système à état infini mais seulement des états de contrôle. C'est par exemple le cas dans [BEM97] où les auteurs proposent de résoudre des problèmes de model-checking dans des automates à piles et les logiques temporelles considérées ne permettent pas de parler de l'état de la pile.

Pour palier à ce dernier point, dans [DFGD06], les auteurs introduisent une logique basée sur CTL* mais où les propositions atomiques ne sont plus seulement des variables propositionnelles mais des ensembles symboliques de configurations d'un système à compteurs définis grâce à des formules de Presburger. Ils appellent cette logique FOCTL*(Pr) . La logique proposée étend CTL* et donc en posant des restrictions sur les quantificateurs de chemin et sur les opérateurs temporels, il est facile d'obtenir une extension similaire pour la logique temporelle linéaire LTL ou de la logique temporelle arborescente CTL .

Nous introduisons maintenant la syntaxe de cette logique. Soit $X_0 = \{x_0, x_1, \ldots, x_n\}$ un ensemble de compteurs et VAR un ensemble de variables tels que $X \cap VAR = \emptyset$. Les formules de la logique FOCTLX*(Pr)[X_0] sont définies par la syntaxe suivante :

$$\Phi \ ::= \psi \mid \exists y.\Phi \mid \neg\Phi \mid \Phi \wedge \Phi \mid \mathtt{X}\Phi \mid \Phi\mathtt{U}\Phi \mid \mathtt{A}\Phi$$

où ψ est une formule de Presburger dans **Presb**$(X_0 \uplus \text{VAR})$ et $y \in$ VAR. Nous dirons qu'une formule Φ de FOCTLX*(Pr)[X_0] est une formule fermée si toutes les variables de VAR apparaissant dans Φ sont liées par un quantificateur. Nous utilisons les raccourcis habituels suivants pour $\Phi \in$ FOCTLX*(Pr)[X_0] :

– $\mathtt{F}\Phi$ équivaut à $true\mathtt{U}\Phi$,
– $\mathtt{G}\Phi$ équivaut à $\neg\mathtt{F}\neg\Phi$,
– $\mathtt{E}\Phi$ équivaut à $\neg\mathtt{A}\neg\Phi$.

Pour cette logique nous ne nous intéressons pas à des problèmes de satisfiabilité, mais directement à des problèmes de model checking comme nous allons le voir à la partie suivante.

Notons qu'il existe également d'autres extensions de logiques temporelles permettant de parler de configurations de systèmes ayant un nombre infini d'états, en particulier d'autres extensions utilisant également des relations arithmétiques sur les entiers sont présentées dans [Gas07].

3.1.2 Sémantique de FOCTL*(Pr) et problèmes de model-checking

Nous donnons dans cette section la sémantique de la logique FOCTLX*(Pr)[X_0] introduite précédemment. Soit $S = \langle Q, X, E\rangle$ un système à n compteurs et $TS(S) = \langle Q \times \mathbb{N}^n, E, \rightarrow\rangle$ le système de transitions qui lui est associé. Soient π une exécution finie de $TS(S)$ et $i \in \mathbb{N}$, nous notons :

- $\pi(i) \in Q \times \mathbb{N}^n$ la i-ème configuration de π,
- $\pi_{\leq i}$ la partie initiale de π jusqu'à la position i,
- $|\pi|$ la longueur de π.

Ainsi si $\pi = c_0 \stackrel{e_0}{\rightarrow} c_1 \stackrel{e_1}{\rightarrow} \dots c_{f-1} \stackrel{e_{f-1}}{\rightarrow} c_f$ avec $f \in \mathbb{N}$ est une exécution de $TS(S)$, nous avons $|\pi| = f + 1$ et pour tout $i \in [0..f]$, $\pi(i) = c_i$ et $\pi_{\leq i} = c_0 \stackrel{e_o}{\rightarrow} ... \stackrel{e_{i-1}}{\rightarrow} c_i$. Nous dirons de plus que deux exécutions $\pi = c_0 \stackrel{e_0}{\rightarrow} c_1 \stackrel{e_1}{\rightarrow} \dots c_{f-1} \stackrel{e_f}{\rightarrow} c_f$ et $\pi' = c'_0 \stackrel{e'_0}{\rightarrow} c'_1 \stackrel{e'_1}{\rightarrow} \dots c'_{f-1} \stackrel{e'_f}{\rightarrow} c'_f$ de $TS(S)$ sont équivalentes, noté $\pi \equiv \pi'$, si et seulement si $|\pi| = |\pi'|$ et pour tout $i \in [0..|\pi|]$, nous avons $c_i = c'_i$. La relation de satisfaction $\models$ que nous allons définir pour la logique FOCTLX*(Pr)[X_0] est paramétrée par un environnement ρ qui est en fait une fonction VAR $\rightarrow \mathbb{N}$ dont on se sert pour interpréter les variables de VAR qui peuvent apparaître dans les formules. Lorsque les notations seront explicites, nous omettrons volontairement cet environnement. Pour $i \in \mathbb{N}$ et une exécution π de $TS(S)$, la relation de satisfaction $\models$ pour les formules de FOCTLX*(Pr)[X_0] est définie par induction à la position i de l'exécution π de la façon suivante :

- $\pi, i \models_\rho \psi$ avec $\psi \in$ **Presb**$(X_0 \uplus \text{VAR})$ si et seulement si $(\pi(i), \rho) \models \psi$ dans l'arithmétique de Presburger,
- $\pi, i \models_\rho \exists y.\Phi$ si et seulement si il existe $m \in \mathbb{N}$ tel que $\pi, i \models_{\rho[y\mapsto m]} \Phi$ où $\rho[y \mapsto m]$ est l'environnement obtenu à partir de ρ en donnant à y la valeur m,
- $\pi, i \models \neg\Phi$ si et seulement si $\pi, i \not\models \Phi$,
- $\pi, i \models \Phi \wedge \Phi'$ si et seulement si $\pi, i \models \Phi$ et $\pi, i \models \Phi'$,
- $\pi, i \models \mathtt{X}\Phi$ si et seulement si $i < |\pi|$ et $\pi, i+1 \models \Phi$,
- $\pi, i \models \Phi \mathtt{U} \Phi'$ si et seulement si il existe $j \in \mathbb{N}$ tel que $i \leq j \leq |\pi| - 1$ et $\pi, j \models \Phi'$ et pour tout $k \in \mathbb{N}$ tel que $i \leq k < j$, $\pi, k \models \Phi$,
- $\pi, i \models \mathtt{A}\Phi$ si et seulement si pour toute exécution π' de $TS(S)$ telle que $\pi_{\leq i} \equiv \pi'_{\leq i}$, nous avons $\pi', i \models \Phi$.

Nous rappelons que de façon à manipuler les configurations d'un système à compteurs $S = \langle Q, X, E\rangle$, nous supposons que Q est le sous-ensemble de $\mathbb{N}$ égal à $[1..|Q|]$. Ainsi une configuration $(q, \mathbf{v}) \in Q \times \mathbb{N}^n$ peut-être vue comme un vecteur $\mathbf{w}$ de $\mathbb{N}^{n+1}$ où la composante $\mathbf{w}(0)$ vaut q. Si $X = \{x_1, \dots, x_n\}$, une formule de Presburger ψ dans **Presb**(X_0) où $X_0 = \{x_0, \dots, x_n\}$ permet alors de représenter un ensemble de configurations égal à $[\![\psi]\!]_{X_0}$, la variable x_0 servant à manipuler les états de contrôle de S.

Maintenant que la sémantique de FOCTLX*(Pr)[X_0] est donnée, nous pouvons définir les problèmes de model-checking auxquels nous nous intéressons dans le cadre de la vérification de systèmes à compteurs. De la même façon que pour les problèmes d'accessibilité introduits dans la section 1.3.1

du chapitre 1, où nous avons distingué le cas où une configuration initiale était donnée et celui où l'on donnait une configuration symbolique initiale (ie un ensemble de configurations initiales), nous définissons ici deux problèmes de model-checking. Tout d'abord le *problème de model-checking symbolique* :

Entrées : Un système à compteurs $S = \langle Q, X, E\rangle$, une configuration initiale $c_0 \in Q \times \mathbb{N}^X$ et une formule Φ de FOCTLX*(Pr)$[X_0]$.
Question : Est-il vrai que pour toutes les exécutions π de $TS(S)$ vérifiant $\pi(0) = c_0$, nous avons $\pi, 0 \models \Phi$?

Nous définissons maintenant un autre problème de telle sorte qu'au lieu de considérer une unique configuration initiale possible, nous prenons en compte une configuration symbolique initiale. Ceci nous donne le *problème de model-checking symbolique généralisé* défini de la façon suivante :

Entrées : Un système à compteurs $S = \langle Q, X, E\rangle$, une configuration symbolique initiale $(q_0, \phi_0) \in Q \times \mathbf{Presb}(X)$ et une formule Φ de FOCTLX*(Pr)$[X_0]$.
Question : Est-il vrai que pour toutes les exécutions π de $TS(S)$ telles que $\pi(0) = (q_0, \mathbf{v}_0)$ avec $\mathbf{v}_0 \models \phi_0$ nous avons $\pi, 0 \models \Phi$?

Le problème du model-checking symbolique est ainsi un cas particulier de model-checking symbolique généralisé.

Exemple 3.1 *Nous donnons quelques exemples de formules de* FOCTL*(Pr) *permettant d'exprimer des propriétés intéressantes sur un système à compteurs. Nous considérons un système à compteurs $S = \langle Q, X, E\rangle$, une configuration initiale c_0. Le problème de l'accessibilité d'un état de contrôle q est un problème de model-checking symbolique avec la formule $\mathtt{EF}(x_0 = q)$. De la même façon le problème d'accessibilité symbolique pour une configuration $(q, \phi) \in Q \times \mathbf{Presb}(X)$ est un problème de model-checking symbolique en utilisant la formule $\mathtt{EF}(x_0 = q \wedge \phi)$. On peut également exprimer d'autres propriétes comme par exemple le fait que le nombre de configurations accessibles dans (S, c_0) est fini, ceci grâce à la formule $\exists y.\mathtt{AG} \bigwedge_{i\in[1..n]} x_i \leq y$.*

3.1.3 Model-checking de systèmes à compteurs

Comme nous l'avons vu précédemment, le problème de l'accessibilité est un cas particulier de model-checking symbolique de formules de FOCTL*(Pr) . Par conséquent, le fait que le problème de l'arrêt soit indécidable dans les machines de Minsky déterministes à deux compteurs (cf. théorème 1.37) implique le résultat d'indécidabilité suivant :

Théorème 3.2 *Le problème de model-checking symbolique est indécidable pour les machines de Minsky déterministes à deux compteurs.*

Cependant, malgré ce résultat d'indécidabilité pour les machines de Minsky, comme nous l'avons vu dans la section 1.4, il existe des restrictions sur les systèmes à compteurs permettant d'obtenir la décidabilité de certains problèmes d'accessibilité. Ainsi le théorème 1.47 et son corollaire 1.48 nous indique que lorsque les systèmes considérés sont des systèmes à compteurs linéaires plats et à monoïde fini, les problèmes d'accessibilité énoncés dans la section 1.3.1 sont décidables. Les auteurs de [DFGD06] ont étendu ce résultat aux problèmes de model-checking.

Théorème 3.3 *[DFGD06] Le problème de model-checking symbolique généralisé est décidable pour les systèmes à compteurs linéaires plats et à monoïde fini.*

La preuve de ce théorème consiste à montrer que le problème de model-checking symbolique généralisé se réduit à un problème de satisfiabilité d'une formule de l'arithmétique de Presburger. Dans [DFGD06], les auteurs énoncent également d'autres résultats de décidabilité pour d'autres classes de systèmes en restreignant les formules de logique temporelle considérées et en appliquant des techniques similaires à celles utilisées pour les systèmes à compteurs aplatissables. Nous ne présentons pas ici ces autres résultats, mais notons seulement que le résultat du théorème précédent ne s'étend pas aux systèmes à compteurs aplatissables, car lorsque l'on "aplatit" un système, on modifie la forme des exécutions possibles, ainsi si l'on veut pouvoir étendre ce résultat, il faut raffiner la manière dont on "aplatit" un système, comme cela est suggéré dans [DFGD06].

3.1.4 Model-checking de machines à compteurs *reversal*-bornées

Dans le chapitre 2, nous avons présenté les machines à compteurs *reversal*-bornées pour lesquels les problèmes d'accessibilité d'un état de contrôle et d'une configuration sont décidables. Nous rappelons que dans ce dernier chapitre nous avons proposé une extension du modèle des machines dites *Ibarra-reversal*-bornées originellement introduites dans [Iba78]. Comme nous l'avons signalé les machines à compteurs *Ibarra-reversal*-bornées correspondent aux machines à compteurs *reversal*-0-bornées.

Dans [DIP01], les auteurs introduisent deux problèmes de vivacité pour les machines à compteurs et ils s'intéressent à la décidabilité de ces problèmes lorsque les machines à compteurs prises en compte sont *reversal*-0-bornées. Tout d'abord, nous présentons le *problème d'accessibilité symbolique répétée existentiel* :

Entrées : Un système à compteurs $S = \langle Q, X, E\rangle$, une configuration initiale $c_0 \in Q \times \mathbb{N}^X$ et une formule $\phi \in \mathbf{Presb}(X_0)$.
Question : Existe-t-il un ensemble infini de configurations $(c_i)_{i\in\mathbb{N}}$ tel que pour tout $i \in \mathbb{N}$, $c_i \rightarrow c_{i+1}$ dans $TS(S)$ et tel que l'ensemble $\{i \in \mathbb{N} \mid c_i \models \phi\}$ soit infini ?

Ce problème s'appelle problème d'accessibilité symbolique répétée existentiel car il consiste à savoir si il existe une exécution infinie dans le système de transitions associé à un système à compteurs dont les configurations vérifient infiniment souvent une formule de Presburger donnée. Nous avons ensuite la version universelle de ce problème qui consiste à savoir si toutes les exécutions infinies d'un système à compteurs initialisée vérifient une telle propriété. C'est ce que nous appelons le *problème d'accessibilité symbolique répétée universel* :

Entrées : Un système à compteurs $S = \langle Q, X, E\rangle$, une configuration initiale $c_0 \in Q \times \mathbb{N}^X$ et une formule $\phi \in \mathbf{Presb}(X_0)$.
Question : Est-ce que tout ensemble infini de configurations $(c_i)_{i\in\mathbb{N}}$ tel que pour tout $i \in \mathbb{N}$, $c_i \rightarrow c_{i+1}$ dans $TS(S)$ est tel que l'ensemble $\{i \in \mathbb{N} \mid c_i \models \phi\}$ est infini ?

Dans [DIP01], les auteurs obtiennent alors les deux résultats suivants :

Théorème 3.4 *[DIP01]*

1. *Le problème d'accessibilité symbolique répétée existentiel est décidable pour les machines à compteurs reversal-0-bornées.*

2. *Le problème d'accessibilité symbolique répétée universel est indécidable pour les machines à compteurs reversal-*0*-bornées.*

Remarquons que si nous considérons un système à compteurs $S = \langle Q, X, E\rangle$, une configuration initiale $c_0 \in Q \times \mathbb{N}^X$ et une formule $\phi \in$ **Presb**(x_0), le problème d'accessibilité symbolique répétée existentiel est équivalent au problème du model-checking symbolique avec la formule de FOCTLX*(Pr)$[X_0]$ EGFϕ. De la même façon, le problème d'accessibilité symbolique répétée universel est équivalent au problème du model-checking symbolique avec la formule de FOCTLX*(Pr)$[X_0]$ AGFϕ. Ceci nous permet de déduire le résultat d'indécidabilité suivant :

Théorème 3.5 *Le problème de model-checking symbolique est indécidable pour les machines à compteurs reversal-bornées.*

En ce qui concerne le résultat de décidabilité obtenu dans [DIP01], il peut quant à lui être étendu aux machines à compteurs intialisées *reversal*-bornées introduites dans le chapitre 2.

Théorème 3.6 *Le problème d'accessibilité symbolique répétée existentiel est décidable pour les machines à compteurs reversal-bornées.*

Preuve : Soit (S, c_0) une machine à n compteurs k-*reversal*-b-bornée avec $S = \langle Q.X, E\rangle$ et $\phi \in$ **Presb**(X_0) une formule de Presburger. Nous supposons que $X_0 = \{x_0, \ldots, x_n\}$. Nous considérons ensuite la machine à n compteurs (S', c'_0) construite à partir de (S, c_0) dans la section 2.2.2. Nous renommons les variables de ϕ pour obtenir une formule $\psi \in$ **Presb**(X'_0) de telle sorte que $[\![\phi]\!]_{X_0} = [\![\psi]\!]_{X'_0}$. En raisonnant de la même façon que pour la preuve du lemme 2.14, nous construisons la formule suivante :

$$\phi' = \exists x'_0 \ldots \exists x'_n . \psi \wedge \bigvee_{(q,\mathbf{u}) \in Q \times B_0^n} \big(x'_0 = (q, \mathbf{u}) \wedge x_0 = q \bigwedge_{i \in [1..n]} (x'_i = 0 \wedge x_i = \mathbf{u}(i)) \vee (x'_i > b \wedge x_i = x'_i)\big)$$

De la même façon que cela est énoncé dans le lemme 2.14, en utilisant les propriétés de la relation $\sim_b$, nous pouvons montrer que le problèmes d'accessibilité répétée symbolique existentiel est vérifié par (S, c_0) avec la formule de Presburger ϕ si et seulement si il est vérifié par (S', c'_0) avec la formule de Presburger ϕ'. Comme de plus la machine à compteurs (S', c'_0) est *reversal*-0-bornée (cf. lemme 2.17), le théorème 3.4 permet de conclure. □

Nous avons ainsi montré que certains problèmes de model-checking de formules de FOCTL*(Pr) étaient décidables lorsque l'on considère des systèmes à compteurs linéaires plats ou des machines à compteurs *reversal*-bornées. Cependant la logique FOCTL*(Pr) étant très expressive, les problèmes deviennent rapidement indécidables.

3.2 Model-checking avec des logiques sur des mots de données

3.2.1 Introduction

Dans cette section, nous allons de nouveau nous intéresser à des problèmes de model-checking en considérant cette fois-ci des logiques permettant d'exprimer des propriétés sur des mots de données. Une partie des résultats que nous présentons ici est issue de [DLS08].

Les mots de données sont des séquences pour lesquelles chaque position est étiquetée par une lettre d'un alphabet fini et par une autre "lettre" d'un alphabet infini (appelée la donnée). Ce simple modèle permet d'englober les mots temporisés acceptés par un automate temporisé pour lesquels la lettre appartient à l'alphabet de l'automate et la donnée est un réel positif correspondant à une valeur d'horloge [AD94]. L'extension aux arbres de cette notion est utilisée également pour modéliser des documents XML avec données comme par exemple dans [BDM$^+$06] et dans [JL07]. Pour accéder aux données présentes dans ces mots, des formalismes logiques ont été introduits pour les mots (ou arbres) de données ; une des caractéristiques de ces formalismes est qu'ils possèdent un mécanisme permettant d'enregistrer une valeur qui est ensuite comparer avec d'autres valeurs, c'est ainsi le cas dans [BMS$^+$06] et dans [DL06]. Comme nous allons le voir par la suite, cette caractéristique est très puissante et mène rapidement à des résultats d'indécidabilité. De plus, il existe de nombreuses autres logiques permettant de caractériser des mots de données et parfois le mécanisme permettant de parler des données est ajouté en complétant une logique temporelle déjà existante comme c'est le cas pour la logique temporelle avec passé et l'opérateur `Now` (en anglais "logic with forgettable past") introduite dans [LMS02] dont les formules permettent entre autre de parler du passé jusqu'à un certain moment. Dans [OW06, OW07], les auteurs étudient une autre logique temporelle linéaire, appelé logique temporelle métrique ("metric temporal logic" en anglais) permettant de décrire des mots temporisés. Cette dernière logique introduite dans [Koy90] est obtenue à partir de LTL en ajoutant des intervalles de temps. Dans [AH94], les auteurs étendent la logique temporelle linéaire en y ajoutant le quantificateur *freeze* qui permet d'enregistrer dans une variable une valeur correspondant à un temps, cette variable pouvant ensuite être utilisée dans la formule. Dans cette section, nous allons nous intéresser à deux logiques permettant de caractériser des mots de données :

1. la logique LTL avec registres, aussi connue sous le nom de LTL avec quantificateur *freeze* ,
2. la logique du premier ordre sur les mots de données.

Alors que la plupart des travaux existants concernant la logique LTL avec registres et la logique du premier ordre sur des mots de données étudient principalement des problèmes de satisfiabilité (cf [BMS$^+$06, DL06, DLN07]), nous étudions ici la capacité de ces logiques à spécifier des exécutions d'un modèle opérationnel à savoir les systèmes à compteurs. Mais comme ces logiques peuvent exprimer l'accessibilité d'un état de contrôle, l'indécidabilité de l'arrêt des machines de Minsky déterministes permet de déduire immédiatement un résultat d'indécidabililté si nous considérons des machines de Minsky à deux compteurs, c'est pourquoi nous allons nous restreindre à l'étude du model-checking de ces logiques dans le cadre particulier des machines de Minsky à un compteur.

3.2.2 Automates à un compteur

Le modèle que nous considérerons sont les automates à un compteur. Un automate à un compteur est une machine de Minsky à un compteur munie d'un état initial et d'un ensemble d'états acceptants. Plus formellement :

Definition 3.7 (Automate à un compteur) *Un automate à un compteur $\mathcal{A}$ est un quadruplet $\langle Q, \{x_1\}, E, q_0, F\rangle$ tel que :*
– *$\langle Q, \{x_1\}, E\rangle$ est une machine de Minsky,*
– *$q_I \in Q$ est un état initial,*
– *$F \subseteq Q$ est un ensemble d'états acceptants.*

Notation 3.8 *Nous noterons un automate à un compteur $\langle Q, E, q_0, F\rangle$ au lieu de $\langle Q, \{x_1\}, E, q_0, F\rangle$ et pour les transitions nous ne préciserons pas le compteur mais seulement les instructions* **inc**, **dec** *ou* **ifzero**.

Nous dirons qu'un automate à un compteur est déterministe si et seulement si la machine de Minsky sous-jacente l'est aussi. Si $\mathcal{A} = \langle Q, E, q_I, F\rangle$ est un automate à un compteur et $TS(\mathcal{A}) = \langle Q \times \mathbb{N}, E, \rightarrow\rangle$ est le système de transitions associé, nous appelons exécution finie de $TS(\mathcal{A})$ toute séquence finie $\pi = (q_0, n_0) \rightarrow (q_1, n_1) \dots (q_f, n_f)$ telle que $(q_0, n_0) = (q_I, 0)$. De plus π est une exécution finie acceptante si et seulement si $q_f \in F$. De la même façon, nous appelons exécution infinie de $TS(\mathcal{A})$ toute séquence infinie $\pi = (q_0, n_0) \rightarrow (q_1, n_1) \dots$ telle que $(q_0, n_0) = (q_I, 0)$. Nous dirons alors que π est une exécution infinie acceptante si et seulement si elle contient des états de contrôle acceptants de F infiniment souvent (condition d'acceptation de Büchi). Remarquons que nous choisissons de faire démarrer les exécutions avec une valeur égale à 0 pour le compteur, mais nous pourrions tout aussi bien prendre une autre valeur et ceci sans perte de généralités.

3.2.3 La logique LTL avec registres

Nous présentons ici la logique LTL avec registres aussi connue sous le nom de logique avec quantificateur *freeze* . Cette logique est obtenue à partir de la logique temporelle linéaire en ajoutant des registres. Elle a été introduite dans [DLN07] et différents problèmes de satisfiabilité ont été étudiées pour cette logique dans [DLN07, Laz06, DL06].

Soit Σ un alphabet fini. Les formules de la logique LTL avec registres sur Σ, notée $\text{LTL}^{\downarrow,\Sigma}$, sont définies par la grammaire suivante :

$$\phi \quad ::= \quad a \mid \uparrow_r \mid \neg\phi \mid \phi \wedge \phi \mid \phi \mathtt{U} \phi \mid \mathtt{X}\phi \mid \downarrow_r \phi$$

où $a \in \Sigma$ est une lettre de l'alphabet et $r \in \mathbb{N}^*$correspond à un numéro de registres. Si une occurrence du symbole $\uparrow_r$ est dans la portée d'un quantificateur *freeze* $\downarrow_r$ avec le même registre, on dit que cette occurrence est liée, sinon elle est dite libre. Nous dirons qu'une formule est fermée si elle ne contient aucune occurrence libre du symbole $\uparrow_r$ et ceci pour chaque registre r. Étant donné un entier naturel $n \in \mathbb{N}^*$, nous notons $\text{LTL}_n^{\downarrow,\Sigma}$ la restriction de $\text{LTL}^{\downarrow,\Sigma}$ aux formules dont les registres sont dans $[1..n]$.

Nous notons $\Sigma^{<\omega}$ l'ensemble des mots finis sur Σ et Σ^{ω} l'ensemble des mots infinis sur Σ. Les modèles de $\text{LTL}^{\downarrow,\Sigma}$ sont des mots de données. Un mot de données sur σ est un mot sur $\Sigma^{<\omega}$ ou sur Σ^{ω}, noté $\mathbf{str}(\sigma)$, associé à une relation d'équivalence $\sim^{\sigma}$ sur les indices de ce mot. C'est cette relation d'équivalence qui permet de parler des données du mot, on peut par exemple dire, comme nous le ferons par la suite, que deux positions sont équivalentes si elles sont associées à la même donnée. $|\sigma|$ représente la longueur du mot et $\sigma(i)$ la lettre à la i-ème position de $\mathbf{str}(\sigma)$ (avec $0 \leq i < |\sigma|$). Nous notons alors $\Sigma^{<\omega}(\sim)$ l'ensemble des mots de données finis et $\Sigma^{<\omega}(\sim)$ l'ensemble des mots de données infinis sur Σ.

Exemple 3.9 *[DL08] Un mot de données de longueur 3 sur l'alphabet $\{a, b\}$ peut être σ tel que $\mathbf{str}(\sigma) = aab$ et les classes d'équivalence de $\sim^{\sigma}$ sont $\{0, 2\}$ et $\{1\}$.*

Avant de donner la sémantique de $\text{LTL}^{\downarrow,\Sigma}$, nous introduisons une dernière notion. Nous appelons une valuation de registres v pour un mot de données σ une fonction partielle $v : \mathbb{N}^* \rightarrow \mathbb{N}$ des registres vers les indices du mot σ. Lorsque $v(r)$ n'est pas défini pour un registre $r \in \mathbb{N} \setminus \{0\}$ (c'est-à-dire

$r \notin \mathbf{dom}(v)$) alors la formule $\uparrow_r$ est interprétée comme fausse. Soient σ un mot de données dans $\Sigma^{<\omega}(\sim) \cup \Sigma^{\omega}(\sim)$, $i \in \mathbb{N}$ tel que $0 \leq i < |\sigma|$ et la relation de satisfaction $\models$ paramétrée par une valuation de registres v est définie par induction à la position i du mot de donnée σ de la façon suivante :

- $\sigma, i \models_v a$ si et seulement si $\sigma(i) = a$,
- $\sigma, i \models_v \uparrow_r$ si et seulement si $r \in \mathbf{dom}(v)$ et $v(r) \sim^{\sigma} i$,
- $\sigma, i \models_v \neg\phi$ si et seulement si $\sigma, i \not\models_v \neg\phi$,
- $\sigma, i \models_v \phi \wedge \phi'$ si et seulement si $\sigma, i \models_v \phi$ et $\sigma, i \models_v \phi'$,
- $\sigma, i \models_v \mathtt{X}\phi$ si et seulement si $i + 1 < |\sigma|$ et $\sigma, i + 1 \models_v \phi$,
- $\sigma, i \models_v \phi \mathtt{U} \phi'$ si et seulement si il existe $j \in \mathbb{N}$ tel que $i \leq j < |\sigma|$ et $\sigma, j \models_v \phi'$ et pour tout $k \in \mathbb{N}$ tel que $i \leq k < j$, $\sigma, k \models_v \phi$,
- $\sigma, i \models_v \downarrow_r \phi$ si et seulement si $\sigma, i \models_{v[r \mapsto i]} \phi$ où $v[r \mapsto i]$ est la valuation de registres obtenue à partir de v en associant au registre r la valeur i.

Par la suite, nous ne noterons plus la valuation de registres v dans la relation de satisfaction $\models_v$ lorsque nous considérerons des formule fermées de $\mathrm{LTL}^{\downarrow,\Sigma}$.
Comme nous l'avons dit précédemment, dans [DL06] les auteurs ont étudié des problèmes de satisfiabilité pour la logique $\mathrm{LTL}^{\downarrow,\Sigma}$. Nous rappelons certains des résultats qu'ils ont obtenus. Nous commençons par définir deux problèmes de satisfiabilité pour $\mathrm{LTL}^{\downarrow,\Sigma}$. Le *problème de satisfiabilité fini* pour LTL avec registres est défini de la façon suivante :

Entrées : Un alphabet fini Σ et une formule fermée ϕ de $\mathrm{LTL}^{\downarrow,\Sigma}$.
Question : Existe-t-il un mot de données σ dans $\Sigma^{<\omega}(\sim)$ tel que $\sigma, 0 \models \phi$?

Le deuxième problème de satisfiabilité que nous introduisons est presque le même que celui introduit ci-dessus excepté qu'il prend en compte des mots de données infinis. Ainsi, nous présentons le *problème de satisfiabilité infini* pour LTL avec registres :

Entrées : Un alphabet fini Σ et une formule fermée ϕ de $\mathrm{LTL}^{\downarrow,\Sigma}$.
Question : Existe-t-il un mot de données σ dans $\Sigma^{\omega}(\sim)$ tel que $\sigma, 0 \models \phi$?

Les résultats obtenus concernant ces deux problèmes s'expriment ainsi :

Théorème 3.10 *[DL06, DLN07]*

1. *La restriction à des formules n'ayant qu'un seul registre du problème de satisfiabilité fini pour* $\mathrm{LTL}^{\downarrow}$ *est décidable avec une complexité non primitive récursive.*
2. *La restriction à des formules n'ayant que deux registres du problème de satisfiabilité fini pour* $\mathrm{LTL}^{\downarrow}$ *est indécidable.*
3. *La restriction à des formules n'ayant qu'un seul registre du problème de satisfiabilité infini pour* $\mathrm{LTL}^{\downarrow}$ *est indécidable.*

Nous voyons ainsi que la logique LTL avec registres est très expressive car les problèmes de satisfiabilité présentés précédemment sont indécidables sauf dans le cas particulier des mots finis avec des formules utilisant un unique registre et même dans ce cas la complexité est très élevée.

Exemple 3.11 *[DL08] Nous considérons la formule ϕ de* $\mathrm{LTL}_1^{\downarrow}$ *sur l'alphabet $\{a, b\}$ telle que :*

$$\phi = \mathtt{G}\big(a \Rightarrow \downarrow_1 \mathtt{X}\big((\mathtt{G}(a \Rightarrow \neg \uparrow_1)) \wedge (\mathtt{F}(b \wedge \uparrow_1))\big)\big)$$

Cette formule exprime qu'il n'y a pas deux positions avec un a qui appartiennent à la même classe d'équivalence et que de plus chaque lettre a est suivie par une lettre b qui appartient à la même classe d'équivalence. Ainsi le mot donné à l'exemple 3.9 ne satisfait pas cette formule en position 0.

3.2.4 La logique du premier ordre sur des mots de données

Nous présentons maintenant la logique du premier ordre sur les mots de données introduites dans [BMS$^+$06]. Dans cette logique du premier ordre, les variables représentent des positions de données dans le mot, et il est également possible de tester la lettre à une position du mot et si deux positions appartiennent à la même classe d'équivalence. Soit Σ un alphabet fini, les formules de la logique du premier ordre sur les mots de données dans Σ, notée FO$^\Sigma(\sim, <, +1)$, sont définies par la grammaire suivante :

$$\phi \quad ::= \quad a(x) \mid x \sim y \mid x < y \mid x = y + 1 \mid \neg\phi \mid \phi \wedge \phi \mid \exists x.\phi$$

où $a \in \Sigma$ et x, y appartiennent à un ensemble dénombrable de variables. Par la suite, nous écrirons FO$(\sim, <, +1)$ lorsque nous ferons référence à la logique FO$^\Sigma(\sim, <, +1)$ pour un alphabet non spécifié et nous noterons FO$(<, +1)$ la restriction de FO$(\sim, <, +1)$ aux formules n'utilisant pas de proposition atomique de la forme $x \sim y$. De la même façon que pour les formules de l'arithmétique de Presburger, nous dirons qu'une variable x est libre dans une formule de FO$(\sim, <, +1)$ si elle n'est pas sous la portée d'un quantificateur $\exists$. Enfin, nous dirons qu'une formule de FO$(\sim, <, +1)$ est fermée si elle ne contient pas de variable libre.

Soit Σ un alphabet fini et VAR un ensemble dénombrable de variables dont nous nous servons dans les formules de FO$^\Sigma(\sim, <, +1)$ considérées. Nous allons donner la sémantique de FO$^\Sigma(\sim, <, +1)$, mais avant nous présentons une notion qui nous sera utile. Une valuation de variables u pour un mot de données σ est une fonction partielle $u : \text{VAR} \to \{0, \ldots, |\sigma| - 1\}$ allant de l'ensemble des variables vers les indices de σ. Soit σ un mot de données dans $\Sigma^{<\omega}(\sim) \cup \Sigma^{\omega}(\sim)$, la relation de satisfaction $\models$ paramétrée par une valuation de variables u est alors définie par induction de la façon suivante :

- $\sigma \models_u a(x)$ si et seulement si $x \in \mathbf{dom}(u)$ et $\sigma(u(x))$,
- $\sigma \models_u x \sim y$ si et seulement si $x, y \in \mathbf{dom}(u)$ et $u(x) \sim^\sigma u(y)$,
- $\sigma \models_u x < y$ si et seulement si $x, y \in \mathbf{dom}(u)$ et $u(x) < u(y)$,
- $\sigma \models_u x = y + 1$ si et seulement si $x, y \in \mathbf{dom}(u)$ et $u(x) = u(y) + 1$,
- $\sigma \models_u \neg\phi$ si et seulement si $\sigma \not\models_u \phi$,
- $\sigma \models_u \phi \wedge \phi'$ si et seulement si $\sigma \models_u \phi$ et $\sigma \models_u \phi'$,
- $\sigma \models_u \exists x.\phi$ si et seulement si il existe $i \in \mathbb{N}$ tel que $0 \leq i < |\sigma|$ et $\sigma \models_{u[x \mapsto i]}$ où $u[x \mapsto i]$ est la valuation de variables obtenue à partir de u en associant à la variable x la valeur i.

Comme pour LTL$^\downarrow$, nous ne noterons plus la valuation de variables u dans la relation de satisfaction $\models_u$ lorsque nous considérerons des formules fermées de FO$(\sim, <, +1)$.

Dans [BMS$^+$06], les auteurs ont également étudié les problèmes de satisfiabilité pour cette logique du premier ordre. De la même façon que pour la logique LTL avec registres, nous faisons la distinction entre le problème de satisfiabilité fini et le problème de satisfiabilité infini. Le *problème de satisfiabilité fini* pour FO$(\sim, <, +1)$ est défini ainsi :

Entrées : Un alphabet fini Σ et une formule fermée ϕ de FO$^\Sigma(\sim, <, +1)$.

Question : Existe-t-il un mot de données σ dans $\Sigma^{<\omega}(\sim)$ tel que $\sigma \models \phi$?

Quant au *problème de satisfiabilité infini*, il s'agit du même problème en considérant des mots de données infinis :

Entrées : Un alphabet fini Σ et une formule fermée ϕ de FO$^{\Sigma}(\sim, <, +1)$.
Question : Existe-t-il un mot de données σ dans $\Sigma^{\omega}(\sim)$ tel que $\sigma \models \phi$?

Les résultats obtenus concernant ces deux problèmes peuvent alors être exprimés ainsi :

Théorème 3.12 *[BMS$^+$06]*

1. *Les problèmes de satisfiabilité fini et infini pour des formules de* FO*$(\sim, <, +1)$ utilisant au plus* 2 *variables sont décidables.*
2. *Les problèmes de satisfiabilité fini et infini pour des formules de* FO*$(\sim, <, +1)$ utilisant* 3 *variables sont indécidables.*

Ainsi la logique du premier ordre sur les mots de données est elle aussi, tout comme la logique LTL avec registres, très puissante. Par la suite, nous noterons FO2$(\sim, <, +1)$ la logique obtenue en restreignant FO$(\sim, <, +1)$ aux formules utilisant au plus deux variables.

Exemple 3.13 *[BMS$^+$06] Dans cet exemple, nous montrons comment exprimer avec une formule de la logique sur les mots de données qu'un mot contient le même nombre de* a *que de* b*. Nous considérons l'alphabet à deux lettres* $\{a, b\}$*. Nous construisons une formule* ϕ *de* FO$^{\{a,b\}}(\sim, <, +1)$ *de sorte que tout mot de donnée fini* σ*, satisfiant* ϕ *sera tel que le mot* $\mathbf{str}(\sigma)$ *aura le même nombre de* a *et de* b*.*

– *La formule* ϕ_a *dit qu'il n'existe pas deux positions avec la lettre* a *appartenant à la même classe d'équivalence :*

$$\phi_a = \forall x.\forall y.\big(x \neq y \wedge a(x) \wedge a(y)\big) \Rightarrow x \not\sim y$$

De la même façon, nous définissons ϕ_b*.*

– *La formule* $\phi_{a,b}$ *dit que chaque classe contenant une position avec un* a *contient également une position avec un* b *:*

$$\phi_{a,b} = \forall x.\exists y.\big(a(x) \Rightarrow (b(y) \wedge x \sim y)\big)$$

De la même façon, nous construisons $\phi_{b,a}$*.*

Ainsi, la formule ϕ *désirée est égale à* $\phi_a \wedge \phi_b \wedge \phi_{a,b} \wedge \phi_{b,a}$*.*

De la même façon qu'il existe une traduction de la logique linéaire temporelle LTL vers la logique du premier ordre, nous pouvons établir le résultat qui suit :

Proposition 3.14 *Soient* Σ *un alphabet fini et* $n \in \mathbb{N}^*$*. Étant donnée une formule fermée* ϕ *de* LTL$_n^{\downarrow,\Sigma}$*, il existe une formule* ψ *de* FO$^{\Sigma}(\sim, <, +1)$ *qui peut être calculée en temps linéaire dans la taille de* ϕ *et telle que :*

(i) ψ *utilise au plus* $n + 3$ *variables, et,*

(ii) ψ *a une unique variable libre (disons* y_0 *par exemple), et,*

(iii) pour tout mot de données σ*, toute valuation de registres* v *et tout entier* $i \in \mathbb{N}$*,* $\sigma, i \models_v \phi$ *si et seulement si* $\sigma \models_u \psi$ *où* u *est telle que pour tout registre* $r \in [1..n]$*, il y a une variable correspondante* x_r *vérifiant* $v(r) = u(x_r)$ *et de plus* $i = u(y_0)$*.*

Preuve : Nous construisons une fonction de traduction T qui prend comme arguments une formule de $\text{LTL}_n^{\downarrow,\Sigma}$ et une variable, et qui construit une formule de $\text{FO}^{\Sigma}(\sim,<,+1)$. Intuitivement la variable donnée en argument sert à indiquer la position courante. Nous nous servons donc des variables $x_1,\ldots,x_n$ pour caractériser les registres et en plus de trois variables auxiliaires y_0, y_1 et y_2. Dans la suite, nous écrirons y pour caractériser indifféremment y_0, y_1 ou y_2 et y_{i+1} sera un raccourci pour $y_{(i+1)mod(3)}$, de même y_{i+2} sera un raccourci pour $y_{(i+2)mod(3)}$. T est alors définie par induction de la façon suivante :

- $T(a,y) = a(y)$ (avec $a \in \Sigma$),
- $T(\uparrow_r, y) = y \sim x_r$,
- $T(\neg\phi, y) = \neg T(\phi, y)$,
- $T(\phi \wedge \phi', y) = T(\phi, y) \wedge T(\phi', y)$,
- $T(\mathtt{X}\phi, y_i) = \exists y_{i+1}.y_{i+1} = y_i + 1 \wedge T(\phi, y_{i+1})$,
- $T(\phi \mathtt{U} \phi', y_i) = \exists y_{i+1}.y_i \leq y_{i+1} \wedge T(\phi', y_{i+1}) \wedge \forall y_{i+2}.y_i \leq y_{i+2} < y_{i+1} \Rightarrow T(\phi, y_{i+2})$,
- $T(\downarrow_r \phi, y) = \exists x_r.y = x_r \wedge T(\phi, y)$.

Par construction si ϕ est une formule de $\text{LTL}_n^{\downarrow,\Sigma}$ et y_1 est une variable choisie pour représenter la position courante dans le mot, alors la formule $T(\phi, y_0)$ vérifie les 3 conditions énoncées par la proposition. Notons que pour le premier point, nous nous servons pleinement du fait que nous pouvons recycler les variables. Plus de détails sur le recyclage de variables sont disponibles dans [Gab81]. □

3.2.5 Différents problèmes de model-checking

Comme nous l'avons vu précédemment, les principaux problèmes étudiés pour la logique LTL avec registres et la logique du premier ordre sur les mots de données sont des problèmes de satisfiabilité. Nous présentons dans cette section les problèmes qui vont plus particulièrement nous intéresser par la suite. Il s'agit de voir si il est possible de résoudre des problèmes de model-checking de formules appartenant à ces deux logiques sur des automates à un compteur. Il est à noter que les automates à un compteur sont des systèmes à compteurs très simples pour lesquels de nombreux problèmes sont décidables (cf. théorème 1.43), mais d'un autre côté les logiques présentées permettent de spécifier des propriétés assez riches et la plupart des problèmes de satisfiabilité sont indécidables pour ces logiques.

Avant de définir les différents problèmes de model-checking, nous explicitons comment un automate à un compteur peut générer un mot de données, car les logiques considérées permettent d'exprimer des propriétés sur des mots de données. Soit $\mathcal{A} = \langle Q, E, q_I, F\rangle$ un automate à un compteur. Une exécution acceptante finie (respectivement infinie) de $TS(\mathcal{A})$ peut-être vue comme un mot de données fini (respectivement infini) sur l'alphabet fini Q des états de contrôle. Soit π une exécution finie ou infinie de $TS(\mathcal{A})$, pour tout $i \in \mathbb{N}$ tel que $i < |\pi|$, nous notons $(q_i, n_i) \in Q \times \mathbb{N}$ la i-ème configuration de π. Nous définissons la relation d'équivalence $\sim^\pi$ de la façon suivante, pour tout $i, j \in \mathbb{N}$ telle que $i < |\pi|$ et $j < |\pi|$, nous avons $i \sim^\pi j$ si et seulement si $n_i = n_j$, c'est-à-dire si et seulement si les valeurs de compteurs à la position i et à la position j sont égales. Le mot de données π est donc tel que $\mathbf{str}(\pi) = q_0 q_1 \ldots$ et est associé à la relation $\sim^\pi$.

3.2.5.1 Problèmes de model-checking pour LTL avec registres

Le *problème de model-checking fini existentiel de* LTL *avec registres sur des automates à un compteur*, noté $\text{MC(LTL)}^{<\omega}$, est défini ainsi :

Entrées : Un automate à un compteur $\mathcal{A} = \langle Q, E, q_I, F\rangle$ et une formule fermée ϕ de LTL$^{\downarrow,Q}$.
Question : Existe-t-il une exécution finie acceptante π de $TS(\mathcal{A})$ telle que $\pi, 0 \models \phi$?

Dans le cas d'une réponse positive à la question, nous écrirons $\mathcal{A} \models^{<\omega} \phi$. De la même façon, nous définissons le *problème de model-checking infini existentiel de* LTL *avec registres sur des automates à un compteur*, noté MC(LTL)$^{\omega}$:

Entrées : Un automate à un compteur $\mathcal{A} = \langle Q, E, q_I, F\rangle$ et une formule fermée ϕ de LTL$^{\downarrow,Q}$.
Question : Existe-t-il une exécution infinie acceptante π de $TS(\mathcal{A})$ telle que $\pi, 0 \models \phi$?

Dans le cas d'une réponse positive à la question, nous noterons $\mathcal{A} \models^{\omega} \phi$. Remarquons que ces deux problèmes de model-checking peuvent être vus comme des variantes de problèmes de satisfiabilité pour lesquels les mots de données satisfaisant les formules appartiennent à la classe des mots de données obtenus à partir d'exécutions acceptantes d'automates à un compteur. Notons de plus, que nous aurions pu définir ces problèmes de model-checking sous forme universelle en exigeant que toutes les exécutions acceptantes, et non plus une seule, satisfassent la formule donnée, le cas universel se réduisant au cas existentiel et réciproquement. De plus, pour $\alpha \in \{\omega, < \omega\}$ et $n \in \mathbb{N}^*$, nous notons MC(LTL)$^{\alpha}_n$ la restriction de MC(LTL)$^{\alpha}$ aux formules dans LTL$^{\downarrow,Q}_n$, c'est-à-dire aux formules utilisant au plus n registres. De plus, il peut être intéressant de pouvoir spécifier uniquement les valeurs prises par le compteur, les états de contrôle étant vus comme des informations internes du système. Nous appelons PureMC(LTL)$^{\alpha}_n$ la restriction de MC(LTL)$^{\alpha}_n$ aux formule dans LTL$^{\downarrow,\emptyset}_n$, c'est-à-dire aux formules dont les propositions atomiques sont uniquement de la forme $\uparrow_r$.

3.2.5.2 Problèmes de model-checking pour la logique du premier ordre sur les mots de donées

De la méme façon, nous définissons les problèmes de model-checking dans le cas de la logique du premier ordre sur les mots de données. Tout d'abord le *problème de model-checking fini existentiel de* FO$(\sim, <, +1)$ *sur des automates à un compteur*, noté MC(FO)$^{<\omega}$:

Entrées : Un automate à un compteur $\mathcal{A} = \langle Q, E, q_I, F\rangle$ et une formule fermée ϕ de FO$^{Q}(\sim, <, +1)$.
Question : Existe-t-il une exécution finie acceptante π de $TS(\mathcal{A})$ telle que $\pi \models \phi$?

Comme nous le faisons dans le cas de la logique LTL avec registres, dans le cas d'une réponse positive à cette question, nous écrirons $\mathcal{A} \models^{<\omega} \phi$. La version "infinie" de ce problème, c'est-à-dire le *problème de model-checking infini existentiel de* FO$(\sim, <, +1)$ *sur des automates à un compteur*, noté MC(FO)$^{\omega}$, est quant à elle donnée par :

Entrées : Un automate à un compteur $\mathcal{A} = \langle Q, E, q_I, F\rangle$ et une formule fermée ϕ de FO$^{Q}(\sim, <, +1)$.
Question : Existe-t-il une exécution infinie acceptante π de $TS(\mathcal{A})$ telle que $\pi \models \phi$?

Nous noterons $\mathcal{A} \models^{\omega} \phi$ dans le cas d'une réponse positive au problème. Comme nous l'avons fait pour la logique LTL avec registres, nous définissons quelques restrictions de ces problèmes. Ainsi, pour $\alpha \in \{< \omega, \omega\}$ et $n \in \mathbb{N}^*$, MC(FO)$^{\alpha}_n$ représente la restriction du problème MC(FO)$^{\alpha}$ aux formules utilisant au plus n variables et PureMC(FO)$^{\alpha}_n$ est le problème obtenu à partir de MC(FO)$^{\alpha}_n$ pour lequel les formules de FO$^{Q}(\sim, <, +1)$ considérées n'utilisent pas de proposition atomique de la forme $q(x)$ (avec $q \in Q$).

3.2.5.3 Model-checking de la logique du premier ordre sur des chemins infinis périodiques

Nous présentons un dernier problème de model-checking ici qui nous servira par la suite. Il s'agit du *problème de model-checking pour un chemin infini périodique* introduit dans [MS03] et qui s'énonce de la façon suivante :

Entrées : Un alphabet fini Σ, deux mots finis s et t dans Σ^* et une formule fermée ϕ de $\mathrm{FO}^{\Sigma}(<,+1)$.
Question : Est-ce que $s.t^\omega \models \phi$?

Par $s.t^\omega$, nous notons le mot infini qui a cette forme $s.t.t.t\ldots$. Ce problème ne fait pas intervenir de mots de données mais seulement des mots infinis. Il s'avère que ce problème est décidable, comme nous l'indique le résultat suivant :

Théorème 3.15 *[MS03] Le problème de model-checking pour un chemin infini périodique est dans* PSPACE.

D'après [MS03], nous savons même qu'étant donnés un alphabet fini Σ, deux mots $s,t \in \Sigma^*$ et une formule fermée ϕ de $\mathrm{FO}^{\Sigma}(<,+1)$, décider si $s.t^\omega \models \phi$ peut être fait en espace $\mathcal{O}\big((|s|+|t|)\times|\phi|^2\big)$.

3.2.5.4 Lemmes de purification

Nous montrons maintenant que pour les problèmes de model-checking présentés précédemment il est possible de ne pas considérer les propositions atomiques parlant des états de contrôle en réduisant les problèmes de model-checking à leur version "pure".

Lemme 3.16 *Soient $\mathcal{A} = \langle Q, E, q_I, F\rangle$ un automate à un compteur, $n \in \mathbb{N}^*$ et ϕ une formule de $\mathrm{LTL}_n^{\downarrow,Q}$. Il est possible de calculer en espace logarithmique en $|\mathcal{A}|+|\phi|$ un automate à un compteur $\mathcal{A}_P$ et une formule ϕ_P de $\mathrm{LTL}^{\downarrow,\emptyset}_{\mathbf{Max}(n,1)}$ tels que :*

– *$\mathcal{A} \models^{<\omega} \phi$ si et seulement si $\mathcal{A}_P \models^{<\omega} \phi_P$, et,*
– *$\mathcal{A} \models^{\omega} \phi$ si et seulement si $\mathcal{A}_P \models^{\omega} \phi_P$.*

De plus, si $\mathcal{A}$ est déterministe $\mathcal{A}_P$ l'est aussi.

Preuve : L'idée principale de la preuve consiste à identifier les états de contrôle de l'automate $\mathcal{A}$ avec des motifs caractérisables par la logique LTL avec registres. Soit $\mathcal{A} = \langle Q, E, q_I, F\rangle$ un automate à un compteur avec $Q = \{q_1,\ldots,q_m\}$. Nous construisons un automate à un compteur $\mathcal{A}_P = \langle Q_P, E_P, q_I, F_P\rangle$. Nous posons $Q_p = Q \uplus Q'$ avec :

$$\begin{aligned} Q' \;=\; & \{q_i^1, q_i^2, q_i^3, q_i^4, q_i^5, q_{i,F} \mid i \in [1..m]\} \\ & \cup \{q'_{i,j}, q_{i,j} \mid i \in [1..m] \text{ et } j \in [1..m+1] \text{ et } j \neq i\} \\ & \cup \{q^0_{i,i}, q^1_{i,i}, q_{i,i}, q^2_{i,i} \mid i \in [1..m]\} \end{aligned}$$

La figure 3.1 nous montre ensuite la séquence de transitions de E_P qui est associée à chaque état q_i de Q, nous avons de plus $(q_i,\delta,q_j) \in E$ avec $i,j \in [1..n]$ et $a \in \{\mathbf{inc},\mathbf{dec},\mathbf{ifzero}\}$ si et seulement si $(q_{i,F},a,q_j) \in E_P$. La séquence associée à chaque q_i est une séquence de $m+2$ "pics", et parmi ces pics le premier est l'unique de hauteur 3 et le $i+1$-ème est l'unique pic de hauteur 2, tous les autres étant de hauteur 1. Remarquons que cette séquence est de longueur fixe et qu'elle comporte exactement $9+2(m+1)$ états de contrôle. Nous supposons de plus que l'ensemble des états acceptants

$F_P = \{q_{i,F} \mid i \in [1..m] \text{ et } q_i \in F\}$. Afin de détecter le premier "pic" de hauteur 3 (c'est-à-dire celui réalisant trois incréments), nous construisons les formules de $\text{LTL}_1^{\downarrow,\emptyset}$ suivantes :

- $\phi_{\neg 3/7}$ qui exprime le fait que "parmi les 7 prochaines valeurs de compteurs (en incluant la valeur courante), il n'y a pas 3 valeurs qui sont égales", et,
- $\phi_{0\sim 6}$ qui exprime le fait que "la valeur courante du compteur est égale à la valeur du compteur à la 6-ème position suivante"

Formellement ces deux formules s'écrivent ainsi :

$$\begin{array}{rcl}
\phi_{\neg 3/7} & = & \neg\big(\downarrow_1 \big(\bigvee_{i,j\in[1..6]\wedge i\neq j}(\mathtt{X}^i \uparrow_1 \wedge \mathtt{X}^j \uparrow_1)\big) \\
& & \vee \mathtt{X}\downarrow_1 \big(\bigvee_{i,j\in[1..5]\wedge i\neq j}(\mathtt{X}^i \uparrow_1 \wedge \mathtt{X}^j \uparrow_1)\big) \\
& & \vee \mathtt{X}^2\downarrow_1 \big(\bigvee_{i,j\in[1..4]\wedge i\neq j}(\mathtt{X}^i \uparrow_1 \wedge \mathtt{X}^j \uparrow_1)\big) \\
& & \vee \mathtt{X}^3\downarrow_1 \big(\bigvee_{i,j\in[1..3]\wedge i\neq j}(\mathtt{X}^i \uparrow_1 \wedge \mathtt{X}^j \uparrow_1)\big) \\
& & \vee \mathtt{X}^4\downarrow_1 \big(\bigvee_{i,j\in[1..2]\wedge i\neq j}(\mathtt{X}^i \uparrow_1 \wedge \mathtt{X}^j \uparrow_1)\big)\big) \\
\\
\phi_{0\sim 6} & = & \downarrow_1 (\mathtt{X}^6 \uparrow_1)
\end{array}$$

Nous notons LOC la formule égale à $\phi_{\neg 3/7} \wedge \phi_{0\sim 6}$. Soient π une exécution de $TS(\mathcal{A}_P)$ et j un entier tel que $0 \leq j \leq |\pi| - 8 - 2(m+1)$, nous montrons que $\pi, j \models LOC$ si et seulement si $\pi(j) = (q, v)$ avec $q \in Q$. Par construction, si $\pi(j) = (q, v)$ avec $q \in Q$ alors $\pi, j \models LOC$ en effet, dans le premier pic de hauteur 3 qui contient 7 états, on est sûr de ne jamais rencontrer 3 fois la même valeur de compteur et de plus la valeur de compteurs 6 "coups" plus loin est bien égale à la valeur courante. Il nous faut maintenant montrer que si $\pi, j \models LOC$ alors $\pi(j) = (q, v)$ avec $q \in Q$. Supposons que $\pi(j) = (q, v)$ et que $\pi, j \models LOC$. Nous procédons par une étude de cas :

1. si q est égal à q_i^2 avec $i \in [2..m]$, on a $\pi, j \not\models \phi_{0\sim 6}$,
2. si q est égal à q_1^2, on a $\pi, j \not\models \phi_{\neg 3/7}$, en effet la valeur de compteur en q_1^2, celle en q_1^4 et celle en $q_{1,1}$ sont égales,
3. si q est égal à q_i^3 avec $i \in [1..m]$, on a $\pi, j \not\models \phi_{0\sim 6}$,
4. si q est égal à q_i^4 avec $i \in [1..m] \setminus \{2\}$, on a $\pi, j \not\models \phi_{0\sim 6}$,
5. si q est égal à q_2^4, on a $\pi, j \not\models \phi_{\neg 3/7}$, en effet la valeur de compteur en q_2^5, celle en $q_{2,1}$ et celle en $q_{2,2}^1$ sont égales,
6. si q est égal à $q_{i,i}$ avec $i \in [1..m-1]$, on a $\pi, j \not\models \phi_{0\sim 6}$,
7. si q est égal à $q_{m,m}$, on a $\pi, j \not\models \phi_{\neg 3/7} \wedge \phi_{0\sim 6}$, en effet si $\pi, j \models \phi_{0\sim 6}$, cela signifie qu'il y a nécessairement une transition $(q_{i,F}, \textbf{inc}, q_k)$ qui est franchie, ainsi la valeur en q_k^1 , c'est-à-dire "6" coups plus loin, est la même que celle en $q_{m,m}$ mais dans ce cas la valeurs en $q_{m,m}^2$, celle en $q_{m,m+1}$ et celle en q_k sont égales, et donc $\pi, j \not\models \phi_{\neg 3/7}$,
8. si q est égal à q_i^1 avec $i \in [1..m]$, on a $\pi, j \not\models \phi_{\neg 3/7}$; en effet la valeur de compteur en q_i^1, celle en q_i^5 et celle en $q_{i,1}$ sont égales,
9. si q est égal à q_i^5 avec $i \in [1..m]$, on a $\pi, j \not\models \phi_{\neg 3/7}$; en effet la valeur de compteur en q_i^5, celle en $q_{i,1}$ et celle en $q_{i,2}$ sont égales,
10. si q est égal à $q_{i,i}^0$ avec $i \in [1..m-1]$, on a $\pi, j \not\models \phi_{\neg 3/7}$; en effet la valeur de compteur en $q_{i,1}^0$, celle en $q'_{i,i+1}$ et celle en $q'_{i,i+2}$ sont égales,
11. si q est égal à $q_{m,m}^0$, on a $\pi, j \not\models \phi_{\neg 3/7}$; en effet la valeur de compteur en $q_{m,m}^0$, celle en $q'_{m,m+1}$ et celle en $q_{i,F}$ sont égales,

12. si q est égal à $q^1_{i,i}$ avec $i \in [1..m]$, on a $\pi, j \not\models \phi_{\neg 3/7}$; en effet la valeur de compteur en $q^1_{i,i}$, celle en $q^3_{i,i}$ et celle en $q_{i,i+1}$ sont égales,
13. si q est égal à $q^2_{i,i}$ avec $i \in [1..m-1]$, on a $\pi, j \not\models \phi_{\neg 3/7}$; en effet la valeur de compteur en $q^2_{i,i}$, celle en $q_{i,i+1}$ et celle en $q_{i,i+2}$ sont égales,
14. si q est égal à $q^2_{m,m}$, on a $\pi, j \not\models \phi_{\neg 3/7} \wedge \phi_{0\sim 6}$; en effet si $\pi, j \models \phi_{0\sim 6}$ comme la valeur de compteur en $q^2_{m,m}$ et celle en $q_{m,m+1}$ sont égales, si celle 6 "coups" plus loin est aussi égale, on a $\pi, j \not\models \phi_{\neg 3/7}$,
15. si q est égal à $q_{i,k}$ avec $i \in [1..m]$ et $k \in [1..m-1]$ et soit $|(i-k)| > 2$ soit $k > i$, on a $\pi, j \not\models \phi_{\neg 3/7}$; en effet la valeur de compteur en $q_{i,k}$, celle en $q_{i,k+1}$ et celle en $q_{i,k+2}$ sont alors égales,
16. si q est égal à $q_{i,i-1}$ avec $i \in [2..m]$, on a $\pi, j \not\models \phi_{\neg 3/7}$; en effet la valeur de compteur en $q_{i,i-1}$, celle en $q^1_{i,i}$ et celle en $q^2_{i,i}$ sont alors égales,
17. si q est égal à $q_{i,i-2}$ avec $i \in [3..m]$, on a $\pi, j \not\models \phi_{\neg 3/7}$; en effet la valeur de compteur en $q_{i,i-2}$, celle en $q_{i,i-1}$ et celle en $q^1_{i,i}$ sont alors égales,
18. si q est égal à $q_{i,m}$ avec $i \in [1..m-1]$, on a $\pi, j \not\models \phi_{\neg 3/7} \wedge \phi_{0\sim 6}$; en effet si $\pi, j \models \phi_{0\sim 6}$ comme la valeur de compteur en $q_{i,m}$ et celle en $q_{i,m+1}$ sont égales, si celle 6 "coups" plus loin est aussi égale, on a $\pi, j \not\models \phi_{\neg 3/7}$,
19. si q est égal à $q_{i,m+1}$ avec $i \in [1..m]$, $\pi, j \not\models \phi_{\neg 3/7} \wedge \phi_{0\sim 6}$; en effet si $\pi, j \models \phi_{0\sim 6}$, cela signifie que il y a nécessairement une transition $(q_{i,F}, \textbf{dec}, q_k)$ qui est franchie, ainsi la valeur en q^4_k , c'est-à-dire "6" coups plus loin est la même que celle en $q_{i,m+1}$ mais aussi la même que en q^2_k, et donc $\pi, j \not\models \phi_{\neg 3/7}$,
20. si q est égal à $q'_{i,k}$ avec $i \in [1..m]$ et $k \in [1..m-1]$ et soit $|(i-k)| > 2$ soit $k > i$, on a $\pi, j \not\models \phi_{\neg 3/7}$; en effet la valeur du compteur en $q'_{i,k}$, celle en $q'_{i,k+1}$ et celle en $q'_{i,k+2}$ sont égales,
21. si q est égal à $q'_{i,i-1}$ avec $i \in [2..m]$, on a $\pi, j \not\models \phi_{\neg 3/7}$; en effet la valeur de compteur en $q'_{i,i-1}$, celle en $q^0_{i,i}$ et celle en $q'_{i,i+1}$ sont alors égales,
22. si q est égal à $q'_{i,i-2}$ avec $i \in [3..m]$, on a $\pi, j \not\models \phi_{\neg 3/7}$; en effet la valeur de compteur en $q'_{i,i-2}$, celle en $q'_{i,i-1}$ et celle en $q^0_{i,i}$ sont alors égales,
23. si q est égal à $q'_{i,m}$ avec $i \in [1..m]$, on a $\pi, j \not\models \phi_{\neg 3/7}$; en effet la valeur du compteur en $q'_{i,m}$, celle en $q'_{i,,+1}$ et celle en $q'_{i,F}$ sont égales,
24. si q est égal à $q'_{i,m+1}$ avec $i \in [1..m-1]$, on a $\pi, j \not\models \phi_{0\sim 6}$; en effet si $\pi, j \models \phi_{0\sim 6}$, la valeur "6" coups plus loin est atteinte en q^3_k pour $k \in [1..m]$ et ne peut pas être égale à la valeur en $q'_{i,m+1}$ quelle que soit la forme de la transition entre $q_{i,F}$ et q_k,
25. si q est égal à $q_{i,F}$ avec $i \in [1..m]$, on a $\pi, j \not\models \phi_{\neg 3/7} \wedge \phi_{0\sim 6}$; en effet si $\pi, j \models \phi_{0\sim 6}$, cela signifie qu'il y a nécessairement une transition $(q_{i,F}, \textbf{dec}, q_k)$ qui est franchie, ainsi la valeur en q^5_k , c'est-à-dire "6" coups plus loin, est la même que celle en $q_{i,F}$ mais aussi la même qu'en q^1_k, et donc $\pi, j \not\models \phi_{\neg 3/7}$.

Cette étude de cas nous montre que nécessairement nous avons $q \in Q$. Nous posons ensuite pour tout $i \in [1..m]$, $\phi_i = \mathtt{X}^{6+2(i-1)} \downarrow_1 \mathtt{X}^2 \neg \uparrow_1$. On peut alors vérifier que dans une exécution de $\mathcal{A}_P$, la formule $LOC \wedge \phi_i$ est vraie si et seulement si l'état de contrôle courant est q_i, ceci pour tout $i \in [1..m]$.
Soit ϕ une formule de $\mathrm{LTL}^{\downarrow,Q}_n$, nous construisons maintenant la formule ϕ_P qui est égale à $T(\phi)$ où T est un opérateur prenant en entrée une formule de $\mathrm{LTL}^{\downarrow,Q}_n$ et retournant une formule dans

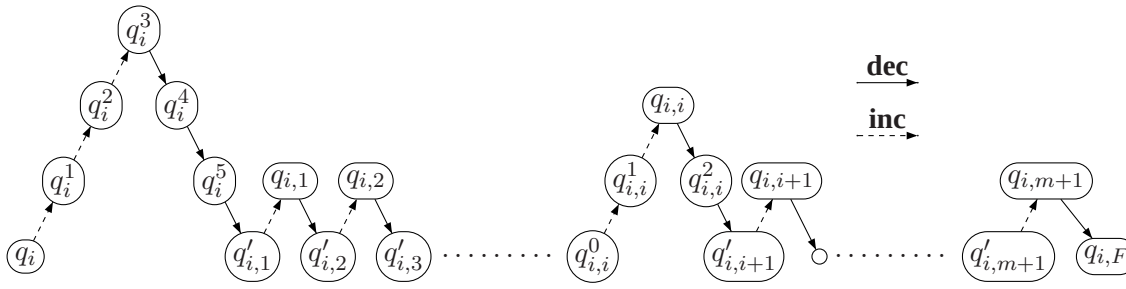


FIGURE 3.1 – Encoder q_i avec un motif caractéristique de longueur $9 + 2(m + 1)$

$\mathrm{LTL}^{\downarrow,\emptyset}_{\mathbf{Max}(n,1)}$ définie par induction comme précisé maintenant :

- $T(q_i) = \phi_i$ pour tout $i \in [1..m]$,
- $T(\uparrow_r) = \uparrow_r$,
- $T(\neg\phi) = \neg T(\phi)$,
- $T(\phi \wedge \phi') = T(\phi) \wedge T(\phi')$,
- $T(\mathtt{X}\phi) = \mathtt{X}^{9+2(m+1)} T(\phi)$,
- $T(\phi \mathtt{U} \phi') = \big(LOC \Rightarrow T(\phi)\big) \mathtt{U} \big(LOC \wedge T(\phi')\big)$,
- $T(\downarrow_r \phi) = \downarrow_r T(\phi)$.

Nous remarquons que ϕ et ϕ_P utilisent le même nombre de registres, sauf dans le cas où ϕ n'utilise aucun registre. Pour chaque exécution acceptante dans $\mathcal{A}$, il existe une exécution acceptante dans $\mathcal{A}_P$, et réciproquement pour chaque exécution acceptante dans $\mathcal{A}_P$, il existe une exécution acceptante dans $\mathcal{A}$, et de plus pour chacune des ces deux exécutions, les valeurs du compteur pour les configurations ayant un état de contrôle dans Q correspondent. Ceci nous permet de déduire le résultat énoncé. □

En utilisant une preuve similaire à celle que nous venons de présenter, nous avons le résultat suivant pour la logique du premier ordre sur les mots de données :

Lemme 3.17 *Soient $\mathcal{A} = \langle Q, E, q_I, F\rangle$ un automate à un compteur, $n \in \mathbb{N}^*$ et ϕ une formule de $\mathrm{FO}^Q(\sim, <, +1)$ avec n variables. Il est possible de calculer en espace logarithmique en $|\mathcal{A}| + |\phi|$ un automate à un compteur $\mathcal{A}_P$ et une formule ϕ_P dans $\mathrm{FO}^\emptyset(\sim, <, +1)$ avec $n + 2$ variables tels que :*

- *$\mathcal{A} \models^{<\omega} \phi$ si et seulement si $\mathcal{A}_P \models^{<\omega} \phi_P$, et,*
- *$\mathcal{A} \models^{\omega} \phi$ si et seulement si $\mathcal{A}_P \models^{\omega} \phi_P$.*

De plus, si $\mathcal{A}$ est déterministe $\mathcal{A}_P$ l'est aussi.

Nous utilisons ici $n + 2$ variables car pour transformer la formule LOC de la preuve précédente en une formule de $\mathrm{FO}^\emptyset(\sim, <, +1)$, nous avons besoin d'écrire des formules de la forme $x = y + k$. En effet, la traduction des formules de la forme $x = y + 1$ donne $x = y + 9 + 2(m + 1)$ (identique au cas $\mathtt{X}\phi$). Or pour encoder $x = y + k$ pour une constante k, il faut utiliser 2 variables auxiliaires. Par exemple, voilà une façon d'encoder la formule $x = y + 4$ avec 2 variables auxiliaires :

$$\exists y_2.x = y_2 + 1 \wedge (\exists y_1.y_2 = y_1 + 1 \wedge (\exists y_2.y_1 = y_2 + 1 \wedge y_2 = y + 1))$$

Nous nous servons ici du fait que l'on puisse recycler les variables y_1 et y_2.

3.2.6 Model-checking d'automates à un compteur déterministes

Nous nous intéressons ici aux problèmes de model-checing présentés précédemment dans le cas particulier où l'automate à un compteur considéré est déterministe.

3.2.6.1 Un problème PSPACE-difficile

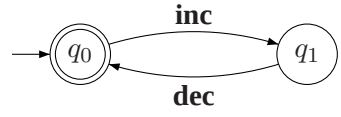
Dans un premier temps, nous allons montrer que le problème de model-checking de formules de la logique LTL avec registres sur des automates déterministes à un compteur est un problème PSPACE-difficile. Pour prouver ce résultat, nous proposons une réduction à partir du problème QBF ("Quantified Boolean Formula") qui est un problème connu pour être PSPACE-complet. Nous avons ainsi la proposition suivante :

Proposition 3.18 *Les problèmes* PureMC(LTL)$^{<\omega}$ *et* PureMC(LTL)$^{\omega}$ *restreints à des automates à un compteur déterministes sont des problèmes* PSPACE*-difficiles.*

Preuve : Nous considérons le problème suivant, qui est une instance de QBF . Soit ψ une formule de la logique propositionnelle égale à :

$$\psi = \forall p_1.\exists p_2.\ldots.\forall p_{2N-1}.\exists p_{2N}.\Psi(p_1,\ldots,p_{2N})$$

où $N \in \mathbb{N}^*$, $p_1, p_2, \ldots, p_{2N}$ sont des variables propositionnelles et $\Psi(p_1,\ldots,p_{2N})$ est une formule de la logique propositionnelle sans quantificateur dont les variables libres sont $p_1, p_2, \ldots, p_{2N}$. Le problème de satisfiabilité étudié consiste à savoir si la formule ψ est satisfaite (ce qui équivalent à savoir si elle est valide, ie égale à $Vrai$, car elle n'a pas de variable libre). Ce problème est connu comme étant PSPACE-complet.
Nous considérons l'automate déterministe $\mathcal{A}$ à un compteur dessiné ci-dessous.



q_0 étant l'état initial et l'unique état acceptant de $\mathcal{A}$. Notons que nous n'avons pas représenté la transition étiqueté par **ifzero** partant de q_1 car dans les exécutions de $TS(\mathcal{A})$, cette transition n'est jamais franchie. Cet automate génère une suite de valeurs de compteurs de la forme $(01)^{\omega}$. De façon, à faciliter la présentation, dans cette preuve, nous enregistrons dans les registres d'une formule de LTL$^{\downarrow}$, non plus la position mais la valeur du compteur associé à cette position. Nous construisons alors une formule ϕ de LTL$^{\downarrow,\emptyset}_{2N+1}$ à partir des formules $\phi_1,\ldots,\phi_{2N+1}$ décrites ci-après de telle sorte que $\phi = \downarrow_{2N+1} \phi_1$. Nous définissons ainsi :

- $\phi_{2N+1} = \Psi[p_i \leftarrow (\uparrow_i \Leftrightarrow \uparrow_{2N+1})]$ qui est la formule obtenue à partir de $\Psi(p_1,\ldots,p_{2N+1})$ en substituant pour chaque $i \in [1..2N+1]$, la variable propositionnelle p_i par la formule $\uparrow_i \Leftrightarrow \uparrow_{2N+1}$,
- pour chaque $i \in [1..N]$, $\phi_{2i} = \mathtt{F}(\downarrow_{2i} \phi_{2i+1})$ et $\phi_{2i-1} = \mathtt{G}(\downarrow_{2i-1} \phi_{2i})$.

Nous allons prouver que ψ est satisfiable si et seulement si $\mathcal{A} \models^{\omega} \psi$.
Pour chaque $i \in 0, 2, 4, 6, \ldots, 2N$, nous notons ψ_i la sous-formule de ψ égale à :

$$\psi_i = \forall p_{i+1}.\exists p_{i+2}.\ldots.\forall p_{2N-1}.\exists p_{2N}.\Psi(p_1,\ldots,p_{2N})$$

De la même façon, pour chaque $i \in \{1, 3, 5, \ldots, 2N-1\}$, nous définissons

$$\psi_i = \exists p_{i+1}.\forall p_{i+2}.\ldots.\forall p_{2N-1}.\exists p_{2N}.\Psi(p_1, \ldots, p_{2N})$$

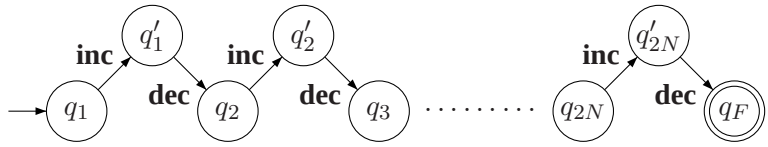
Remarquons que ψ_0 est précisément la formule ψ et que pour chaque $i \in [0..2N]$, les variables libres de ψ_i sont les variables propositionnelles $p_1, \ldots, p_i$. Pour tout $i \in [1..2N]$, pour toute valuation $u_i : \{p_1, \ldots, p_i\} \to \{true, false\}$ qui à chaque variable propositionnelle dans $\{p_1, \ldots, p_i\}$ associe une valeur booléenne, nous construisons une valuation de registres v_i telle que $\mathbf{dom}(v_i) = \{1, \ldots, i, 2N+1\}$ et $v_i(2N+1) = 0$ et telle que pour tout $j \in [1..i]$, $u_i(p_j) = true$ si et seulement si $v_i(j) = 0$. Nous montrons que pour tout $i \in [0..2N]$, $u_i : \{p_1, \ldots, p_i\} \to \{true, false\}$ vérifie $u_i \models \psi_i$ si et seulement si pour tout $k \in \mathbb{N}$, $\pi^\omega_{\mathcal{A}}, k \models_{v_i} \phi_{i+1}$ (où $\pi^\omega_{\mathcal{A}}$ est l'unique exécution infinie acceptante de $TS(\mathcal{A})$). Nous montrons cela par induction sur i.

Tout d'abord si $i = 2N$, nous avons $\psi_{2N} = \Psi(p_1, \ldots, p_n)$. Soit u_{2N} une valuation pour les variables $\{p_1, \ldots, p_i\}$, et pour tout $j \in [1..2N]$, nous avons $v_{2N}(j) = 0$ si et seulement si $u_{2N}(j) = true$, et comme $v(2N+1) = 0$, nous en déduisons que $\pi^\omega_{\mathcal{A}}, k \models_{v_{2N}} \uparrow_j \Leftrightarrow \uparrow_{2N+1}$ si et seulement si $u_{2n}(j) = true$. Comme $\phi_{2N+1} = \Psi[p_i \leftarrow (\uparrow_i \Leftrightarrow \uparrow_{2N+1})]$, on en déduit que $u \models \Psi(p_1, \ldots, p_{2N})$ si et seulement si pour tout $k \in \mathbb{N}$, $\pi^\omega_{\mathcal{A}}, k \models_{v_{2N}} \phi_{2N+1}$.

Soit $i \in [0..(2N-1)]$ et supposons que la propriété est vraie pour $i+1$. Nous distinguons le cas où i est pair et la cas où i est impair. Si i est pair, nous avons $\psi_i = \forall p_{i+1}\psi_{i+1}$ et $\phi_{i+1} = \mathtt{G}(\downarrow_{i+1} \phi_{i+2})$. Soit u_i une valuation pour les variables $\{p_1, \ldots, p_i\}$, et pour tout $j \in [1..i]$, nous avons $v_i(j) = 0$ si et seulement si $u_i(j) = true$. Tout d'abord supposons que $u_i \models \forall p_{i+1}\psi_{i+1}$, alors pour toute valuation de registres u_{i+1} sur les variables $\{p_1, \ldots, p_i, p_{i+i}\}$ telle que pour tout $j \in [1..i]$, $u_{i+1}(p_j) = u_i(p_j)$, nous avons $u_{i+1} \models \psi_{i+1}$. En utilisant l'hypothèse d'induction, on en déduit que pour tout $k \in \mathbb{N}$, $\pi^\omega_{\mathcal{A}}, k \models_{v_{i+1}} \phi_{i+2}$ pour toute valuation de registres v_{i+1} telle que pour tout $j \in [1..i]$, $v_{i+1}(j) = 0$ si et seulement si $u_{i+1}(p_j) = true$, ie si et seulement si $u_i(p_j) = true$. Par conséquent, on déduit que nécessairement, pour tout $k \in \mathbb{N}$, $\pi^\omega_{\mathcal{A}}, k \models_{v_i} \mathtt{G}(\downarrow_{i=1} \phi_{i+2})$, puisque la formule sera satisfaite peu importe si l'on enregistre un 0 ou 1 dans le registre $i+1$. De la même façon, si pour tout $k \in \mathbb{N}$, $\pi^\omega_{\mathcal{A}}, k \models_{v_i} \mathtt{G}(\downarrow_{i=1} \phi_{i+2})$, nous avons $u_i \models \forall p_{i+1}\psi_{i+1}$. Le cas où i est pair se prouve exactement de manière identique.

Ceci nous permet de déduire que ψ est satisfiable si et seulement si $\mathcal{A} \models^\omega \phi$, car la valeur mise dans le $2N+1$ registre est bien 0, étant donné qu'il s'agit de la première valeur de l'exécution. Comme la construction de ϕ est réalisée en espace logarithmique en fonction de $|\psi|$, nous obtenons alors que le problème PureMC(LTL)$^\omega$ restreint à des automates à un compteur déterministes est PSPACE-difficile.

Cette preuve ne fonctionne plus dans le cas fini, car l'opérateur G pourrait nous amener dans la dernière position de l'exécution finie, et après il ne serait plus possible d'avoir le choix entre différentes valeurs de compteurs pour le quantificateur existentiel. Pour prouver ce résultat dans le cas fini, c'est-à-dire pour PureMC(LTL)$^{<\omega}$, il convient d'utiliser un autre automate à un compteur déterministe avec $4N+1$ états de contrôle de telle façon que la suite de valeurs générée soit égale à $(01)^{2N}0$ et ensuite on complète les formules ϕ_i en utilisant l'opérateur X. On considère ainsi le nouvel automate à un compteur déterministe $\mathcal{A}$ suivant :



Nous construisons une nouvelle formule ϕ de $\text{LTL}_{2N+1}^{\downarrow,\emptyset}$ à partir des formules $\phi_1, \ldots, \phi_{2N+1}$ décrites ci-après de telle sorte que $\phi = \downarrow_{2N+1} \phi_1$. Nous définissons ainsi :

- $\phi_{2N+1} = \Psi[p_i \leftarrow (\uparrow_i \Leftrightarrow \uparrow_{2N+1})]$ qui est la formule obtenue à partir de $\Psi(p_1, \ldots, p_{2N+1})$ en substituant pour chaque $i \in [1..2N+1]$, la variable propositionnelle p_i par la formule $\uparrow_i \Leftrightarrow \uparrow_{2N+1}$,
- pour chaque $i \in [1..N]$, nous définissons ensuite $\phi_{2i} = \texttt{F}((\texttt{X}^{4N-4i+2} true) \wedge \downarrow_{2i} \phi_{2i+1})$ et $\phi_{2i-1} = \texttt{G}((\texttt{X}^{4N-4i+4} true) \Rightarrow \downarrow_{2i-1} \phi_{2i})$.

Grâce à une preuve par induction similaire à celle que nous avons faite pour le cas infini, nous obtenons là aussi le résultat stipulant que $\mathcal{A} \models^{<\omega} \phi$ si et seulement si la formule ψ est satisfiable. □

Une chose que nous pouvons remarquer dans la réduction de QBF que nous proposons est qu'elle utilise un nombre non borné de registres, et donc nous n'avons pas la PSPACE-dureté dans le cas où les formules ont un nombre borné de registres.

En utilisant le résultat de la proposition 3.14 qui nous dit que nous pouvons transformer en temps linéaire toute formule de $\text{LTL}^{\downarrow,\emptyset}$ en une formule de $\text{FO}^{\emptyset}(\sim, <, +1)$, nous déduisons le corollaire suivant :

Corollaire 3.19 *Les problèmes* PureMC(FO)$^{<\omega}$ *et* PureMC(FO)$^{\omega}$ *restreints à des automates à un compteur déterministes sont des problèmes* PSPACE*-difficiles.*

3.2.6.2 Propriétés des exécutions pour les automates à un compteur déterministes

Chaque automate à un compteur déterministe $\mathcal{A} = \langle Q, E, q_I, F \rangle$ a au plus une exécution infinie qui peut passer par un nombre infini de valeurs de compteurs. Si cette exécution infinie n'est pas acceptante, c'est-à-dire si cette exécution ne passe pas infiniment souvent par un état de contrôle acceptant, alors pour aucune formule ϕ de $\text{LTL}^{\downarrow,Q}$ ou de $\text{FO}^{Q}(\sim, <, +1)$, nous avons $\mathcal{A} \models^{\omega} \phi$. Nous allons montrer par la suite que nous pouvons décider en temps polynomial si $\mathcal{A}$ a des exécutions acceptantes finies ou infinies, et nous verrons aussi que l'unique exécution infinie de $\mathcal{A}$ vérifie certaines propriétés de régularité.

Soit $\mathcal{A} = \langle Q, E, q_I, F \rangle$ un automate à un compteur déterministe. Soit $\pi_{\mathcal{A}}^{\omega}$ l'unique exécution infinie de $TS(\mathcal{A})$ (si elle existe). Nous supposons que $\pi_{\mathcal{A}}^{\omega}$ peut être représentée par la séquence de configurations suivante :

$$(q_0, n_0)\ (q_1, n_1)\ (q_2, n_2)\ \ldots$$

Nous allons voir que $\pi_{\mathcal{A}}^{\omega}$ vérifient certaines conditions particulières dont nous nous servirons pour résoudre les différents problèmes de model-checking.

Tout d'abord, nous donnons un résultat caractérisant l'ensemble des positions de l'exécution pour lesquelles un test à 0 a été réalisé, c'est-à-dire les positions à partir desquelles une transition étiquetée par **ifzero** a été franchie . Nous notons **ZERO**$(\mathcal{A})$ cet ensemble. Par convention 0 appartient à **ZERO**$(\mathcal{A})$ car nous imposons aux exécutions de $TS(\mathcal{A})$ de commencer avec la valeur 0, ainsi **ZERO**$(\mathcal{A}) = \{0\} \cup \{i > 0 \mid n_i = n_{i+1} = 0\}$. Nous avons alors le lemme suivant :

Lemme 3.20 *Soient $i, j \in \mathbb{N}$ tels que $i < j$ et $i, j \in$* **ZERO**$(\mathcal{A})$*. Si il n'existe pas de $k \in$* **ZERO**$(\mathcal{A})$ *tel que $i < k < j$ alors $(j - i) \leq |Q|^2$.*

Preuve : La preuve repose sur le fait que la valeur du compteur ne peut pas aller au-dessus de $|Q|$ entre deux positions consécutives pour lesquelles un test à 0 est réalisé avec succès. D'abord, nous observons qu'il n'y a pas de $k' \in \mathbb{N}$ tel que $i < k < k' < j$ et $q_k = q_{k'}$ et $n_k = n_{k'}$. En effet, si cela était le cas, comme il n'y a pas de test à zéro réalisé dans la séquence $(q_{i+1}, n_{i+1}) \dots (q_k, n_k) \dots (q_{k'}, n_{k'})$, nous pourrions construire une exécution infinie répétant infiniment souvent la méme suite d'états de contrôle que celle entre q_k et $q_{k'}$ et qui ne réaliserait jamais de tests à zéro (ceci car $n_k \leq n_{k'}$); or cela est une contradiction avec le fait que $\mathcal{A}$ est déterministe et avec l'existence de la configuration (q_j, n_j) à partir de laquelle un test à zéro est réalisé. Nous en déduisons que si il existe $k' \in \mathbb{N}$ tel que $i < k < k' < j$ et $q_k = q_{k'}$ alors nécessairement $n_{k'} < n_k$. Nous supposons maintenant qu'il existe $k \in \mathbb{N}$ tel que $i < k < j$ et $|Q| \leq n_k$. Alors nous pouvons extraire une sous-suite $(q_{i_0}, n_{i_0}) \dots (q_{i_s}, n_{i_s})$ de $(q_i, n_i) \dots (q_k, n_k)$ telle que $i_0 = i$, $i_s = k$ et pour tout $l \in [0..s]$, $n_{i_l+1} = n_{i_l} + 1$ (car la valeur du compteur augmente au plus de 1 unité à chaque franchissement de transitions). Par conséquent, il existe $l, l' \in [0..s]$ tels que $q_{i_l} = q_{i_{l'}}$ et $n_{i_l} < n_{i_{l'}}$, ce qui constitue une contradiction avec le point précédent. Par conséquent, pour $k \in [i..j]$, nous avons $n_k \leq |Q| - 1$. Comme $\mathcal{A}$ est déterministe, cela implique que $(j - i) \leq |Q| \times (|Q| - 1)$ car entre les positions i et j on ne peut pas avoir deux configurations identiques. □

Ce lemme nous permet de caractériser l'unique exécution infinie $\pi^\omega_{\mathcal{A}}$ de $TS(\mathcal{A})$ de la façon suivante :

Lemme 3.21 *Il existe $K_1, K_2, K_{inc} \in \mathbb{N}$ tels que :*

1. $K_1 + K_2 \leq |Q|^3$,
2. $K_{inc} \leq |Q|$, *et,*
3. *pour tout $i \in \mathbb{N}$ tel que $K_1 \leq i$, $(q_{i+K_2}, n_{i+K_2}) = (q_i, n_i + K_{inc})$.*

Preuve : Dans un premier temps, nous supposons que l'ensemble **ZERO**$(\mathcal{A})$ possède un nombre infini d'éléments. Soit $i_0 < i_1 < i_2 < \dots$ la séquence infinie des éléments de **ZERO**$(\mathcal{A})$ (avec $i_0 = 0$). Il y a $l, l' \in \mathbb{N}$ tels que $l, l' \leq |Q|$ et tels que $(q_{i_l}, n_{i_l}) = (q_{i_{l'}}, n_{i_{l'}})$. Par le lemme 3.20, nous avons $i'_l \leq |Q| \times |Q|^2$. On prend ainsi $K_1 = i_l$, $K_2 = i_{l'} - i_l$ et $K_{inc} = 0$. Supposons maintenant que l'ensemble **ZERO**$(\mathcal{A})$ est fini et égal à $\{0, i_1, \dots, i_l\}$ pour $l \leq |Q| - 1$ (car si $l \geq |Q|$ nous nous retrouvons dans le cas précédent). Par le lemme 3.20, $i_l \leq (|Q| - 1) \times |Q|^2$. Pour tout $k, k' \in \mathbb{N}$ tels que $l < k < k'$, si $q_k = q'_k$, nous avons nécessairement $n_k < n_{k'}$ (si cela n'était pas le cas, il y aurait à un certain moment un test à zéro franchi dans l'exécution partant de (q_{i_l}, n_{i_l})). Il y a donc $k, k' \in \mathbb{N}$ tels que $i_l \leq k < k' \leq i_l + |Q|$ et tels que $q_k = q_{k'}$, et par conséquent aussi $n_k \leq n_{k'}$. En posant $K_1 = k$, $K_2 = k' - k$ et $K_{inc} = n_{k'} - n_k$, le lemme est vérifié (notons que $K_{inc} \leq |Q|$ car $k' - k \leq |Q|$). □

De ce dernier lemme, nous pouvons en déduire que lorsqu'elle existe, l'unique exécution infinie $\pi^{\omega}_{\mathcal{A}}$ de $TS(\mathcal{A})$ a une structure simple, elle est composée d'un préfixe de taille polynomiale de la forme :

$$(q_0, n_0) \dots (q_{K_1-1}, n_{K_1-1})$$

suivie par une boucle de taille polynomiale de la forme :

$$(q_{k_1}, n_{K_1}) \dots (q_{K_1+K_2-1}, n_{K_1+K_2-1})$$

dont la suite des états de contrôle est répétée infiniment souvent et les valeurs des compteurs sont augmentés de K_{inc} unités à chaque tour de boucle. Ainsi tester si l'automate à un compteur $\mathcal{A}$ a une exécution infinie $\pi^{\omega}_{\mathcal{A}}$ qui est acceptante revient à tester si il existe un état de contrôle acceptant dans la boucle, ce qui peut être fait en temps cubique par rapport à $|Q|$. Dans la suite de cette section, nous supposerons que $\pi^{\omega}_{\mathcal{A}}$ est acceptant. De la même façon, tester si $\mathcal{A}$ admet une exécution finie acceptante revient à vérifier, si il existe un état de contrôle acceptant dans le préfixe ou dans la boucle.

Remarquons que lorsque $\mathcal{A}$ a effectivement une exécution infinie et que la constante K_{inc} est égale à 0, $\pi^{\omega}_{\mathcal{A}}$ est exactement le mot :

$$(q_0, n_0) \dots (q_{K_1-1}, n_{K_1-1})\big((q_{K_1}, n_{K_1}) \dots (q_{K_1+K_2-1}, n_{K_1+K_2-1})\big)^{\omega}$$

3.2.6.3 Un algorithme PSPACE pour le model-checking

Nous allons donner une procédure de décision pour résoudre les problèmes MC(FO)$^{<\omega}$ et MC(FO)$^{\omega}$ lorsque les automates à un compteur considérés sont déterministes. Nous reprenons les notations introduites précédemment pour le lemme 3.21. Nous supposons de plus que l'automate à un compteur déterministe $\mathcal{A}$ que nous considérons admet une exécution infinie $\pi^{\omega}_{\mathcal{A}}$.

Nous montrons que lorsque $K_{inc} > 0$, alors deux positions avec la même valeur de compteur sont séparées par une distance qui peut être bornée par un polynôme dépendant de $|Q|$. Avant de prouver cela, nous introduisons quelques constantes concernant l'automate à un compteur $\mathcal{A}$ et son unique exécution infinie $\pi^{\omega}_{\mathcal{A}}$ lorsque $K_{inc} > 0$. Ainsi nous notons :

- $\beta_1, \beta_2 \in \mathbb{N}$ les plus petits entiers naturels tels que pour tout $i \in [K_1..K_1 + K_2 - 1]$, $n_i \in [n_{K_1} - \beta_1..n_{K_1} + \beta_2]$,
- $\gamma \in \mathbb{N}$ la plus grande valeur parmi $\{n_0, \dots, n_{K_1-1}\}$, et,
- $L = 1 + \gamma + \ulcorner \frac{\beta_1+\beta_2}{K_{inc}} \urcorner$

Nous notons par $\ulcorner . \urcorner$ la fonction partie entière par excès, c'est-à-dire la fonction qui étant donné un nombre réel renvoie le plus petit entier supérieur ou égal à ce nombre.

Intuitivement la constante LK_2 majore la distance maximale qu'il existe entre deux positions dans la boucle qui ont des valeurs de compteurs identiques. Cette constante est importante car elle nous permet de dire que si deux positions dans le mot de données généré par l'automate à un compteur considéré sont à la distance LK_2 l'une de l'autre alors leurs données sont nécessairement différentes. Le lemme qui suit formalise ce point.

Lemme 3.22 *Supposons que $K_{inc} > 0$. Soient $i, j \in \mathbb{N}$. Alors, les propriétés suivantes sont vérifiées :*

1. *Si $i \geq K_1$ et $j \geq K_1$ et $|i - j| \geq LK_2$, alors $n_i \neq n_j$.*
2. *Si $i < K_1$ et $j \geq K_1 + LK_2$, alors $n_1 \neq n_j$.*

Preuve : (1.) Supposons que $i \geq K_1$ et $j \geq K_1$ et $(i-j) \geq LK_2$. Nous considérons les entiers $r_i = (i-K_1)mod(K_2)$ et $r_j = (j-K_1)mod(K_2)$. De plus, nous définissons les quotients a_i et a_j de telle façon que $i - K_1 = a_iK_2 + r_i$ et $j - K_1 = a_jK_2 + r_j$. Notons que par définition de l'opération modulo et compte tenu du fait que $(i-j) \geq LK_2$, nous avons nécessairement $0 \leq r_i < K_2$ et $0 \leq r_j < K_2$ et par conséquent $a_i - a_j > L - 1$. D'après la définition des constantes β_1 et β_2, alors $n_{r_i+K_1}, n_{r_j+K_1} \in [n_{K_1} - \beta_1..n_{K_1} + \beta_2]$. Comme $i = a_iK_2 + r_i + K_1$ et $j = a_jK_2 + r_j + K_1$, en utilisant le lemme 3.21, nous obtenons que $n_i = n_{r_i+K_1} + a_iK_{inc}$ et $n_j = n_{r_j+K_1} + a_jK_{inc}$. Nous en déduisons que :

$$n_{K_1} - \beta_1 + a_iK_{inc} \leq n_i \leq n_{K_1} + \beta_2 + a_iK_{inc}$$
$$n_{K_1} - \beta_1 + a_jK_{inc} \leq n_j \leq n_{K_1} + \beta_2 + a_jK_{inc}$$

D'où nous déduisons les inégalités suivantes :

$$-\beta_1 - \beta_2 + (a_i - a_j)K_{inc} \leq n_i - n_j \leq \beta_1 + \beta_2 + (a_i - a_j)K_{inc}$$

En considérant le fait que $(a_i - a_j) > L - 1$ et en utilisant la définition de L, on obtient :

$$0 \leq \gamma K_{inc} < n_i - n_j$$

Par conséquent $n_1 \neq n_j$. Le même raisonnement peut-être mené dans le cas où $(j-i) \geq LK_2$.

(2.) Nous supposons maintenant que $i < K_1$ et $j \geq K_1 + LK_2$. En adoptant les mêmes notations que précédemment pour j et par le même raisonnement, on obtient l'inégalité suivante :

$$n_{K_1} - \beta_1 + a_jK_{inc} \leq n_j \leq n_{K_1} + \beta_2 + a_jK_{inc}$$

Comme $\beta_2 \geq 0$, on obtient :

$$n_{K_1} - \beta_1 - \beta_2 + a_jK_{inc} - n_i \leq n_j - n_i$$

De plus comme $j \geq K_1 + LK_2$, on a nécessairement $a_j \geq L$, d'où :

$$n_{K_1} - \beta_1 - \beta_2 + LK_{inc} - n_i \leq n_j - n_i$$

Si $n_i = n_j$ et en utilisant la définition de L, on a :

$$n_{K_1} + K_{inc}(1 + \gamma) - n_i \leq 0$$

or, comme $i \in [0..K_1 - 1]$, on a $n_i \leq \gamma$ et $K_{inc} > 0$, on obtient finalement $n_{K_1} + K_{inc} \leq 0$ ce qui est une contradiction avec le fait que $n_{K_1} \geq 0$ et $K_{inc} > 0$.

□

Nous allons également utiliser les ensemble $P^1_\sim$ et $P^2_\sim$ définis ainsi :

$$P^1_\sim = \left\{(i,j) \in [0..K_1 + LK_2 - 1]^2 \mid n_i = n_j \wedge i \leq j\right\}$$

$$P^2_\sim = \left\{(i,j) \in [0..K_1 + LK_2 - 1]^2 \mid n_i = n_j + LK_{inc} \wedge j < i\right\}$$

L'idée principale que nous allons développer consiste à utiliser les ensembles $P^1_\sim$ et $P^2_\sim$ et les propriétés des constantes L, K_1, K_2 et K_{inc} pour caractériser les positions de l'exécution $\pi^\omega_\mathcal{A}$ qui ont des valeurs identiques.

Lemme 3.23 *Supposons que $K_{inc} > 0$. Soient $i, j \in \mathbb{N}$ tels que $i \leq j$. Alors $n_i = n_j$ si et seulement si une des conditions suivante est vérifiée :*

1. *$i < K_1 + LK_2$ et $j < K_1 + LK_2$ et $(i,j) \in P^1_\sim$, ou,*
2. *$i \geq K_1$ et $j \geq K_1$ et $(K_1 + (i - K_1)mod(LK_2), K_1 + (j - K_1)mod(LK_2)) \in P^1_\sim$ et $(j - i) < L.K_2$, ou,*
3. *$i \geq K_1$ et $j \geq K_1$ et $(K_1 + (i - K_1)mod(LK_2), K_1 + (j - K_1)mod(LK_2)) \in P^2_\sim$ et $(j - i) < LK_2$, ou,*
4. *$i \in [0..K_1 - 1]$ et $K_1 \leq j$, alors $(i,j) \in P^1_\sim$.*

Preuve : Soient $i, j \in \mathbb{N}$ tels que $i \leq j$. Supposons que la condition 1. est vérifiée, c'est-à-dire que $i < K_1 + LK_2$ et $j < K_1 + LK_2$ et $(i,j) \in P^1_\sim$. Alors par définition de $P^1_\sim$, nous avons bien $n_i = n_j$.

Supposons que la condition 2. est vérifiée, c'est-à-dire que $i \geq K_1$ et $j \geq K_1$ et $(K_1 + (i - K_1)mod(LK_2), K_1 + (j - K_1)mod(LK_2)) \in P^1_\sim$ et $(j - i) < LK_2$. Nous posons $r_i = (i - K_1)mod(LK_2)$ et $r_j = (j - K_1)mod(LK_2)$. Nous posons ensuite a_i tel que $i - K_1 = a_i LK_2 + r_i$ et a_j tel que $j - K_1 = a_j LK_2 + r_j$. D'après le lemme 3.21, nous avons $n_i = n_{r_i + K_1 + a_i LK_2} = n_{r_i + K_1} + a_i LK_{inc}$ et $n_j = n_{r_j + K_1 + a_j LK_2} = n_{r_j + K_1} + a_j LK_{inc}$. Comme de plus $(j - i) < LK_2$, nous avons $(a_j - a_i)LK_2 + (r_j - r_i) < LK_2$ et comme $(K_1 + r_i, K_1 + r_j) \in P^1_\sim$, nous en déduisons que $r_i \leq r_j$ et donc $a_i = a_j$ Par conséquent $n_i = n_j$.

Supposons que la condition 3. est vérifiée, c'est-à-dire que $i \geq K_1$ et $j \geq K_1$ et $(K_1 + (i - K_1)mod(LK_2), K_1 + (j - K_1)mod(LK_2)) \in P^2_\sim$ et $(j - i) < LK_2$. Nous posons $r_i = (i - K_1)mod(LK_2)$ et $r_j = (j - K_1)mod(LK_2)$. Nous posons ensuite a_i tel que $i - K_1 = a_i LK_2 + r_i$ et a_j tel que $j - K_1 = a_j LK_2 + r_j$. D'après le lemme 3.21, nous avons $n_i = n_{r_i + K_1 + a_i LK_2} = n_{r_i + K_1} + a_i LK_{inc}$ et $n_j = n_{r_j + K_1 + a_j LK_2} = n_{r_j + K_1} + a_j LK_{inc}$. Comme de plus $(j - i) < LK_2$, nous avons $(a_j - a_i)LK_2 + (r_j - r_i) < LK_2$ et comme $(K_1 + r_i, K_1 + r_j) \in P^2_\sim$, nous en déduisons que $r_j < r_i$ et donc $a_j = a_i + 1$. Par conséquent $n_j = n_{r_j + K_1} + (a_i + 1)LK_{inc}$ et comme $n_{r_j + K_1} + LK_{inc} = n_{r_i + K_1}$. On en déduit que $n_i = n_j$.

Supposons que la condition 4. est vérifiée, c'est-à-dire que $i \in [0..K_1 - 1]$ et $K_1 \leq j$, et $(i,j) \in P^1_\sim$. Comme $(i,j) \in P^1_\sim$, nous avons $n_i = n_j$.

Nous supposons maitenant que $n_i = n_j$. Et nous procédons par étude de cas :

- Supposons que $i < K_1 + LK_2$ et $j < K_1 + LK_2$. Par définition de $P^1_\sim$, nous avons effectivement $(i,j) \in P_\sim$ et la condition 1. est vérifiée.
- Supposons que $i \geq K_1$ et $j \geq K_1$. D'après le lemme 3.22, nous avons nécessairement $(j - i) < LK_2$ (sinon nous aurions $n_i \neq n_j$). Nous posons $r_i = (i - K_1)mod(LK_2)$ et $r_j = (j - K_1)mod(LK_2)$. Nous posons ensuite a_i tel que $i - K_1 = a_i LK_2 + r_i$ et a_j tel que $j - K_1 = a_j LK_2 + r_j$. D'après le lemme 3.21, nous avons $n_i = n_{r_i + K_1 + a_i LK_2} = n_{r_i + K_1} + a_i LK_{inc}$ et $n_j = n_{r_j + K_1 + a_j LK_2} = n_{r_j + K_1} + a_j LK_{inc}$. Deux cas se posent alors, soit $a_i = a_j$, soit $a_i \neq a_j$.
  - Si $a_i = a_j$. Alors nous avons directement $n_{r_i + K_1} = n_{r_j + K_!}$ et comme $i \leq j$, nous avons $r_i \leq r_j$ et la condition 2. est vérifiée.
  - Si $a_i \neq a_j$. Comme $(j - i) < LK_2$, en utilisant les propriétés de l'opération modulo, nous avons nécessairement $a_j = a_i + 1$, nous avons donc $n_{r_j + K_1} = n_i - (a_i + 1)LK_{inc}$, et comme

$(a_j - a_i)LK_2 + (r_j - r_i) < LK_2$, nous avons aussi $r_j < r_i$ d'où nous déduisons que la condition 3. est vérifiée.

– Supposons que $i < K_1$ et $j \geq K_1$, alors d'après le lemme 3.22, nous avons $j < K_1 + L.K_2$, et par conséquent $(i, j) \in P^1_\sim$.

Cette étude de cas couvre bien tous les cas possibles pour i et j. □

Nous notons $P_\sim$ l'ensemble $P^1_\sim \cup P^2_\sim$. Nous considérons l'alphabet fini $\Sigma = \{0, \ldots, LK_2 - 1\}$ et les mots $s, t \in \Sigma^*$ définis de la façon suivante :

$$s = 0.1.\ldots.K_1 - 1 \quad \text{et} \quad t = K_1.K_1 + 1.\ldots.LK_2 - 1$$

Nous allons maintenant transformer une formule fermée ϕ de $\text{FO}^\emptyset(\sim, <, +1)$ en une formule fermée $T(\phi)$ appartenant à $\text{FO}^\Sigma(<, +1)$, ce qui nous permettra de réduire le problème de model-checking de formules de $\text{FO}(\sim, <, +1)$ sur des automates à un compteur déterministes au model-checking de $\text{FO}(<, +1)$ sur des chemins infinis périodiques et d'utiliser ensuite le résultat du théorème 3.15.

Nous définissons maintenant par induction la fonction T qui prend comme entrée une formule de $\text{FO}^\emptyset(\sim, <, +1)$ et retourne une formule de $\text{FO}^\Sigma(<, +1)$:

– $T(x < y) = \ x < y$,
– $T(x = y + 1) = \ x = y + 1$,
– $T(\neg\phi) = \ \neg T(\phi)$,
– $T(\phi \wedge \phi') = \ T(\phi) \wedge T(\phi')$,
– $T(\exists x.\phi) = \exists x.T(\phi)$,
– et finalement $T(x \sim y) = (x \leq y \wedge T_1(x, y)) \vee (y \leq x \wedge T_1(y, x))$ où $T_1(x, y)$ est égale à :

$$\begin{array}{c}(x < K_1 + LK_2 \wedge y < K_1 + LK_2 \wedge \bigvee_{(I,J)\in P_\sim} I(x) \wedge J(y)) \vee \\ \big(\neg(x < K_1 + LK_2 \wedge y < K_1 + LK_2) \wedge \bigvee_{(I,J)\in P_\sim} I(x) \wedge J(y) \wedge \\ (x \geq K_1 \wedge y \geq K_1) \Rightarrow (y - x) < LK_2\big)\end{array}$$

Nous remarquons que les formules de la forme $x < K_1 + LK_2$ et $(y - x) < LK_2$ sont en réalité des "raccourcis" pour des formules de $\text{FO}^\Sigma(<, +1)$ de taille polynomiale en $|\mathcal{A}|$. En recyclant les variables, $x < K_1 + LK_2$ peut être facilement transformée en une formules de $\text{FO}^\Sigma(<, +1)$ utilisant au plus 3 variables. Par exemple, la formule $x \geq 3$ peut être écrit comme suit :

$$\exists y_1.\big(x > y_1 \wedge (\exists y_2.(y_1 = y_2 + 1 \wedge (\exists y_1.y_2 = y_1 + 1 \wedge \neg(\exists y_2.y_2 < y_1))))\big)$$

Nous avons déjà utilisé une technique similaire dans l'explication suivant le lemme 3.17. De la même façon, quand la formule $x \geq K_1 \wedge y \geq K_1 \wedge y > x$ est satisfaite, $(y - x) \leq K_2$ est équivalente à une formule utilisant au plus 3 variables, à savoir :

$$\neg \bigwedge_{I=K_1}^{K_1+LK_2-1} \exists z.x \leq z < y \wedge I(z)$$

En effet, si la formule précédente se trouvant dans la négation était vraie, cela signifierait qu'il existerait plus de LK_2 positions entre la position x et la position y, et par conséquent la formule

$(y - x) \leq LK_2$ ne serait pas satisfaite. La réciproque étant également vraie.

Une fois la transformation T définie, en utilisant les propriétés énoncées par le lemme 3.23, nous obtenons le lemme suivant qui nous assure la correction de la réduction du model-checking de formules de $\text{FO}^{\emptyset}(\sim, <, +1)$ sur des automates à un compteur déterministes au problème de model-checking de formules de $\text{FO}^{\Sigma}(<, +1)$ sur des chemins infinis périodiques :

Lemme 3.24 *Soit ϕ une formule de $\text{FO}^{\emptyset}(\sim, <, +1)$. Nous avons $\mathcal{A} \models^{\omega} \phi$ si et seulement si $s.t^{\omega} \models T(\phi)$.*

Preuve : La preuve se fait par induction structurelle. Nous prouvons que pour chaque sous-formule ψ de ϕ et pour chaque valuation de registres u, $\mathcal{A} \models^{\omega}_{u} \psi$ si et seulement si $s.t^{\omega} \models_{u} T(\phi)$. Comme $T(x < y) = \ x < y$, $T(x = y + 1) = \ x = y + 1$, $T(\neg\phi) = \ \neg T(\phi)$, $T(\phi \wedge \phi') = \ T(\phi) \wedge T(\phi')$ et $T(\exists x.\phi) = \exists x.T(\phi)$, l'unique cas qui doit être vérifié concerne les formules atomiques de la forme $x \sim y$. Avant de donner la preuve pour ce cas remarquons que par construction si σ est le mot infini $s.t^{\omega}$ de $\Sigma = \{0, \ldots, K_1 + LK_2 - 1\}$ alors pour tout $i \in \mathbb{N}$ tel que $i \geq K_1$, $\sigma(i) = K_1 + (i - K_1) mod(LK_2)$. Soit u une valuation de variables telle que $x, y \in \mathbf{dom}(u)$ (si x ou y n'appartiennent pas à u, il est facile de montrer que $\mathcal{A} \not\models_{u} x \sim y$ et $s.t^{\omega} \not\models T(x \sim y)$).
Tout d'abord supposons que $\mathcal{A} \models^{\omega}_{u} x \sim y$, cela signifie que l'unique exécution infinie acceptante $\pi^{\omega}_{\mathcal{A}}$ de $\mathcal{A}$ vérifie $\pi^{\omega}_{\mathcal{A}} \models_{u} x \sim y$, par conséquent nous avons $n_{u(x)} = n_{u(y)}$. Montrons que $s.t^{\omega} \models_{u} T(x \sim y)$. Nous supposons que $\sigma = s.t^{\omega}$ et que $u(x) \leq u(y)$ (le cas $u(y) \leq u(x)$ se traitant de façon identique). Nous procédons par une étude de cas et utilisons les résultats du lemme 3.23 et la définition de $T(x \sim y)$:

- si $u(x) < K_1 + LK_2$ et $u(x) < K_1 + LK_2$, alors $\sigma(u(x)) = u(x)$ et $\sigma(u(y)) = u(y)$ et de plus $(u(x), u(y)) \in P_{\sim}$, du coup on a bien $\sigma \models_{u} T(x \sim y)$,
- si $u(x) \geq K_1$ et $u(x) \geq K_1$, alors nécessairement $(u(y) - u(x)) < LK_2$ et $\sigma(u(x)) = K_1 + (i - u(x)) mod(LK_2)$ et $\sigma(u(y)) = K_1 + (i - u(x)) mod(LK_2)$ et par conséquent $(\sigma(u(x), \sigma(u(y)) \in P_{\sim}$, ce qui nous permet de dire que $\sigma \models_{u} T(x \sim y)$,
- si $u(x) < K_1$ alors nécessairement $u(y) < K_1 + LK_2$ et nous retombons dans le premier cas,
- si $u(y) < K_1$ alors nécessairement $u(x) < K_1 + LK_2$ et nous retombons dans le premier cas.

Nous supposons maintenant que le mot $\sigma = s.t^{\omega}$ est tel que $s.t^{\omega} \models_{u} T(x \sim y)$. Nous procédons par étude de cas et supposons que $u(x) \leq u(y)$ (le cas $u(y) \leq u(x)$ se traitant de façon identique) :

- si $u(x) < K_1 + LK_2$ et $u(x) < K_1 + LK_2$, alors $\sigma(u(x)) = u(x)$ et $\sigma(u(y)) = u(y)$ et de plus $(u(x), u(y)) \in P_{\sim}$, du coup on a bien $n_{u(x)} = n_{u(y)}$,
- si $u(x) \geq K_1$ et $u(x) \geq K_1$, alors nécessairement $(u(y) - u(x)) < LK_2$ et $(\sigma(u(x), \sigma(u(y)) \in P_{\sim}$, et comme $\sigma(u(x)) = K_1 + (i - u(x)) mod(LK_2)$ et $\sigma(u(y)) = K_1 + (i - u(x)) mod(LK_2)$, nous en déduisons que $n_{u(x)} = n_{u(y)}$,
- si $u(x) < K_1$ alors nécessairement $u(y) < K_1 + LK_2$ et nous retombons dans le premier cas, de même si $u(y) < K_1$.

□

3.2.6.4 Décidabilité et complexité

Les propositions et lemmes que nous avons démontrés auparavant nous permettent maintenant d'exprimer les résultats de décidabilité et de complexité pour les problèmes de model-checking des lo-

giques LTL avec registres et FO($\sim, <, +1$) lorsque les modèles considérés sont des automates à un compteur déterministes. Tout d'abord, nous avons :

Théorème 3.25 *Le problème* MC(FO)$^\omega$ *restreint aux automates à un compteur déterministes est* PSPACE*-complet.*

Preuve : Soit $\mathcal{A}$ un automate à un compteur déterministe et ϕ une formule de FO$^\emptyset(\sim, <, +1)$. Comme nous l'avons vu précédemment à partir de ϕ et $\mathcal{A}$, il est possible de construire les mots s et t en temps polynomial en $|\mathcal{A}|$ et la formule $T(\phi)$ en temps polynomial en $|\mathcal{A}| + |\phi|$ tel que $\mathcal{A} \models^\omega \phi$ si et seulement si $s.t^\omega \models T(\phi)$ (cf lemmes 3.21 et 3.24). De plus par le théorème 3.15, $s.t^\omega \models T(\phi)$ peut-être vérifié en espace polynomial en $|s|+|t|+|T(\phi)|$. Par conséquent, le problème PureMC(FO)$^\omega$ est dans PSPACE. Comme il existe une réduction logarithmique en espace de MC(FO)$^\omega$ vers PureMC(FO)$^\omega$ (cf lemme 3.17), on déduit que MC(FO)$^\omega$ est aussi dans PSPACE. Quant à la PSPACE-dureté, elle nous est donnée par le corollaire 3.19. □

De la même façon, nous pouvons prouver le résultat similaire dans le cas fini. En effet :

Théorème 3.26 *Le problème* MC(FO)$^{<\omega}$ *restreint aux automates à un compteur déterministes est* PSPACE*-complet.*

Preuve : Soit $\mathcal{A} = \langle Q, E, q_I, F\rangle$ un automate à un compteur déterministe et ϕ une formule de FO$^\emptyset(\sim, <, +1)$. Les mots s et t sont alors les mêmes que pour le cas infini. Quant à la formule ϕ, elle est transformée en une formule de la forme :

$$\exists x_{end}.\big(\bigvee_{q\in F} q(x_{end})\big) \wedge T'(\phi)$$

où $T'(\phi)$ est définie comme $T(\phi)$ mise à part pour les clauses de quantification existentielle qui deviennent $T'(\exists x.\psi) = \exists x.x \leq x_{end} \wedge T'(\psi)$. La variable x_{end} nous permet ainsi de détecter la fin du mot. Nous déduisons ensuite les résultats souhaités comme nous l'avons fait dans le cas infini avec le théorème 3.25. □

En utilisant la traduction des formules de LTL$^\downarrow$ vers des formules de la logique FO($\sim, <, +1$) donnée par la proposition 3.14, nous avons également le corollaire suivant :

Corollaire 3.27 *Les problèmes* MC(LTL)$^\omega$ *et* MC(LTL)$^{<\omega}$ *restreints aux automates à un compteur déterministes sont* PSPACE*-complets.*

Ainsi, comme nous l'espérions, le modèle des automates à un compteur déterministes étant assez simple, nous obtenons des résultats de décidabilité avec des bornes de complexité relativement bonnes. Nous allons voir dans la partie qui suit que lorsque les automates à un compteur considérés ne sont plus déterministes, on passe de la décidabilité à l'indécidabilité.

3.2.7 Model-checking d'automates à un compteur

Nous allons montrer dans cette section que les problèmes de model-checking introduits précédemment sont indécidables, lorsque les modèles considérés sont des automates à un compteur non déterministes. Pour se faire, nous réduisons différents problèmes connus comme étant indécidables sur des machines

de Minsky à deux compteurs. Le plus surprenant dans les résultats d'indécidabilité que nous allons présenter est le fait qu'ils sont préservés même si les formules considérées n'utilisent qu'un unique registre. Ceci contraste en effet avec le fait que le problème de satisfiabilité fini pour les formules de $\text{LTL}^{\downarrow,}$ avec un seul registre est décidable (cf. théorème 3.10) et que le problème de satisfiabilité (finis et infinis) de $\text{FO}(\sim, <, +1)$ pour des formules avec au plus deux variables est lui aussi décidable (cf. théorème 3.12).

Dans [DL06], les auteurs ont introduit le modèle des machines de Minsky avec erreurs d'incrémentations, dans lesquelles les compteurs peuvent être incrémentés de façon non déterministes lors des franchissements de transitions. Il s'avère qu'autoriser des erreurs d'incrémentation permet d'obtenir la décidabilité du problème de l'arrêt des machines de Minsky à deux compteurs. Pour prouver cela, on utilise un résultat sur les machines à canaux, c'est-à-dire des systèmes qui peuvent s'échanger des messages à travers des canaux. En effet, le problème d'accessibilité d'un état de contrôle est indécidable pour les machines à canaux, mais lorsque les canaux sont non fiables, c'est-à-dire lorsque que l'on suppose que des messages peuvent être perdus, alors ce problème d'accessibilité devient décidable avec une complexité non primitive récursive, comme cela est prouvé dans [Sch02]. Dans [OW06], les auteurs étudient une variante de ce problème en supposant que des insertions et non plus des pertes, peuvent avoir lieu sur les canaux, le problème reste alors toujours décidable avec la même complexité que dans le cas des pertes. Par la suite, dans [DL06], les auteurs prouvent que le problème de l'arrêt des machines de Minsky avec erreur d'incrémentation peut être réduit aux problème de satisfiabilité fini de formules de LTL avec un registre. Nous allons prouver maintenant que si nous considérons le problème de model-checking existentiel introduit précédemment sur les automates à un compteur plutôt que le problème de satisfiabilité, alors nous pouvons raffiner la réduction proposée dans [DL06] de façon à exclure les exécutions réalisant des erreurs d'incrémentation. Plus précisément, dans la réduction que l'on trouve dans [DL06], les auteurs n'étaient pas capables d'exclure les exécutions réalisant des erreurs d'incrémentation, car la logique LTL avec registres est trop faible pour pouvoir exprimer que pour chaque décrementation, la donnée l'étiquetant a été vue auparavant, ceci car il n'y a pas d'opérateur exprimant le passé dans cette logique. Comme nous allons le voir dans la preuve du théorème suivant, nous allons ainsi nous servir de l'automate à un compteur pour assurer que les erreurs d'incrémentation ne peuvent pas avoir lieu.

Théorème 3.28 *La restriction de* $\text{MC(LTL)}_1^{<\omega}$ *aux formules utilisant uniquement les opérateurs temporels* X *et* F *est indécidable.*

Preuve : La preuve consiste à réduire le problème d'accessibilité d'un état de contrôle pour les machines de Minsky à deux compteurs, qui est indécidable (cf théorème 1.37), au problème $\text{MC(LTL)}_1^{<\omega}$. Soit $S = \langle Q, \{x_1, x_2\}, E\rangle$ une machine de Minsky à deux compteurs munie d'une configuration initiale $c_0 = (q_0, \mathbf{v}_0)$. Nous considérerons de plus $q_F \in Q$ un état de contrôle de S de telle sorte que $q_0 \neq q_F$. Sans perte de généralité, nous supposerons que $\mathbf{v}_0(x_1) = \mathbf{v}_0(x_2) = 0$ et que toutes les transitions partant de l'état q_0 réalisent des incrémentations. Nous allons construire un automate à un compteur $\mathcal{A} = \langle Q', E', q_I, F\rangle$ et une formule ϕ de $\text{LTL}_1^{\downarrow,Q'}$ tels que S atteint l'état q_F à partir de la configuration initiale c_0 si et seulement si $\mathcal{A} \models^{<\omega} \phi$. La principale difficulté consiste à encoder le comportement de deux compteurs dans un unique compteur. Pour cela nous allons procéder de la façon suivante. Pour chaque exécution de $TS(S)$ de la forme :

$$\begin{pmatrix} q_0 \\ \mathbf{v}_0(x_1) \\ \mathbf{v}_0(x_2) \end{pmatrix} \xrightarrow{e_0} \begin{pmatrix} q_1 \\ \mathbf{v}_1(x_1) \\ \mathbf{v}_1(x_2) \end{pmatrix} \xrightarrow{e_1} \dots \begin{pmatrix} q_m \\ \mathbf{v}_m(x_1) \\ \mathbf{v}_m(x_2) \end{pmatrix}$$

nous associons une exécution de $TS(\mathcal{A})$ de la forme :

$$\begin{pmatrix} q_I \\ 0 \end{pmatrix} \rightarrow^* \begin{pmatrix} e_0 \\ n_1 \end{pmatrix} \rightarrow^* \begin{pmatrix} e_1 \\ n_2 \end{pmatrix} \dots \begin{pmatrix} e_{m-1} \\ n_m \end{pmatrix}$$

Ainsi chaque transition de la machine de Minsky à deux compteurs S sera aussi un état de contrôle de l'automate à un compteur $\mathcal{A}$, et nous allons contraindre les exécutions possibles de $\mathcal{A}$ en utilisant l'unique compteur et des formules de LTL avec un registre de façon à ce qu'elles représentent des exécutions possibles de (S, c_0). L'idée principale consiste à associer à chaque action possible dans $TS(S)$ un entier qui sert à tester par exemple que chaque décrémentation est précédée par une incrémentation ou encore que pour chaque test à zéro, il y a eu auparavant autant d'incrémentations que de décrémentations.

Nous construisons maintenant l'automate à un compteur $\mathcal{A}$. Pour simplifier la lecture de la preuve, une partie de la construction est donnée sous forme graphique. Ainsi nous avons $\mathcal{A} = \langle Q', E', q_I, F \rangle$ avec :

– Q' est l'ensemble des états de contrôle vérifiant :

$$\begin{aligned} Q' = \quad & E \uplus \{q_I\} \uplus \{i_0\} \\ & \uplus \{i_e^{last}, i_e^{\neg last} \mid e = (q, \mathbf{inc}(x_i), q') \in E\} \\ & \uplus \{d_e^{last}, d_e^{\neg last} \mid e = (q, \mathbf{dec}(x_i), q') \in E\} \\ & \uplus \{z_e^{down} \mid e = (q, \mathbf{ifzero}(x_i), q') \in E\} \\ & \uplus \{z_q \mid q \in Q\} \uplus Q_{aux} \end{aligned}$$

où Q_{aux} est un ensemble d'états auxiliaires que nous ne décrirons pas (mais que l'on peut cependant identifier comme les états de contrôle dans les figures 3.3, 3.4 et 3.5 qui n'ont pas d'étiquette),
– l'ensemble des états acceptants F' est égal à $\{z_{q_f}\}$,
– q_I est l'état initial,
– et en ce qui concerne la relation de transitions E', il s'agit de la plus petite relation, contenant les règles suivantes :
  – la relation de transition décrite par la figure 3.2 appartient à E',
  – pour chaque transition de la forme $(q, \mathbf{inc}(x_i), q')$, nous avons dans E' la relation décrite par la figure 3.3,
  – pour chaque transition de la forme $(q, \mathbf{dec}(x_i), q')$, nous avons dans E' la relation décrite par la figure 3.4,
  – pour chaque transition de la forme $(q, \mathbf{ifzero}(x_i), q')$, nous avons dans E' la relation décrite par la figure 3.5.

Avant de donner la formule ϕ de $\mathrm{LTL}_1^{\downarrow, Q'}$, nous introduisons quelques formules intermédiaires, pour chaque $i \in \{1, 2\}$:

– I_i est la disjonction de i_0 et de toutes les transitions e telles que e est une transition réalisant une incrémentation du compteur x_i, c'est-à-dire $I_i = i_0 \vee \bigvee_{\{e \in E \mid e = (q, \mathbf{inc}(x_i), q')\}} e$,
– D_i est la disjonction de i_0 et de toutes les transitions $e \in E$ telles que e est une transition réalisant un décrémentation du compteur x_i, c'est-à-dire $D_i = i_0 \vee \bigvee_{\{e \in E \mid e = (q, \mathbf{dec}(x_i), q')\}} e$,

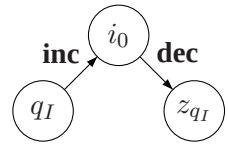


FIGURE 3.2 – Transitions initiales de E'

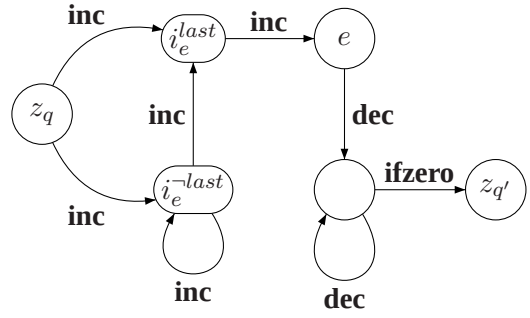


FIGURE 3.3 – Encodage des transitions de la forme $e = (q, \textbf{inc}(x_i), q')$

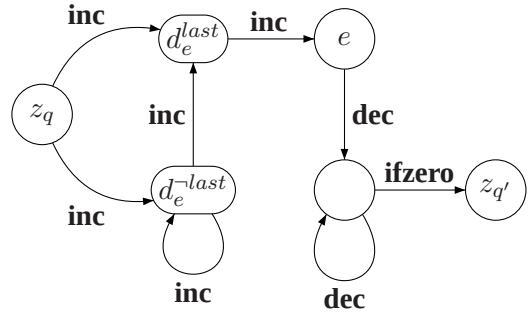


FIGURE 3.4 – Encodage des transitions de la forme $e = (q, \textbf{dec}(x_i), q')$

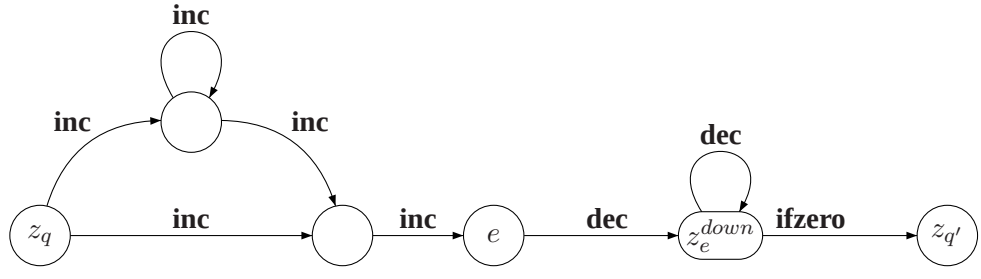


FIGURE 3.5 – Encodage des transitions de la forme $e = (q, \textbf{ifzero}(x_i), q')$

– I_i^{last} est la disjonction de tous les états de contrôle de la forme i_e^{last} où $e \in E$ est une transition réalisant une incrémentation du compteur x_i, c'est-à-dire $I_i^{last} = \bigvee_{\{e \in E | e=(q,\textbf{inc}(x_i),q')\}} i_e^{last}$,
– $I_i^{\neg last}$ est la disjonction de tous les états de contrôle de la forme $i_e^{\neg last}$ où $e \in E$ est une transition réalisant une incrémentation du compteur x_i, c'est-à-dire $I_i^{\neg last} = \bigvee_{\{e \in E | e=(q,\textbf{inc}(x_i),q')\}} i_e^{\neg last}$,
– D_i^{last} est la disjonction de tous les états de contrôle de la forme i_e^{last} où $e \in E$ est une transition réalisant une décrémentation du compteur x_i, c'est-à-dire $D_i^{last} = \bigvee_{\{e \in E | e=(q,\textbf{dec}(x_i),q')\}} d_e^{last}$,
– $D_i^{\neg last}$ est la disjonction de tous les états de contrôle de la forme $i_e^{\neg last}$ où $e \in E$ est une transition réalisant une décrémentation du compteur x_i, c'est-à-dire $D_i^{\neg last} = \bigvee_{\{e \in E | e=(q,\textbf{dec}(x_i),q')\}} d_e^{\neg last}$,
– Z_i est la disjonction de tous les états de contrôle de la forme e où $e \in E$ est une transition réalisant un test à zéro de x_i, c'est-à-dire $Z_i = \bigvee_{\{e \in E | e=(q,\textbf{ifzero}(x_i),q')\}} e$,
– Z_i^{down} est la disjonction de tous les états de contrôle de la forme z_e^{down} où $e \in E$ est une transition réalisant un test à zéro du compteur x_i, c'est-à-dire $Z_i^{down} = \bigvee_{\{e \in E | e=(q,\textbf{ifzero}(x_i),q')\}} z_e^{down}$.

Nous allons maintenant donner les formules de $\text{LTL}_1^{\downarrow,Q'}$ qui servent à contraindre les exécutions de l'automate à un compteur $\mathcal{A}$ de façon à faire correspondre ces exécutions contraintes avec les exécutions de (S, c_0). L'idée consiste à faire correspondre pour chacun des deux compteurs à chaque incrémentation un entier différent et à chaque décrémentation un entier différent. Nous souhaitons également que les entiers associés à chaque incrémentation évolue d'une unité à chaque fois, de même pour les entiers associées aux décrementations qui de plus ne doivent jamais être supérieurs à l'entier associé à la dernière incrémentation (on est ainsi sûr qu'il n'y a pas eu plus de décrementations que d'incrémentations). Pour réaliser le test à zéro, nous faisons évoluer la valeur du compteur au-dessus de toutes les valeurs utilisées jusqu'à présent pour les incrémentations, et ensuite la valeur du compteur redescend jusqu'à zéro en testant qu'à chaque incrémentation correspond une décrémentation.

Dans les formules que nous présentons, les opérateurs $\mathtt{G}^+$ et $\mathtt{F}^+$ sont des abréviations pour $\mathtt{XG}$ et $\mathtt{XF}$, de plus nous n'indiquons pas de registre en indice des opérateurs $\uparrow$ et $\downarrow$ car nous utilisons toujours le même registre. Voilà les différentes règles que nous allons utiliser, pour chaque $i \in \{1, 2\}$:

1. après chaque configuration satisfaisant I_i, il n'y a pas de configuration "strictement" dans le futur satisfaisant également I_i et ayant la même valeur de compteur :

$$\mathtt{G}\big(I_i \Rightarrow\downarrow \mathtt{G}^+(I_i \Rightarrow \neg \uparrow)\big)$$

2. après chaque configuration satisfaisant D_i, il n'y a pas de configuration "strictement" dans le futur satisfaisant également D_i et ayant la même valeur de compteur :

$$\mathtt{G}\big(D_i \Rightarrow\downarrow \mathtt{G}^+(D_i \Rightarrow \neg \uparrow)\big)$$

3. après chaque configuration satisfaisant D_i, il n'y a pas de configuration "strictement" dans le futur satisfaisant I_i et ayant la même valeur de compteur :

$$\mathtt{G}\big(D_i \Rightarrow\downarrow \mathtt{G}^+(I_i \Rightarrow \neg \uparrow)\big)$$

4. lorsque l'on a besoin d'une nouvelle valeur de compteur pour une incrémentation du compteur x_i, la nouvelle valeur choisie est égale à la plus haute valeur utilisée pour la dernière incrémentation de x_i augmentée de 1 :

$$\mathtt{G}\big(I_i \Rightarrow (\downarrow \mathtt{F}(I_i^{\neg last} \wedge \uparrow) \Rightarrow\downarrow \mathtt{F}(I_i^{last} \wedge \uparrow))\big) \wedge \mathtt{G}\big((I_i^{last} \vee I_i^{\neg last}) \Rightarrow\downarrow \mathtt{G}^+(I_i \Rightarrow \neg \uparrow)\big)$$

5. lorsque l'on a besoin d'une nouvelle valeur de compteur pour une décrémentation du compteur x_i, la nouvelle valeur choisie est égale à la plus haute valeur utilisée pour la dernière décrémentation de x_i augmentée de 1 :

$$\mathtt{G}\big(D_i \Rightarrow (\downarrow \mathtt{F}(D_i^{\neg last} \wedge \uparrow) \Rightarrow\downarrow \mathtt{F}(D_i^{last} \wedge \uparrow))\big) \wedge \mathtt{G}\big((D_i^{last} \vee D_i^{\neg last}) \Rightarrow\downarrow \mathtt{G}^{+}(D_i \Rightarrow \neg \uparrow)\big)$$

6. la valeur du compteur associée à une décrementation de x_i n'est jamais strictement plus grande que la valeur associée à la dernière incrémentation de x_i :

$$\begin{array}{l}\mathtt{G}\big(I_i \Rightarrow (\downarrow \mathtt{F}(D_i^{\neg last} \wedge \uparrow) \Rightarrow\downarrow \mathtt{F}(I_i^{last} \wedge \uparrow))\big)\\ \wedge\mathtt{G}\big(I_i \Rightarrow (\downarrow \mathtt{F}(D_i^{last} \wedge \uparrow) \Rightarrow\downarrow \mathtt{F}(I_i^{last} \wedge \uparrow))\big)\\ \wedge\mathtt{G}\big(D_i^{\neg last} \Rightarrow\downarrow \mathtt{G}^{+}(I_i^{last} \Rightarrow \neg \uparrow)\big)\end{array}$$

7. la valeur du compteur associée à l'état Z_i est toujours strictement plus grande que la valeur associée à la dernière incrémentation de x_i :

$$\mathtt{G}\big(I_i \Rightarrow\downarrow \mathtt{G}(Z_i \Rightarrow \neg \uparrow)\big)$$

8. lorsque $\mathcal{A}$ se trouve dans la pente descendante pour encoder un test à zéro, c'est-à-dire lorsque la formule Z_i^{down} est satisfaite, et qu'une valeur déjà utilisée pour une incrémentation est rencontrée, alors la méme valeur de compteur doit être utilisée avant pour une décrementation :

$$\begin{array}{l}\neg\mathtt{F}\big(I_i \wedge \downarrow \mathtt{F}(Z_i^{down} \wedge \uparrow) \wedge \neg \downarrow \mathtt{F}(\uparrow \wedge D_i)\big)\\ \wedge\neg\mathtt{F}\big(Z_i^{down} \wedge \downarrow \mathtt{F}(D_i \wedge \uparrow)\big)\end{array}$$

Nous résumons ici ce qu'impliquent les formules données précédemment sur les valeurs prises par le compteur dans les exécutions de $\mathcal{A}$ satisfaisant ces formules :

- Une nouvelle valeur de compteur associée à une incrémentation correspond toujours à la plus grande valeur utilisée jusqu'à présent pour une incrémentation à laquelle on ajoute 1 (règle 4.). La première valeur de compteur pour une incrémentation est de plus toujours 2.
- Une nouvelle valeur de compteur associée à une décrémentation est toujours la plus grande valeur utilisée jusqu'à présent pour une décrémentation à laquelle on ajoute 1 (règle 5.) et est toujours inférieure ou égale à la plus grande valeur utilisée jusqu'à présent pour une incrémentation (règle 6.). La première valeur de compteur pour une décrémentation est de plus toujours 2.
- Les tests à zéro sont réalisés de la façon suivante :
  - la valeur du compteur est incrémentée au-delà de toutes les valeurs de compteurs utilisées jusqu'à présent pour une incrementation (règle 7.)
  - on fait ensuite décroître le compteur jusqu'à 0 (encodé dans $\mathcal{A}$, cf figure 3.5) et à chaque valeur de compteur rencontrée qui est utilisée précédemment pour une incrémentation, on vérifie qu'il y a une décrementation qui lui est associée (règle 8.).

Afin de faciliter la compréhension, nous détaillons pourquoi la règle 6. permet d'assurer que la valeur du compteur associée à une décrémentation de x_i n'est jamais strictement plus grande que la valeur associée à la dernière incrémentation de x_i. Pour cela supposons que les formule des règles 1. à 6. soient vérifiées et que la dernière valeur associée à une décrémentation de x_i soit strictement plus grande que la valeur associée à la dernière incrémentation de x_i. Tout d'abord supposons que la valeur associée à la décrementation considérée soit plus grande d'une unité. Alors nous sommes dans le cas de la deuxième ligne de la formule de la règle 6., par conséquent il doit y avoir une incrémentation

avec la même valeur que la décrementation, et cette incrémentation a forcément lieu entre l'incrémentation considérée au début et la décrémentation d'après les règles 1. et 3.. Nous avons donc une contradiction car l'incrémentation considérée n'est pas la dernière. Supposons que la valeur associée à la décrementation considérée soit plus grande de k unités (avec $k > 1$). Nous sommes alors dans le cas de la première ligne de la règle 6., il existe donc une incrémentation après l'incrémentation considérée ayant la valeur supérieure à une unité, et la troisième ligne de la formule de la règle 6. nous indique que cette incrémentation est forcément avant la décrementation considérée, ce qui nous permet d'obtenir une contradiction.

La figure 3.6 nous donne un exemple du début d'une exécution de $\mathcal{A}$ respectant les règles énoncées et encodant la suite d'instructions **inc**(x_1), **inc**(x_1), **dec**(x_1), **dec**(x_1), **ifzero**(x_1). Cette suite d'instructions est effectivement réalisable et sur la figure, on voit que dans la pente après l'état satisfaisant Z_1, on fait correspondre à chaque coup une incrémentation avec une décrémentation.

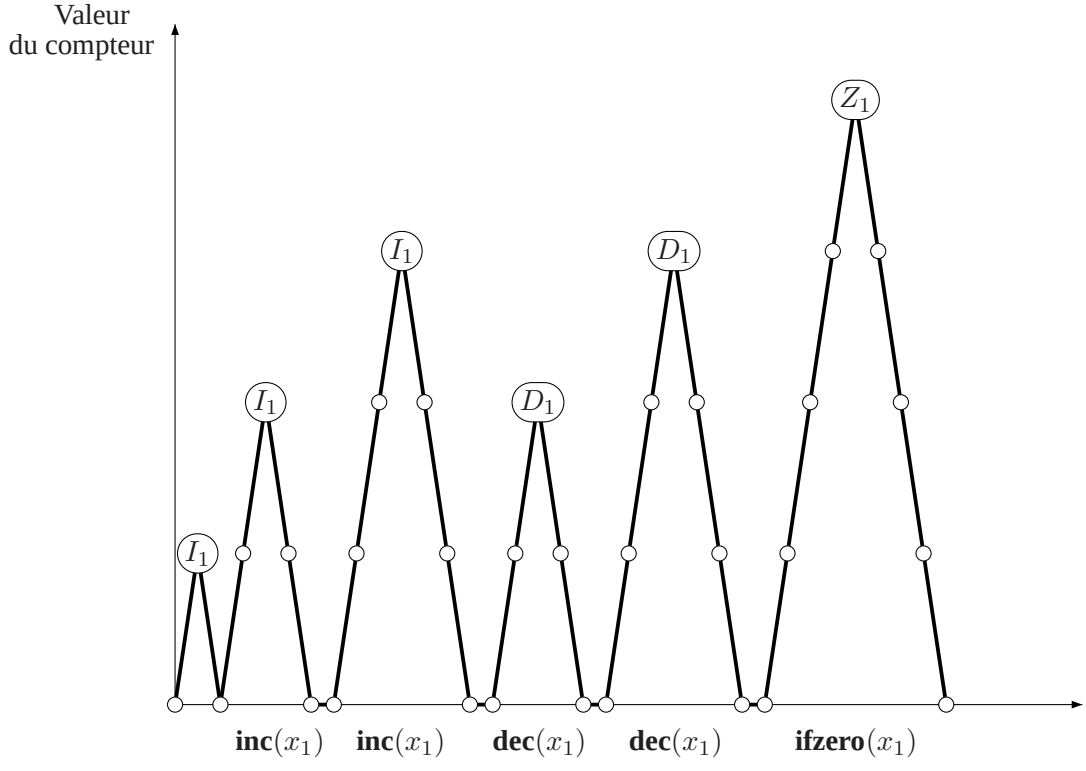


FIGURE 3.6 – Début d'une exécution de $\mathcal{A}$ satisfaisant les contraintes désirées

La formule ϕ que nous allons considérer est ainsi la conjonction pour chaque $i \in \{1, 2\}$ des formules exprimées dans les règles 1. à 8. à laquelle on ajoute une formule disant que l'état dans F' est visité. Nous allons maintenant nous attacher à montrer que $\mathcal{A} \models^{<\omega} \phi$ si et seulement si la machine de Minsky à deux compteurs S atteint l'état de contrôle q_F en partant de la configuration $(q_0, \mathbf{0})$.

Soit $\pi = (p_0, 0) \xrightarrow{e_0} (p_1, n_1) \xrightarrow{e_1} (p_2, n_2) \dots (p_m, n_m)$ une exécution finie de $\mathcal{A}$ vérifiant les règles 1. à 8. et telle que $p_0 = q_I$ et telle que $p_m = z_q$ pour $q \in Q$. Nous considérons la sous-suite d'indices

$i_1, \ldots, i_k \in [0..m]$ telle que pour tout $j \in [1..m]$, $p_{i_j} \in E$ et telle qu'il n'existe pas d'indice $i \in [1..m]$ tel que $p_i \in E$ et $i \notin \{i_1, \ldots, i_k\}$. Nous allons montrer que la suite de transitions de E, $p_{i_1} p_{i_2} \ldots p_{i_k}$ correspond à une trace de $TS(S)$, c'est-à-dire qu'il existe $c_1, c_2, \ldots c_k \in Q \times \mathbb{N}^2$ tels que dans $TS(S)$ nous avons $c_0 \xrightarrow{p_{i_1}} c_1 \xrightarrow{p_{i_2}} c_2 \ldots \xrightarrow{p_{i_k}} c_k$.

Nous raisonnons par induction sur k. Si $k = 1$, alors par construction de l'automate à un compteur $\mathcal{A}$, il existe $i \in \{1, 2\}$ et $q' \in Q$ tels que $p_{i_1} = (q_0, \mathbf{inc}(x_i), q')$. Comme une incrémentation peut toujours avoir lieu et comme $c_0 = (q_0, 0, 0)$, nous en déduisons qu'il existe effectivement $c_1 \in Q \times \mathbb{N}^2$ tel que $c_0 \xrightarrow{p_{i_1}} c_1$.

Supposons la propriété vraie pour k et montrons qu'elle est vraie pour $k + 1$.

Notons tout d'abord les propriétés vérifiées par la suite de configurations $(p_{i_0}, n_{i_0}), \ldots, (p_{i_k}, n_{i_k})$. Pour chaque $i \in \{1, 2\}$, on note Inc_i l'ensemble $\{j \in [1..k] \mid p_{i_j}$ est de la forme $(q, \mathbf{inc}(x_i), q')\}$ et Dec_i l'ensemble $\{j \in [1..k] \mid p_{i_j}$ est de la forme $(q, \mathbf{dec}(x_i), q')\}$. Soit $i \in \{1, 2\}$. La règle 1. garantit que pour tout $j \in Inc_i$, $n_{i_k} > 1$, et que pour tout $j, l \in Inc_i$, $n_{i_j} \neq n_{i_l}$, ceci car i_0 satisfait I_i et la valeur du compteur en i_0 est toujours 1 et pour tout $j \in Inc_i$, q_{i_j} satisfait I_i par définition. De plus la règle 4. implique que pour tout $j, l \in Inc_i$ tel que $j < l$, si il n'existe pas d'indice $j' \in Inc_i$ tel que $j < j' < l$, alors nécessairement $n_{i_l} = n_{i_j} + 1$ et si j est le plus petit indice de Inc_i alors $n_{i_j} = 2$. En effet, si j est le plus petit indice de Inc_i, n_{i_j} est forcément supérieure à 2 (car la valeur du compteur en i_0 est toujours 1), si maintenant n_{i_j} est strictement supérieure à 2, alors le système doit passer par un état vérifiant I_i^{last} ou $I_i^{\neg last}$ avec une valeur de compteur égale à 1, mais comme j est le plus petit indice de Inc_i, la règle 4. n'est pas satisfaite. Pour montrer l'autre propriété, il suffit de faire une induction et d'utiliser de nouveau la règle 4.. De la même façon, nous pouvons montrer les propriétés similaires sur l'ensemble Dec_i. Nous avons alors $\{n_{i_j} \mid j \in Inc_i\} = [2..|Inc_i| + 1]$ et $\{n_{i_j} \mid j \in Dec_i\} = [2..|Dec_i| + 1]$. Finalement, la règle 6. garantit que pour tout $j \in Dec_i$, il existe $l \in Inc_i$ tel que $i_l \leq i_j$ et $n_{i_j} \leq n_{i_l}$. De ces différentes propriétés on en déduit que nécessairement $|Dec_i| \leq |Inc_i|$.

Nous supposons que $p_{i_k} = (q, a, q')$. Par construction de $\mathcal{A}$, nous avons $p_{i_{k+1}} = (q', a', q'')$. Si a' est de la forme $\mathbf{inc}(x_i)$ alors la propriété est vérifiée car une incrémentation peut toujours être réalisée. Supposons que $a' = \mathbf{dec}(x_i)$ avec $i \in \{1, 2\}$. La seule façon pour que cette transition ne soit pas franchissable est que $|Dec_i| = |Inc_i|$, ce qui n'est pas possible car comme π satisfait les règles 1. à 8., nous avons $n_{i_{k+1}} = n_{i_H} + 1$ où H est le plus grand indice de $|Dec_i|$ et aussi il existe $h \in |Inc_i|$ tel que $i_h \leq i_{k+1}$ et $n_{i_{k+1}} \leq n_{i_h}$. Or si $|Dec_i| = |Inc_i|$, par les propriétés précédentes, on aurait aussi qu'il existe $j \in Dec_i$ tel que $n_{i_h} = n_{i_j}$ et par conséquent $n_{i_H} + 1 \leq n_{i_j}$ ce qui n'est pas possible (par définition de H). Si $a' = \mathbf{ifzero}(x_i)$, alors la seule façon pour que cette transition ne soit pas franchissable est que $|Inc_i| > |Dec_i|$, ce qui n'est pas possible. En effet, comme π satisfait les règles 1. à 8., d'après la règle 7. et les propriétés vérifiées par Inc_i, nous avons pour tout $j \in Inc_i$, $n_{i_j} < n_{i_{k+1}}$. Après cette i_{k+1}-ème configuration, les $n_{i_{k+1}}$ configurations suivantes sont étiquetées par un état satisfaisant Z_i^{down}. Comme de plus $|Inc_i| > |Dec_i|$ cela signifie qu'il existe $h \in Inc_i$ tel que pour tout $j \in Dec_i$, $n_{i_j} < n_{i_h}$ et il existe également $l \in [i_{k+1}..i_{k+1} + n_{i_{k+1}}]$ tel que p_l satisfait Z_i^{down} et $n_l = n_h$, et ceci contredit la règle 8..

Nous en déduisons que si π est une exécution finie de $\mathcal{A}$ satisfaisant les règles 1. à 8. et visitant z_{q_F} alors la machine de Minsky S partant de la configuration c_0 visite l'état q_F.

Nous considérons maintenant une exécution de $TS(S)$ de la forme $c_0 \xrightarrow{e_0} c_1 \xrightarrow{e_1} \ldots \xrightarrow{e_{h-1}} c_h$. Nous construisons alors une exécution de l'automate à un compteur $\mathcal{A}$, $(p_0, 0) \rightarrow (p_1, n_1) \rightarrow \ldots \rightarrow (p_m, n_m)$ avec $p_0 = q_I$ et $p_m = z_q$ pour un $q \in Q$. Nous considérons la sous-suite d'indices $i_0, \ldots, i_k \in [0..m]$ telle que pour tout $j \in [1..m]$, $p_{i_j} \in E$ et telle qu'il n'existe pas d'indices

$i \in [1..m]$ tel que $p_i \in E$ et $i \notin \{i_1, \ldots, i_k\}$. Pour $i \in \{1, 2\}$, nous notons de plus Inc_i l'ensemble $\{j \in [0..k] \mid p_{i_j}$ est de la forme $(q, \textbf{inc}(x_i), q')\}$, de la même façon Dec_i est l'ensemble $\{j \in [0..k] \mid p_{i_j}$ est de la forme $(q, \textbf{dec}(x_i), q')\}$ et $Zero_i = \{j \in [0..k] \mid p_{i_j}$ est de la forme $(q, \textbf{ifzero}(x_i), q')\}$. Nous construisons alors l'exécution π de façon à ce que les propriétés suivantes soient vérifiées :

(a) $k = h - 1$ et pour tout $j \in [0..k]$, $p_{i_j} = e_j$,

(b) si j est le plus petit indice de Inc_i alors $n_{i_j} = 2$,

(c) si j est le plus petit indice de Dec_i alors $n_{i_j} = 2$,

(d) pour tout $j, l \in Inc_i$ tel que $j < l$, si il n'existe pas d'indice $j' \in Inc_i$ tel que $j < j' < l$, alors nécessairement $n_{i_l} = n_{i_j} + 1$,

(e) pour tout $j, l \in Dec_i$ tel que $j < l$, si il n'existe pas d'indice $j' \in Inc_i$ tel que $j < j' < l$, alors nécessairement $n_{i_l} = n_{i_j} + 1$,

(f) pour tout $j \in Dec_i$, il existe $l \in Inc_i$ tel que $i_l < i_j$ et $n_{i_j} \leq n_{i_l}$,

(g) pour tout $j \in Zero_i$, pour tout $l \in Inc_i$ tel que $i_l < i_j$, $n_{i_l} < n_{i_j}$ et il existe $m \in Inc_i$ tel que $i_m < i_j$ et $n_{i_j} = n_{i_m} + 1$.

D'abord remarquons que par définition de $\mathcal{A}$, il est possible de construire une exécution π de $\mathcal{A}$ vérifiant les propriétés exposées précédemment. Supposons maintenant que π soit une exécution de $\mathcal{A}$ vérifiant ces propriétés et montrons que π satisfait les règles 1. à 8..Les règles 1. et 2. sont satisfaites car tous les éléments de Inc_i et de Dec_i sont bien différents. La règle 3. est satisfaire grâce aux propriétés (e) et (f). La règle 4. est satisfaite, car si l'on est dans une position i_j avec $j \in Inc_i$ et si il existe une position l dans le futur satisfaisant $I_i^{\neg last}$ alors il existe une position $i_{j'}$ tel que $l < i_{j'}$ avec $j' \in Inc_i$ et $n_{i_{j'}} > n_{i_j} + 1$ (par construction de $\mathcal{A}$) et la règle (d) et la définition de $\mathcal{A}$ nous indique qu'il existe nécessairement une position m tel que $i_j < m < l$ qui satisfait I_i^{last} et tel que $n_m = n_{i_j}$ et q_{m+1} satisfait I_i et $n_{m+1} = n_{i_j} + 1$. De la même façon, on prouve que la règle 5. est satisfaite en utilisant les propriétés (c) et (e). La règle 6. est vérifiée grâce à la propriété (f) et pour finir les règles 7. et 8. sont vérifiées grâce à la propriété (g) et aux propriétés des ensembles Inc_i et Dec_i. Ainsi si il y a une exécution de $TS(S)$ partant de c_0 et visitant q_F, nous pouvons construire une exécution finie de π tel que $\pi \models^{<\omega} \phi$.

Nous notons de plus que la formule ϕ construite n'utilise que des opérateurs temporels de la forme `X` ou `F` (l'opérateur `G` pouvant être facilement obtenu à partir de `F`). □

Pour montrer le résultat précédent dans le cas du model-checking infini, au lieu de réduire le problème de l'accessibilité d'un état de contrôle dans une machine de Minsky à 2 compteurs, nous réduisons le problème d'accessibilité répétée pour les mêmes machines, qui consistent à savoir si, étant donné un état de contrôle, il existe une exécution passant infiniment souvent par cette état de contrôle. Ce problème est également indécidable pour les machines de Minsky à 2 compteurs. Nous obtenons alors le théorème suivant :

Théorème 3.29 *La restriction de* MC(LTL)$_1^\omega$ *aux formules utilisant uniquement les opérateurs temporels* `X` *et* `F` *est indécidable.*

De plus, en utilisant le lemme de purification (lemme 3.16), on en déduit que ces résultats d'indécidabilité restent vrais si l'on considère des formules parlant uniquement des données. Et donc :

Théorème 3.30 *Les restrictions de* PureMC(LTL)$_1^{<\omega}$ *et de* PureMC(LTL)$_1^\omega$ *aux formules utilisant uniquement les opérateurs temporels* `X` *et* `F` *sont indécidables.*

Finalement, nous montrons que ces résultats d'indécidabilité sont encore valables lorsque l'on considère la logique du premier ordre sur des mots de données, même dans le cas où les formules utilisées ne contiennent que 2 variables, ce qui contraste avec le fait que les problèmes de satisfiabilité fini et infini pour les formules de FO$(\sim, <, +1)$ utilisant au plus 2 variables sont décidables (cf. théorème 3.12). Pour prouver cela, nous faisons appel à un résultat énoncé dans [DL08] stipulant que pour toute formule de $\text{LTL}_1^{\downarrow,\Sigma}$ n'utilisant que les opérateurs `X` et `XF` et telle que chaque opérateur temporel est immédiatement précédé par un quantificateur $\downarrow_1$ et telle qu'il n'y a pas d'autre occurrence de quantificateur $\downarrow_1$, il existe une formule de $\text{FO2}^{\Sigma}(\sim, <, +1)$ équivalente. En regardant la preuve du théorème 3.28, on constate que la formule ϕ de $\text{LTL}_1^{\downarrow,Q'}$ peut être écrite de façon à satisfaire cette dernière propriété.

Théorème 3.31

1. *Les problèmes* $\text{MC(FO)}_2^{<\omega}$ *et* MC(FO)_2^{ω} *sont indécidables.*
2. *Les problèmes* $\text{PureMC(FO)}_4^{<\omega}$ *et* $\text{PureMC(FO)}_4^{\omega}$ *sont indécidables.*

Preuve : Pour prouver le point (1.), on utilise la constatation précédent l'énoncé du théorème et le résultat des théorèmes 3.28et 3.29. Le point (2.) se déduit ensuite du point (1.) en utilisant le lemme 3.17. □

Nous finissons par montrer que les résultats d'indécidabilité énoncés précédemment restent vrais même si nous restreignons les automates à un compteur pris en compte, en supposant qu'à partir de chaque état de contrôle, il n'y a jamais deux transitions sortantes différentes étiquetées avec la même action. Nous introduisons ainsi les automates à un compteur faiblement déterministes, qui correspondent à des automates finis déterministes lorsque l'alphabet prise en compte est $\{\textbf{inc}, \textbf{dec}, \textbf{ifzero}\}$. En effet, un automate à un compteur $\mathcal{A} = \langle Q, E, q_I, F\rangle$ est dit faiblement déterministe, si pour toute état de contrôle $q \in Q$, si il y a deux transitions $(q, a, q'), (q, a', q'') \in E$ alors $a = a'$ implique $q' = q''$. Remarquons que contrairement aux automates à un compteur déterministe, le système de transitions associé à un automate à un compteur faiblement déterministe n'est pas nécessairement déterministe.

Théorème 3.32 *Les problèmes* $\text{PureMC(LTL)}_1^{<\omega}$ *et* $\text{PureMC(LTL)}_1^{\omega}$ *restreints aux automates à un compteur faiblement déterministes sont indécidables.*

Preuve : Dans la preuve du lemme de purification 3.16, le caractère faiblement déterministe est préservé. Il nous suffit donc de montrer que étant donné un automate à un compteur $\mathcal{A} = \langle Q, E, q_I, F\rangle$ est une formule fermée ϕ de $\text{LTL}_1^{\downarrow,Q}$, on peut construire un automate faiblement déterministe $\mathcal{A}' = \langle Q', E', q_I, F\rangle$ et une formule fermée ϕ' de $\text{LTL}_1^{\downarrow,Q'}$ telle que $\mathcal{A} \models^{<\omega} \phi$ si et seulement si $\mathcal{A}' \models^{<\omega} \phi'$ et $\mathcal{A} \models^{\omega} \phi$ si et seulement si $\mathcal{A}' \models^{\omega} \phi'$.

La figure 3.7 illustre sur un ensemble exhaustif d'exemples comment à partir d'un automate à un compteur quelconque on obtient un automate à un compteur faiblement déterministe. Les constructions que nous donnons peuvent facilement être généralisées pour traiter toutes les transitions de $\mathcal{A}$. Nous constatons que nous avons $Q \subseteq Q'$. La formule ϕ' est quant à elle obtenue en appliquant la transformation T à ϕ, T étant définie par induction de la façon suivante :

– $T(q) = q$ pour tout $q \in Q$,

– $T(\uparrow_1) = \uparrow_1$,
– $T(\neg\phi) = \neg T(\phi)$,
– $T(\phi \wedge \phi') = T(\phi) \wedge T(\phi')$,
– $T(\mathtt{X}\phi) = \mathtt{X}\big((\neg \bigvee_{q\in Q} q)\mathtt{U}(\bigvee_{q\in Q} q \wedge T(\phi))\big)$,
– $T(\phi \mathtt{U} \phi') = \big((\bigvee_{q\in Q} q) \Rightarrow T(\phi)\big)\mathtt{U}\big(\bigvee_{q\in Q} q \wedge T(\phi')\big)$.
– $T(\downarrow_1 \phi) = \downarrow_1 T(\phi)$

Par la façon dont $\mathcal{A}'$ est construit, on vérifie facilement que ϕ' et $\mathcal{A}'$ ont les propriétés souhaitées. □

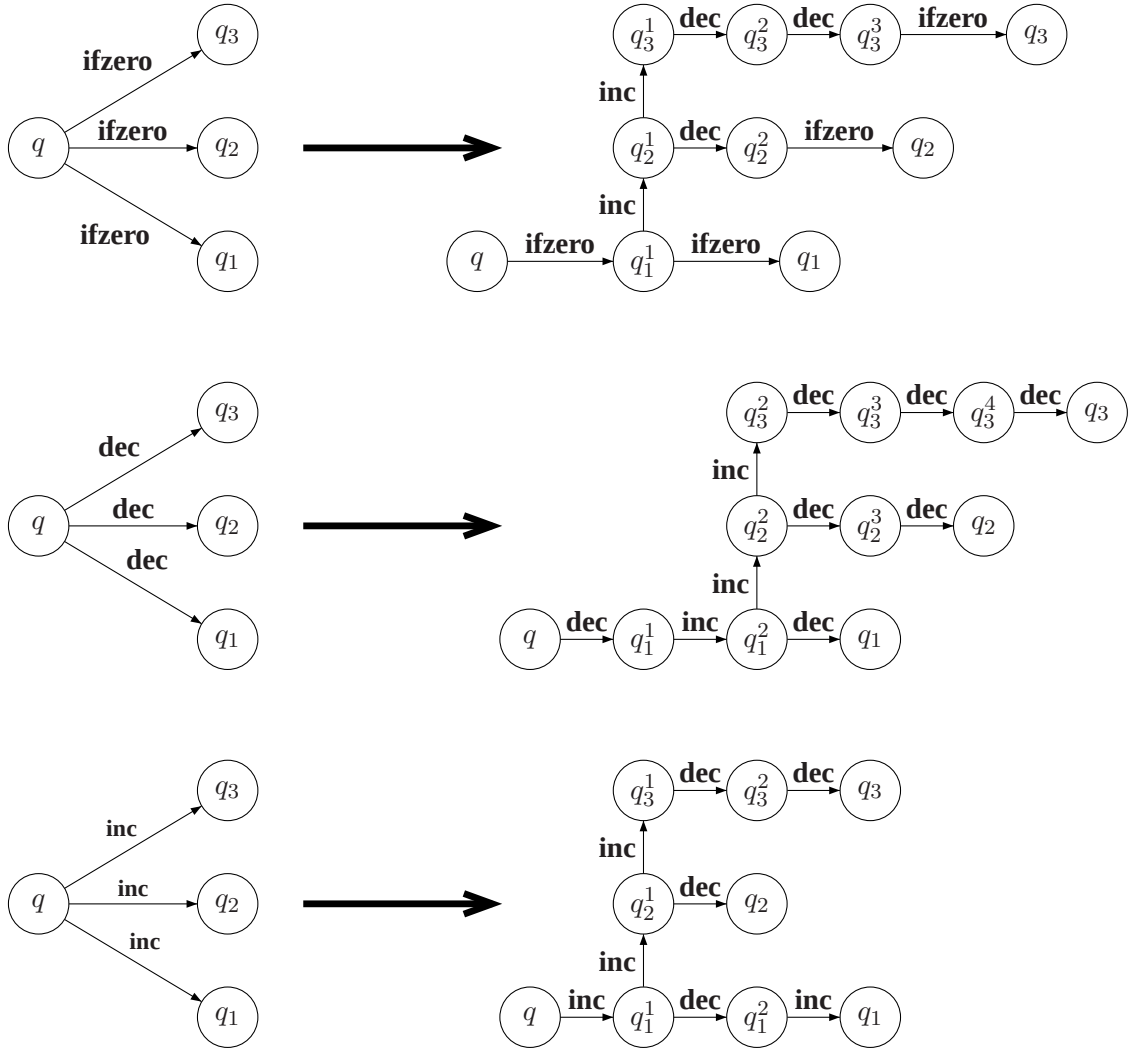


FIGURE 3.7 – Comment rendre un automate à un compteur faiblement déterministe

Nous avons également un résultat similaire pour la logique FO($\sim, <, +1$) obtenue grâce à la traduction de LTL$^{\downarrow}$ vers FO($\sim, <, +1$) donnée par la proposition 3.14 :

Corollaire 3.33 *Les problèmes* PureMC(FO)$_4^{<\omega}$ *et* PureMC(FO)$_4^{\omega}$ *restreints aux automates à un compteur faiblement déterministes sont indécidables.*

<table>
<tr><th></th><th>Automates à un compteur déterministes</th><th>Automates à un compteur non déterministes</th><th>Automates à un compteur faiblement déterministes</th></tr>
<tr><td>MC(LTL)$^{\omega}$</td><td rowspan="7">PSPACE-complet</td><td colspan="2" rowspan="13">Indécidable</td></tr>
<tr><td>MC(LTL)$^{<\omega}$</td></tr>
<tr><td>PureMC(LTL)$^{\omega}$</td></tr>
<tr><td>PureMC(LTL)$^{<\omega}$</td></tr>
<tr><td>MC(FO)$^{\omega}$</td></tr>
<tr><td>MC(FO)$^{<\omega}$</td></tr>
<tr><td>PureMC(FO)$^{<\omega}$</td></tr>
<tr><td>MC(LTL)$^{\omega}_1$</td><td rowspan="6">PSPACE</td></tr>
<tr><td>MC(LTL)$^{<\omega}_1$</td></tr>
<tr><td>PureMC(LTL)$^{\omega}_1$</td></tr>
<tr><td>PureMC(LTL)$^{<\omega}_1$</td></tr>
<tr><td>PureMC(FO)$^{\omega}_4$</td></tr>
<tr><td>PureMC(FO)$^{<\omega}_4$</td></tr>
<tr><td>MC(FO)$^{\omega}_2$</td><td rowspan="2">?</td></tr>
<tr><td>MC(FO)$^{<\omega}_2$</td></tr>
</table>

FIGURE 3.8 – Tableau récapitulatif

Ainsi ce derniers résultat nous montrent que même en restreignant les automates à un compteur, nous n'obtenons pas la décidabilité.

3.2.8 Tableau récapitulatif des résultats obtenus

Le tableau de la figure 3.8 récapitule les différents résultats que nous avons établis concernant les problèmes de model-checking de la logique LTL avec registres et de la logique du premier ordre sur des mots de données lorsque les modèles considérés sont des automates à un compteur. Remarquons que le seul cas pour lequel nous n'avons pas le résultat concerne le model-checking de formules de la logique du premier ordre sur les mots de données pour les automates à un compteur faiblement déterministes lorsque les formules considérées n'utilisent que deux variables.

Conclusion

Dans ce chapitre, nous avons étudié des problèmes de model-checking sur des systèmes à compteurs en considérant différentes logiques.
La première logique que nous avons considérée est une extension de CTL* pour laquelle les propositions atomiques sont des formules de Presburger permettant de caractériser les configurations d'un système à compteurs. Nous avons vu que lorsque les systèmes à compteurs sont linéaires, plats et à monoïde fini, le model-checking de cette logique est décidable. Nous avons également montré que cela n'est plus le cas lorsque nous prenons en compte les machines à compteurs *reversal*-bornées introduites au chapitre précédent, mais qu'il était cependant possible de poser des restrictions sur les formules de façon à obtenir également la décidabilité de certains problèmes pour ce modèle.

Ensuite, nous avons étudié les problèmes de model-checking de logiques sur des mots de données en prenant comme modèle des automates à un compteur. Les exécutions d'un automate à un compteur peuvent en effet être vues comme des mots de données, la donnée étant la valeur prise par le compteur à chaque pas. Malgré le fait que les automates à un compteur soient un modèle très simple, nous obtenons des résultats d'indécidabilité pour les problèmes de model-checking, sauf dans le cas où nous prenons des automates à un compteur déterministes. La méthode que nous proposons permet d'ouvrir la voie pour étudier d'autres problèmes de model-checking en considérant des modèles autres que les automates à un compteur et pour lesquels les problèmes d'accessibilité sont connus pour être décidables. Il pourrait en particulier être intéressant de voir si ces problèmes de model-checking sont décidables pour les machines à compteurs *reversal*-bornées. Nous avons de plus vu que les logiques considérées sont très expressives, une solution pour obtenir la décidabilité dans plus de cas pourrait être de restreindre la syntaxe des formules utilisées.

Deuxième partie

Vérification de programmes avec pointeurs

Chapitre 4

Un modèle pour vérifier les programmes avec pointeurs

Dans ce chapitre, nous présentons les problématiques liées à la vérification de programmes manipulant dynamiquement la mémoire. Puis, nous donnons un état de l'art sur les différentes techniques existantes pour résoudre ces problèmes. Dans un deuxième temps, nous définissons le modèle dont nous nous servirons pour modéliser des programmes manipulant des structures à un sélecteur (structures également appelées listes simplement chaînées). Nous appelons ce modèle les systèmes à pointeurs. Nous finissons par proposer une représentation symbolique pour caractériser de façon finie des ensembles infinis de configurations de systèmes à pointeurs.

4.1 Introduction du problème

4.1.1 Vérification de programmes manipulant dynamiquement la mémoire

Dans un système informatique, les sources d'erreur sont multiples. Pour n'en citer que quelques-unes : les débordements d'indices de tableaux, l'utilisation de variables ou pointeurs non initialisés, les divisions par zéro, etc. Ces erreurs peuvent être difficiles à détecter (manuellement ou automatiquement) car elles sont souvent liées au comportement dynamique du programme, c'est en particulier le cas des erreurs induites par l'utilisation des mécanismes d'allocation dynamique de la mémoire.

Dans cette partie, nous nous intéressons aux programmes pouvant manipuler explicitement la mémoire. Ces programmes peuvent réserver de façon dynamique des emplacements de la mémoire pour y stocker des données. Les zones réservées sont accessibles au programme grâce à des variables stockant l'adresse de ces zones, ces variables sont communément appelées des pointeurs. L'opération qui consiste à lire une donnée se trouvant dans une zone mémoire pointée par une variable se nomme le déréférencement. La plupart des programmes manipulant explicitement la mémoire utilisent des structures de données comme par exemple, les listes simplement chaînées, les listes doublement chaînées, les arbres, etc. Le principe de ces structures de données est le suivant : elles sont composées de cellules et chacune de ces cellules contient des données mais aussi les adresses d'autres cellules, par exemple dans le cas d'une liste simplement chaînée, chaque cellule contient l'adresse de la cellule suivante, dans le cas d'une liste doublement chaînée, chaque cellule contient l'adresse de la cellule précédente et celle de la cellule suivante. Ces structures de données sont très utiles car elles peuvent avoir un nombre non borné de cellules (contrairement aux tableaux statiques par exemple). Cependant

cet avantage des structures de données dynamiques fait que les programmes les utilisant sont, comme nous le verrons par la suite, difficiles à analyser et susceptibles de réaliser des erreurs dans la manipulation des différentes adresses mémoire. Nous attirons l'attention du lecteur sur le point suivant, dans la plupart des logiciels critiques utilisés à l'heure actuelle, la manipulation dynamique de mémoire a été bannie, à cause du risque d'erreur que ce genre de programmation engendre. C'est ainsi le cas pour la programmation de certaines applications que l'on trouve dans les avions, les fusées ou encore les centrales nucléaires. Néanmoins l'allocation dynamique de mémoire et les structures de données qui y sont associées permettent souvent d'améliorer les performances d'un système ainsi que la manipulation des données, d'où l'intérêt de développer des méthodes permettant de vérifier de façon automatique de tels programmes.

Parmi les différents langages de programmation offrant des mécanismes de manipulation de la mémoire, nous prenons ici comme référence le langage C, ceci car il s'agit d'un langage bas niveau proposant au développeur de nombreuses fonctionnalités permettant de manipuler explicitement les adresses des zones mémoire. Notons que d'autres langages de programmation offrent également cette possibilité et que les techniques que nous proposons peuvent facilement être adaptées à ces autres langages. Nous avons choisi le langage C car de nombreux programmes manipulant explicitement la mémoire sont écrits dans ce langage. À titre d'indication, nous rappelons que, en C , l'allocation de la mémoire se fait grâce à la primitive `malloc` qui renvoie l'adresse de la zone mémoire allouée, quant à la libération d'une zone mémoire, elle est réalisée grâce à la fonction `free`. Il existe de plus en C une adresse référence que nous utiliserons et qui est l'adresse `NULL`. Cette adresse permet d'avoir un point d'ancrage. Par exemple pour marquer le dernier élément d'une liste, on met dans la dernière cellule l'adresse `NULL` ce qui permet ensuite de tester si le programme a atteint ou non la fin de la liste.

Nous donnons maintenant quelques exemples de propriétés que l'on souhaite vérifier sur les programmes allouant dynamiquement la mémoire :

1. **Absence d'erreur de segmentation** L'erreur de segmentation parvient lorsque le programme tente d'accéder à une zone qui n'a pas été allouée ou lorsque le programme tente de déréférencer le pointeur `NULL` ; cette erreur est une des plus importantes car elle peut endommager complètement le système, si par exemple le programme écrit ou efface des zones qui sont réservées pour d'autres programmes.
2. **Absence de fuite mémoire** Une fuite mémoire arrive lorsqu'une zone mémoire qui a été allouée précédemment ne peut plus être accédée par aucun des pointeurs du programme ; lorsque cette erreur arrive, cela signifie que le programme a réservé une zone mémoire qui ne lui sert plus mais qui reste néanmoins bloquée. Cette erreur n'est pas aussi critique que l'erreur de segmentation, toutefois elle caractérise une mauvaise utilisation de l'allocation dynamique. De plus dans certains logiciels embarqués, la mémoire est une ressource critique, dont il est important de contrôler parfaitement la façon dont elle est allouée et libérée.
3. **Respect des invariants structurels** Cette dernière classe de propriétés est liée à l'utilisation de structures de données. En effet, lorsqu'un développeur implante une structure de données, le bon fonctionnement des algorithmes est fortement lié à la "forme" de la mémoire donnée en entrée, ainsi certaines fonctions sur les listes simplement chaînées ne fonctionnent plus correctement si la liste donnée en entrée est cyclique, il est donc important pour assurer le bon fonctionnement d'un système, de s'assurer que certains invariants sont respectés pour les structures de données utilisées. Nous distinguons deux types de propriétés, les propriétés dites qualitatives portant sur la forme générale de la mémoire, comme par exemple, le partage de listes, ou la présence de

listes cycliques, et les propriétés quantitatives qui considèrent également le nombre de cellules allouées, par exemple savoir si une liste a la même longueur qu'une autre liste.

4.1.2 État de l'art

La vérification de programmes manipulant dynamiquement la mémoire est un champ de recherche très actif et pour lequel de nombreuses techniques ont été développées. Nous rappelons dans cette partie certaines d'entre elles afin de positionner ensuite notre recherche.

4.1.2.1 L'analyse de forme ("shape analysis") : un point de départ

La plupart des méthodes développées pour vérifier des programmes manipulant dynamiquement la mémoire sont basées sur ce que l'on appelle l'analyse de forme ("shape analysis" en anglais). Le problème de l'analyse de forme consiste à déterminer pour chaque point d'un programme donné une caractérisation finie des différentes "formes" que les structures de données manipulées peuvent prendre en mémoire. L'analyse de forme a été introduite par Reynolds dans [Rey68] qui l'a étudiée pour analyser des programmes dans un langage proche du Lisp qui ne modifiaient pas la forme générale des structures de donnés, mais qui pouvaient uniquement changer les emplacements pointés par les variables. La plupart des méthodes basées sur l'analyse de forme cherchent à établir un modèle fini pour représenter un ensemble infini de configurations de la mémoire et calculent une surapproximation du comportement du programme sur ces représentations.

4.1.2.2 Méthodes à base de graphes

Basée sur l'interprétation abstraite. Dans [SRW98], les auteurs proposent un algorithme qui utilise un ensemble fini de graphes spéciaux (appelés "shape graphs") pour approximer les différentes formes que les structures allouées peuvent prendre dans la mémoire lors de l'exécution du programme. Leur algorithme est basé sur la technique de l'interprétation abstraite introduite par Patrick Cousot et Radhia Cousot dans [CC77]. L'interprétation abstraite est une théorie d'approximation de la sémantique de programmes basée sur l'utilisation de fonctions monotones sur des ensembles ordonnées. Cette technique permet de calculer une surapproximation du comportement réel des programmes, il se peut qu'elle détecte des erreurs qui sont en fait des fausses alarmes, c'est-à-dire des erreurs liées à l'approximation qui n'ont pas réellement lieu lors de l'exécution du programme. Les "shape graphs" introduits dans [SRW98] permettent de représenter de façon bornée des tas mémoire dont le nombre de cellules est non borné en regroupant les cellules mémoire qui ne sont pas pointées par des variables au sein d'une même cellule abstraite tandis que les cellules concrètes qui sont pointées par des ensembles de variables différents sont séparées dans des cellules abstraites différentes. Pour certains programmes, cette technique est capable de déterminer des propriétés telles que :

– lorsque la donnée d'entrée d'un programme est une liste simplement chaînée, la donnée de sortie est aussi une liste simplement chaînée,
– lorsque la donnée d'entrée d'un programme est un arbre, la donnée de sortie est aussi un arbre.

Vérification de structures de données paramétrées. Dans [DEG06], les auteurs s'intéressent au problème suivant : étant donnés un programme et une propriété de spécification sur les graphes initiaux, vérifier si les graphes obtenus en sortie du programme vérifient une propriété et ceci quelle que soit la taille des graphes d'entrée. Ils présentent alors une méthode automatique pour résoudre ce problème pour une classe de programmes. Les programmes pris en compte doivent en effet :

– terminer, et,
– modifier la forme générale de la structure de données qu'un nombre borné de fois.
Cet algorithme a l'avantage de fournir une réponse exacte sur le comportement du programme, en revanche les propriétés qui doivent être respectées par le programme sont en général indécidables.

4.1.2.3 Méthodes à base de logique

Logique à 3 valeurs. Dans [SRW02], Sagiv, Reps et Wilhelm poursuivent le travail qu'ils avaient commencé en utilisant l'interprétation abstraite sur les graphes (cf. section précédente). L'idée nouvelle qu'ils introduisent consiste à modéliser un graphe par des formules de la logique à 3 valeurs, les 3 valeurs étant *vrai*, *faux* et *indéterminé*. Cette logique a été introduite par Kleene dans [Kle87]. En évaluant les valeurs des formules, on peut :
– exécuter de façon abstraite une suite d'instructions sur les structures de données,
– extraire des propriétés du tas mémoire à partir d'une formule logique le caractérisant.
La logique à 3 valeurs est utilisée afin d'abstraire certaines propriétés du graphe représentant la mémoire, cela étant réalisé en rajoutant des prédicats d'abstraction dans les formules. Intuitivement, un ensemble possiblement infini de graphes est représenté par une structure contenant un ensemble fini de prédicats évalués à *vrai*, *faux* ou *indéterminé*. Ensuite étant donnée une formule de la logique du premier ordre utilisant ces prédicats, nous avons les propriétés suivantes :
– si cette formule est évaluée à *vrai*, alors tous les graphes mémoire décrits par la structure vérifient la formule,
– si cette formule est évaluée à *faux*, alors aucun des graphes mémoire décrits par la structure ne vérifie la formule,
– si cette formule est évaluée à *indéterminé*, il n'est pas possible de savoir si la formule est vérifiée par tous les graphes, aucun des graphes ou seulement certains graphes décrits par la structure.
Bien que cette méthode ait permis l'analyse de nombreux programmes manipulant dynamiquement la mémoire, elle comporte certains points faibles. Tout d'abord, étant donné qu'elle se base sur la logique à 3 valeurs, le résultat de l'algorithme d'analyse peut être imprécis. De plus, dans la plupart des cas, l'utilisateur doit fournir des prédicats d'abstraction pour raffiner l'analyse, il doit donc avoir une bonne connaissance du programme qu'il vérifie ainsi que de l'algorithme d'abstraction utilisé.
L'équipe de Sagiv a développé l'outil TVLA (Three-Valued Logic Analyser) basé sur la logique à 3 valeurs [LAMS04, BLARS07]. À partir de la définition dans une logique du premier ordre de la sémantique opérationnelle des instructions du programme et d'une formule logique représentant l'état abstrait initial, cet outil génère une sémantique abstraite et produit pour chaque point du programme une représentation abstraite du tas mémoire. Cet outil est disponible à l'adresse :
`http://www.cs.tau.ac.il/~tvla`

Logique monadique du second ordre. Dans [JJKS97], les auteurs proposent d'exprimer des propriétés sur le tas mémoire grâce à des formules de la logique monadique du second ordre sur des mots finis. La méthode utilisée permet de vérifier des triplets de Hoare sur des programmes ne comportant pas de boucle. Les triplets de Hoare sont des propriétés caractérisant un code informatique sous forme de la donnée de préconditions, postconditions et invariants. Dans cet article, les auteurs proposent une méthode pour analyser des programmes travaillant uniquement sur des listes simplement chaînées, et par la suite dans [EMS00] cette technique fut étendue aux programmes travaillant sur certains types d'arbres en considérant une logique monadique à k successeurs à la place d'une logique monadique à un successeur. Le type de structure de données pour lequel l'analyse fonctionne étant limité par la logique utilisée. L'avantage de cette méthode est que lorsqu'un triplet de Hoare n'est pas vérifié,

un contre-exemple est fourni qui permet de déduire la cause de l'erreur ou de raffiner le triplet de Hoare. En revanche, comme cette technique ne fonctionne pas pour les programmes avec boucles, l'utilisateur doit annoter chaque boucle du programme qu'il souhaite analyser. L'algorithme de l'outil PALE (Pointer Assertion Logic Engine) [MS01] utilise cette méthode. Cet outil est disponible à l'adresse : `http://www.brics.dk/PALE/`

Logique de séparation. La logique de séparation a été introduite par Reynolds dans [Rey02]. Les formules de la logique de séparation utilisent les prédicats standards de calcul des prédicats y compris les booléens et les quantificateurs, et elles admettent en plus quatre nouvelles assertions qui permettent de décrire le tas mémoire, en particulier pour décrire le tas vide, ou encore qu'un noeud contient l'adresse d'un autre noeud, ou bien que deux variables pointent à deux endroits différents de la mémoire. La logique de séparation permet de raisonner sur le tas mémoire de façon locale, ce qui peut présenter un avantage par rapport à la technique utilisant la logique à 3 valeurs, car la modification du tas mémoire après l'exécution d'une instruction peut être plus facile à réaliser symboliquement. Dans [BCO05b], les auteurs proposent une méthode pour réaliser l'exécution symbolique d'instructions sur certaines formules de logique de séparation ; grâce à cette méthode, ils peuvent vérifier des triplets de Hoare sur des programmes ne comportant pas de boucle. L'outil `Smallfoot` [BCO05a] implante cette idée. Cet outil est disponible à l'adresse suivante :
`http://www.dcs.qmul.ac.uk/research/logic/theory/projects/smallfoot/`.
En revanche, cette méthode n'est pas adaptée à la définition d'une sémantique abstraite car elle peut amener à construire une infinité de formules logique et il n'est donc pas possible de garantir un calcul de point fixe lors de l'analyse de programmes avec boucles. C'est pourquoi dans [DOY06], les auteurs présentent une adaptation de ce calcul symbolique en introduisant un opérateur d'abstraction qui transforme une formule de la logique de séparation en une certaine forme canonique. Ces formes canoniques sont définies de telle façon qu'il en existe un nombre borné. Ceci permet donc d'effectuer un calcul symbolique qui termine. Dans un premier temps, cette méthode ne fonctionnait que pour des programmes manipulant des listes simplement chaînées, mais elle fut étendue ensuite dans [BCC$^+$07] de façon à pouvoir vérifier des programmes travaillant sur des structures de données plus complexes comme des listes doublement chaînées ou des listes de listes. Là encore, une abstraction étant réalisée, l'algorithme de vérification calcule une surapproximation du comportement réel du programme et par conséquent il se peut que de fausses erreurs soient détectées.

Autres formalismes logique Dans [BIL04], les auteurs proposent une logique appelée "alias logic" utilisant des expressions régulières sur un alphabet fini pour modéliser le tas mémoire. Ils commencent par donner un formalisme logique permettant de décrire un grand nombre de propriétés mais il s'avère que le problème de satisfiabilité pour ce formalisme est indécidable. Ils trouvent ensuite une restriction décidable qui ne permet pas de décrire certaines propriétés intéressantes sur le tas mémoire, comme par exemple la présence d'une liste simplement chaînée. Les auteurs prouvent que leur formalisme logique permet de vérifier des programmes dans lesquels chaque boucle est annotée par un triplet de Hoare. Plus tard, dans [BI06], les auteurs ont modifié un peu cette logique pour ne considérer que des programmes manipulant des listes simplement chaînées, mais en autorisant dans la logique de parler de la longueur des listes. Ils développent ainsi une méthode de vérification de triplets de Hoare leur permettant de vérifier par exemple que deux listes ont la même longueur après l'exécution d'un nombre fini d'exécutions, alors qu'aucun des formalismes logiques vus auparavant ne permettait d'exprimer de telles propriétés.
Finalement, dans [PW05], les auteurs proposent un autre contexte symbolique appelés tas booléen

("boolean heap" en anglais) pour vérifier des programmes manipulant dynamiquement la mémoire. Comme pour la méthode utilisant la logique à 3 valeurs, les auteurs définissent un tas mémoire grâce à un ensemble de prédicats et ils appliquent ensuite la technique de l'abstraction de prédicats pour calculer une surapproximation des configurations du programme. Leur méthode consiste à choisir un ensemble de prédicats sur le tas mémoire (prédicats qui serviront à définir le domaine abstrait), puis à construire une fonction exécutant symboliquement les instructions du programme sur le domaine abstrait, et enfin à appliquer les techniques de model-checking classique pour des systèmes finis (l'abstraction assurant que le nombre de configurations abstraites possibles est fini). Un des avantages de cette dernière méthode est qu'elle peut bénéficier des techniques de l'abstraction de prédicats développées pour l'outil SLAM [BR01, BMMR01].

4.1.2.4 Méthodes à base de reconnaissance de langages

Model-checking régulier abstrait. Dans [BHMV05], les auteurs proposent une méthode basée sur le model-checking régulier (pour plus d'informations, on peut se référer à [WB98, BJNT00, KMM$^+$01]) pour analyser des programmes manipulant dynamiquement la mémoire. La technique qu'ils développent consiste à encoder les configurations d'un programme comme des mots et le programme devient alors un transducteur (c'est à dire un automate avec entrée/sortie) travaillant sur ces mots. De façon à encoder les listes dans des mots (ce qui peut poser problème par exemple dans le cas de listes partagées), ils utilisent des symboles spéciaux. Les ensembles infinis de configurations peuvent alors être représentés par des automates finis. Cependant l'algorithme du model-checking régulier ne permet pas toujours de calculer l'ensemble d'accessibilité exact (ceci car on n'arrive pas forcément à un point fixe). Aussi des techniques d'abstraction pour le model-checking régulier ont été développées (cf. [BHV04]). L'avantage est que l'on peut raffiner la surapproximation calculée de façon automatique grâce à des contre-exemples. Cette technique a été utilisée dans un premier temps pour vérifier des programmes manipulant uniquement des listes simplement chaînées [BHMV05] puis plus tard dans pour vérifier des programmes manipulant des arbres [BHRV06a, BHRV06b].

Automates d'arbres avec contraintes. Dans [HIV06], les auteurs introduisent une méthode pour vérifier des programmes manipulant des arbres équilibrés. Leur méthode utilisent des automates d'arbre avec en plus des contraintes de taille qui permettent de représenter des langages non réguliers. Avec cette technique ils arrivent à vérifier des triplets de Hoare, mais l'analyse de programmes avec boucles n'est pas faisable, sauf si bien entendu les boucles sont étiquetées avec des invariants.

4.1.2.5 Récapitulatif

Le tableau de la figure 4.1 résume les propriétés des différentes techniques présentées auparavant. Dans ce tableau, nous distinguons la vérification de propriétés qualitatives et de propriétés quantitatives. Nous appelons propriétés qualitatives toutes les propriétés concernant la forme générale du tas mémoire, comme par exemple présence ou non de listes cycliques, partage ou non de listes entre variables, mais sans qu'il y est d'indication sur le nombre de cellules concernées. Les propriétés quantitatives permettent quant à elles de donner des informations sur le nombre de cellules présentent dans le tas mémoire, par exemple le fait que deux listes aient la même longueur est une propriété quantitative. Nous tenons à faire cette différence sur les propriétés vérifiées car la méthode que nous présenterons dans le chapitre suivant permet de vérifier des propriétés qualitatives et quantitatives pour des programmes manipulant des listes simplement chaînées. Nous constatons avec ce tableau que la plupart des méthodes existantes soit calculent une surapproximation du comportement réel

du programme ce qui peut amener à la présence de fausses erreurs, soit ont besoin que les boucles des programmes fournis en entrée soient annotées avec des invariants. Il existe de plus très peu de méthodes permettant de vérifier des propriétés quantitatives.

Techniques	**Structures de données**	**Propriétés qualitatives**	**Propriétés quantitatives**	**Automatisation**
Graphes + Interprétation abstraite	Toutes	Oui	Non	Abstraction
Structures de données paramétrées	Toutes	Oui	Non	Restriction sur les programmes
TVLA	Toutes	Oui	Non	Abstraction
PALE	Restreintes	Oui	Non	Annotations
Logique de séparation	Listes de Listes	Oui	Non	Abstraction
Logique avec compteurs	Listes	Oui	Oui	Annotations
Tas booléens	Toutes	Oui	Non	Abstraction
Model-checking régulier	Arbres	Oui	Non	Abstraction
Automates d'arbres avec contraintes	Arbres équilibrés	Oui	Oui	Annotation

FIGURE 4.1 – Tableau récapitulatif des méthodes existantes

4.2 Systèmes à pointeurs

Dans cette section, nous présentons la façon dont nous modélisons un programme manipulant des listes simplement chaînées.

4.2.1 Modélisation du tas mémoire par des graphes

Nous nous intéressons aux programmes manipulant des listes simplement chaînées et nous ne prenons pas en compte les données stockées dans les différentes cellules des listes mais nous considérons uniquement la forme de la structure de données. Pour représenter la mémoire, on peut alors utiliser un graphe dans lequel chaque noeud représente une cellule et a au plus un successeur. Les noeuds peuvent de plus éventuellement être étiquetés (ou pointés) par des variables, appelées pointeurs. Nous introduisons de plus deux noeuds spéciaux qui sont les noeuds `null` pour représenter l'adresse `NULL` et le noeud $\perp$ dont nous nous servirons pour caractériser une zone mémoire indéfinie. Ainsi, nous définissons ce que nous appelons les graphes mémoire.

Definition 4.1 (Graphe mémoire) *[BFN04] Un graphe mémoire est un quadruplet $GM = (N, P, succ, loc)$ tel que :*

- *N est un ensemble fini de noeuds vérifiant $\{\texttt{null}, \perp\} \cap N = \emptyset$,*
- *P est un ensemble fini de variables appelées pointeurs,*
- *$succ : N \rightarrow N \cup \{\texttt{null}, \perp\}$ est une fonction associant à chaque noeud un noeud successeur,*
- *$loc : P \rightarrow N \cup \{\texttt{null}, \perp\}$ est une fonction associant un noeud à chaque pointeur,*
- *pour tout noeud $n \in N$, il existe $p \in P$ et $i \in \mathbb{N}$ tels que $n = succ^i(loc(p))$.*

Remarquons que la dernière condition exprime l'absence de fuite mémoire car elle implique que chaque noeud de la mémoire puisse être accédé par un pointeur. Nous notons $\mathcal{GM}_P$ l'ensemble des graphes mémoire ayant P comme ensemble de pointeurs. Soit GM est un graphe mémoire dans $\mathcal{GM}_P$ tel que $GM = (N, P, succ, loc)$. Si $n \in N$, nous définissons l'ensemble **List**$(GM, n) = \{m \in N \cup \{\texttt{null}, \perp\} \mid \exists i \in \mathbb{N}$ tel que $m = succ^i(n\}$. De plus nous avons **List**$(GM, \texttt{null}) = \{\texttt{null}\}$ et **List**$(GM, \perp) = \{\perp\}$. Si p est un pointeur de P, nous étendons cette définition de telle façon que **List**$(GM, p) =$ **List**$(GM, loc(p))$ est l'ensemble des noeuds de GM pouvant être accédés depuis le noeud pointé par p. Ainsi par définition d'un graphe mémoire, pour tout $n \in N$, il existe $p \in P$ tel que $n \in$ **List**(GM, p).

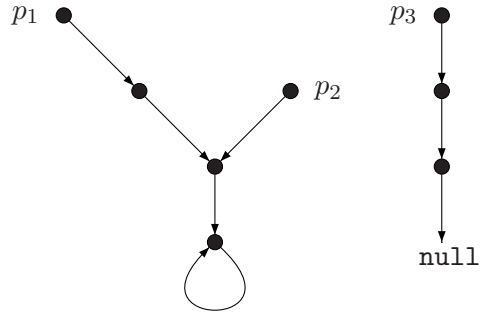


FIGURE 4.2 – Un graphe mémoire

Exemple 4.2 *La figure 4.2 donne une représentation d'un graphe mémoire avec trois pointeurs p_1, p_2 et p_3. Les listes pointées par p_1 et p_2 sont partagées et aboutissent sur une liste cyclique. Quant à la liste pointée par p_3, elle contient trois cellules et la dernière cellule pointe vers `NULL`.*

Notation 4.3 *Par la suite, lorsque l'ensemble des pointeurs P considéré sera explicite, nous ne le noterons pas dans $\mathcal{GM}_P$ en écrivant seulement $\mathcal{GM}$.*

Étant donnés deux graphes mémoire $GM_1 = (N_1, P, succ_1, loc_1)$ et $GM_2 = (N_2, P, succ_2, loc_2)$ dans $\mathcal{GM}_P$, nous dirons qu'une fonction $f : N_1 \cup \{\bot, \texttt{null}\} \mapsto N_2 \cup \{\texttt{null}, \bot\}$ est un isomorphisme de GM_1 vers GM_2 si et seulement si les conditions suivantes sont respectées :

- f est une bijection,
- $f(\texttt{null}) = \texttt{null}$ et $f(\bot) = \bot$,
- pour tout $p \in P$, $loc_2(p) = f(loc_1(p))$,
- pour tout $n \in N_1$, $f(succ_1(n)) = succ_2(f(n))$.

Par définition, f étant une bijection, si il existe un isomorphisme de GM_1 vers GM_2 alors il existe un isomorphisme de GM_2 vers GM_1. Nous dirons que deux graphes mémoire sont isomorphes si il existe un isomorphisme de l'un vers l'autre. Finalement, nous dirons que deux graphes mémoire sont égaux si et seulement si ils sont isomorphes.

4.2.2 Syntaxe

De façon à faire le parallèle avec les systèmes à compteurs que nous avons étudiés dans la partie précédente de cette thèse, nous introduisons maintenant ce que nous appelons les systèmes à pointeurs qui correspondent à des automates finis, c'est à dire une structure de contrôle finie, manipulant des pointeurs. Avant de définir formellement le modèle des systèmes à pointeurs, nous présentons la syntaxe des gardes et des actions qui étiquetteront les transitions de ces systèmes. Nous donnons l'interprétation sémantique de ces gardes et de ces actions dans la section suivante.

Étant donné un ensemble de pointeurs P, nous définissons un ensemble $\mathcal{G}_P$ de gardes g sur les pointeurs de P grâce à la grammaire suivante :

$$g ::= true \mid p == \texttt{null} \mid p == p' \mid \neg g \mid g \wedge g$$

où p, p' sont des pointeurs appartenant à P. Intuitivement, ces gardes permettent de tester si deux pointeurs pointent sur le même noeud dans un graphe mémoire, ou si un pointeur pointe sur le noeud `null` ; il est de plus possible de composer les gardes en utilisant les opérateurs booléens classiques.

De la même façon, nous définissons un ensemble $\mathcal{A}_P$ d'actions a sur les pointeurs de P par la grammaire suivante :

$$\begin{aligned} a ::= \quad & p := p' \mid p := \texttt{null} \mid p := p'.\texttt{succ} \mid p.\texttt{succ} := p' \mid \\ & p := \texttt{malloc} \mid \texttt{free}(p) \mid \texttt{skip} \end{aligned}$$

avec $p, p' \in P$. Ces différentes actions servent dans l'ordre où elles apparaissent à faire pointer une variable à l'endroit où pointe une autre variable, à faire pointer une variable vers le noeud `null`, à faire pointer une variable vers le successeur d'un noeud pointé par une autre variable, à changer le successeur d'un noeud pointé par une variable, à allouer une zone mémoire, à libérer une zone mémoire et à ne rien faire. Remarquons que si l'on compare avec les primitives du langage C, la fonction `malloc` que nous utilisons ici, ne prend pas d'argument, alors qu'en C le programmeur doit spécifier la taille de la zone mémoire qu'il alloue. Cette différence est liée au fait que nous considérons ici qu'un seul type de structure de données pour lequel nous réalisons l'allocation dynamique de mémoire, à savoir les listes simplement chaînées et par conséquent la taille de la zone mémoire allouée est implicitement

```
typedef struct node {
  struct node* succ;
  int data;
}* List;

void deleteAll(List p){
  List q;
  while (p!=NULL){
    q = p;
    p = p->succ;
    free(q);
  }
}
```

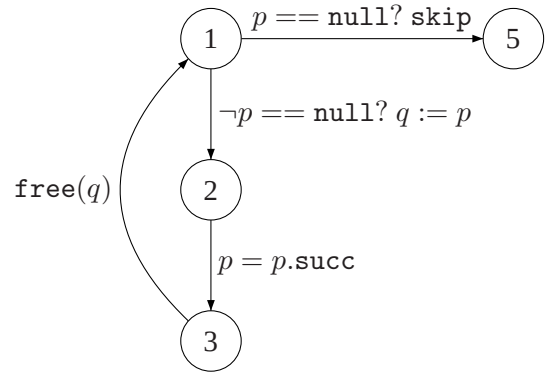


FIGURE 4.3 – Un programme C et le système à pointeurs correspondant

toujours égale à la taille d'une cellule de liste simplement chaînée.

Notation 4.4 *Par la suite, comme pour l'ensemble $\mathcal{GM}$, lorsque l'ensemble des pointeurs P considéré sera explicite, nous ne le noterons pas dans $\mathcal{G}_P$ et $\mathcal{A}_P$ en écrivant seulement $\mathcal{G}$ et $\mathcal{A}$.*

Nous donnons maintenant la définition des systèmes à pointeurs.

Definition 4.5 (Système à pointeurs) *Un système à pointeurs est un triplet $S = \langle Q, P, E \rangle$ tel que :*

- *Q est un ensemble fini d'états de contrôle,*
- *P est un ensemble fini de pointeurs,*
- *$E \subseteq Q \times \mathcal{G} \times \mathcal{A} \times Q$ est un ensemble fini de transitions, chacune étant étiquetée par une garde et une action sur les pointeurs de P.*

Exemple 4.6 *La figure 4.3 représente un programme C manipulant une liste simplement chaînée ainsi que le système à pointeurs correspondant. Ce programme prend en entrée une liste simplement chaînée et libère chacune de ces cellules. Étant donné que ce programme ne fait que modifier la forme de la liste, sans toucher aux données contenues dans les différentes cellules, le programme et le système à pointeurs sont quasiment identiques.*

Comme les systèmes à pointeurs manipulent uniquement des pointeurs et qu'il n'est pas possible d'accèder aux données stockées dans les cellules des listes simplement chaînées, pour obtenir un système à pointeurs à partir d'un programme C, il est souvent nécessaire d'effectuer une abstraction du programme. Dans le travail que nous présentons ici, nous ne nous intéresserons pas à cette phase d'abstraction mais seulement à l'étude des systèmes à pointeurs. Notons juste que développer des techniques pour réaliser une telle abstraction est un projet de recherche à part entière. En effet, l'abstraction doit couvrir tous les comportements du programme, de façon à ne pas laisser passer d'éventuelles erreurs, mais elle ne doit pas être trop approximative de façon à limiter le nombre de fausses alarmes, c'est-à-dire d'erreurs qui ont lieu pour l'abstraction mais qui ne sont pas réalisées par le programme.

4.2.3 Sémantique

Nous allons maintenant donner la sémantique associée à un système à pointeurs. De la même façon que nous avons associé un système de transitions à un système à compteurs, nous allons définir le système de transitions associé à un système à pointeurs. Pour ce faire, nous définissons la sémantique des gardes et des actions introduites précédemment.

Dans cette section, nous considérons un ensemble P de pointeurs. Étant donnés une garde $g \in \mathcal{G}$ et un graphe mémoire $GM \in \mathcal{GM}$ tel que $GM = (N, P, succ, loc)$, nous définissons par induction la relation de satisfiabilité $GM \models g$ de la façon suivante :

- $GM \models true$ est toujours vraie,
- $GM \models p == \texttt{null}$ si et seulement si $loc(p) = \texttt{null}$,
- $GM \models p == p'$ si et seulement si $loc(p) = loc(p')$,
- $GM \models \neg g$ si et seulement si $GM \not\models g$,
- $GM \models g \wedge g'$ si et seulement si $GM \models g$ et $GM \models g'$.

En regardant de plus près la relation de satisfiabilité définie ci-dessus, on peut remarquer que si deux variables p et p' pointent toutes deux vers $\perp$, alors la garde $p == p'$ est satisfaite. Ce choix a été fait de façon à ce que si une action fait pointer la variable p vers le noeud où pointe la variable p', alors on souhaite que la garde soit satisfaite (comme c'est le cas en C). Toutefois cette sémantique peut aussi poser problème car, comme nous le verrons plus loin, si l'on libère le noeud pointé par p en faisant $\texttt{free}(p)$ et si ensuite on libère le noeud pointé par p' avec un $\texttt{free}(p')$ alors nous considérons que les deux variables p et p' pointent toutes deux sur le noeud $\perp$ et par conséquent la garde $p == p'$ est aussi satisfaite, ce qui n'est plus du tout sûr en C. Pour palier ce problème, une solution aurait pu être d'utiliser plusieurs noeuds de type $\perp$ mais afin de ne pas compliquer encore plus la sémantique, nous n'avons pas choisi cette option.

Nous nous attachons maintenant à définir la sémantique des actions manipulant les pointeurs. Nous introduisons trois éléments spéciaux qui sont `Seg`, `Leak` et `LeakSeg` qui n'appartiennent pas à $\mathcal{GM}$. Intuitivement, `Seg` symbolise une erreur de segmentation ("segmentation fault" en anglais), `Leak` une fuite mémoire ("memory leakage" en anglais) et `LeakSeg` symbolise le fait que les deux erreurs ont eu lieu. Nous utilisons ces trois éléments car, comme nous le signalions précédemment, les actions d'un programme manipulant la mémoire sont susceptibles de réaliser ces erreurs. Ainsi étant donnée une action de pointeurs $a \in \mathcal{A}$, nous allons définir une fonction $[\![a]\!]_P : \mathcal{GM} \to \mathcal{GM} \cup \{\texttt{Seg}, \texttt{Leak}, \texttt{LeakSeg}\}$ qui prend comme argument un graphe mémoire et renvoie soit le graphe mémoire résultat de l'action, soit `Seg` si l'action réalise une erreur de segmentation, soit `Leak` si l'action réalise une fuite mémoire, soit `LeakSeg` si l'action réalise une fuite mémoire et une erreur de segmentation.

Notation 4.7 *Nous noterons $\mathcal{GM}_P^{err}$ (ou $\mathcal{GM}^{err}$ lorsqu'il n'y aura pas d'ambiguïté sur l'ensemble de variables de pointeurs considéré) l'ensemble $\mathcal{GM}_P \cup \{\texttt{Leak}, \texttt{Seg}, \texttt{LeakSeg}\}$.*

Les figures 4.5 à 4.10 donnent la sémantique opérationnelle associée à chacune des actions possibles d'un système à pointeurs, exceptée celle de l'action `skip` qui est évidente, $[\![\texttt{skip}]\!]_P$ étant la fonction identité sur $\mathcal{GM}$. Considérons par exemple la sémantique des actions de la forme $p.\texttt{succ} := p'$ décrite à la figure 4.8. Cette action a pour but de changer le successeur du noeud pointé par p. Ainsi si p pointe vers `null` ou $\perp$, cette action réalise une erreur de segmentation, comme cela est indiqué par la troisième ligne du tableau. Il faut de plus faire attention à ce que la modification du graphe ne génère

pas de fuite mémoire. Pour ce faire, il faut s'assurer qu'une fois que l'on aura changer le successeur du noeud pointé par p, l'actuel successeur de ce noeud sera encore accessible par une variable (sauf bien entendu si il s'agit de `null` ou de $\perp$), et c'est à cela que servent les autres hypothèses exprimées dans la première ligne du tableau. Notons par ailleurs que pour définir cette sémantique, nous avons fait certains choix qui peuvent être discutés, en particulier en ce qui concerne la sémantique des actions de la forme $\texttt{free}(p)$. En effet, nous supposons que si la variable p pointe vers `null` ou $\perp$, l'action $\texttt{free}(p)$ réalise une erreur de segmentation, comme cela est indiqué par la deuxième ligne du tableau de la figure 4.10. Il s'avère que pour certains systèmes, libérer un pointeur vers `null` est une opération qui ne réalise pas d'erreur de segmentation, cependant nous considérons que si cela arrive, il s'agit quand même d'un dysfonctionnement du programme et c'est pour cette raison que nous considérons tout de même qu'une erreur de segmentation survient. Notons que dans les figures 4.5 à 4.10, la notation $loc[p \mapsto n]$ est utilisée pour définir la fonction loc' telle que pour tout $p' \neq p$, $loc'(p') = loc(p')$ et $loc'(p) = n$.

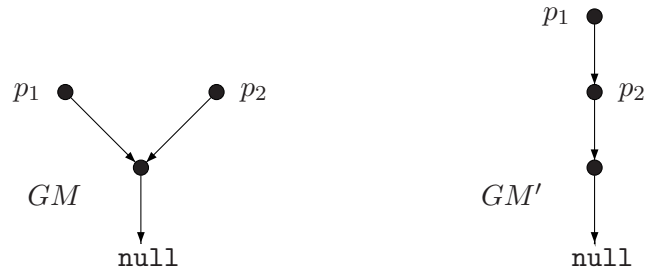


FIGURE 4.4 – Effet de l'action $p_1.\texttt{succ} := p_2$ sur GM

Exemple 4.8 *La figure 4.4 montre l'effet de l'action $p_1.\texttt{succ} := p_2$ sur le graphe GM. Comme le pointeur p_1 ne pointe pas sur `null` ou sur $\perp$ et comme le successeur de $loc(p_1)$ est accessible à partir du pointeur p_2, on en déduit que cette action ne réalise ni une erreur de segmentation ni une fuite mémoire. En revanche, l'action $p_1 := p_2$ réalise une fuite mémoire sur le graphe mémoire GM car si l'on fait pointer la variable p_1 sur le noeud $loc(p_2)$, le noeud pointé par p_1 n'est plus accessible à partir d'aucune des variables du graphe.*

Nous pouvons maintenant donner la définition du système de transitions associé à un système à pointeurs. Une configuration d'un système à pointeurs est soit :

1. une paire (q, GM) où $q \in Q$ est un état de contrôle et GM est un graphe mémoire dans $\mathcal{GM}_P$,
2. une paire $(q, \texttt{Leak})$ avec $q \in Q$; une telle configuration caractérise une fuite mémoire,
3. une paire $(q, \texttt{Seg})$ avec $q \in Q$; une telle configuration caractérise une erreur de segmentation,
4. une paire $(q, \texttt{LeakSeg})$ avec $q \in Q$; une telle configuration caractérise une fuite mémoire et une erreur de segmentation.

La sémantique opérationnelle d'un système à pointeurs $S = \langle Q, P, E\rangle$ est donnée par un système de transitions étiqueté $TS(S) = \langle Q \times \mathcal{GM}_P^{err}, E, \rightarrow\rangle$ de telle sorte que $\rightarrow \subseteq (Q \times \mathcal{GM}_P) \times E \times (Q \times$

$\mathcal{GM}_P^{err}$) est la relation définie de la façon suivante, pour tout (q, GM) dans $Q \times \mathcal{GM}_P$, pour tout $(q', GM') \in Q \times \mathcal{GM}_P^{err}$, pour tout $e \in E$:

$(q, GM) \xrightarrow{e} (q', GM')$ si et seulement si $e = (q, (g, a), q')$ et $GM \models g$ et $GM' = [\![a]\!]_P(GM)$

Hypothèses	$[\![a]\!]_P(GM)$
$GM = (N, P, succ, loc)$ et $p \neq p'$ et soit il existe $p'' \in P \setminus \{p\}$ tel que $loc(p) \in$ **List**(GM, p''), soit $loc(p) \in \{\texttt{null}, \bot\}$	$(N, P, succ, loc[p \mapsto loc(p')])$
$p = p'$	GM
$GM = (N, P, succ, loc)$ et $p \neq p'$ et il n'existe pas $p'' \in P \setminus \{p\}$ tel que $loc(p) \in$ **List**(GM, p'') et $loc(p) \notin \{\texttt{null}, \bot\}$	`Leak`

FIGURE 4.5 – Sémantique de l'action $a = (p := p')$

Hypothèses	$[\![a]\!]_P(GM)$
$GM = (N, P, succ, loc)$ et soit il existe $p' \in P \setminus \{p\}$ tel que $loc(p) \in$ **List**(GM, p'), soit $loc(p) \in \{\texttt{null}, \bot\}$	$(N, P, succ, loc[p \mapsto \texttt{null}])$
$GM = (N, P, succ, loc)$ et il n'existe pas $p' \in P \setminus \{p\}$ tel que $loc(p) \in$ **List**(GM, p') et $loc(p) \notin \{\texttt{null}, \bot\}$	`Leak`

FIGURE 4.6 – Sémantique de l'action $a = (p := \texttt{null})$

Hypothèses	$[\![a]\!]_P(GM)$
$GM = (N, P, succ, loc)$ et $loc(p') \notin \{\texttt{null}, \bot\}$ et $succ(loc(p')) \neq loc(p)$ et soit il existe $p'' \in P \setminus \{p\}$ tel que $loc(p) \in$ **List**(GM, p''), soit $loc(p) \in \{\texttt{null}, \bot\}$ soit $p = p'$ et $loc(p) \in$ **List**$(GM, succ(loc(p)))$	$(N, P, succ, loc[p \mapsto succ(loc(p'))])$
$GM = (N, P, succ, loc)$ et $loc(p') \notin \{\texttt{null}, \bot\}$ et $succ(loc(p')) = loc(p)$	GM
$GM = (N, P, succ, loc)$ et $loc(p') \in \{\texttt{null}, \bot\}$ et soit il existe $p'' \in P \setminus \{p\}$ tel que $loc(p) \in$ **List**(GM, p''), soit $loc(p) \in \{\texttt{null}, \bot\}$ soit $p = p'$ et $loc(p) \in$ **List**$(GM, succ(loc(p)))$	`Seg`
$GM = (N, P, succ, loc)$ et $loc(p') \notin \{\texttt{null}, \bot\}$ et $succ(loc(p')) \neq loc(p)$ et il n'existe pas $p'' \in P \setminus \{p\}$ tel que $loc(p) \in$ **List**(GM, p''), et $loc(p) \notin \{\texttt{null}, \bot\}$ et ($p \neq p'$ ou $loc(p) \notin$ **List**$(GM, succ(loc(p)))$)	`Leak`
$GM = (N, P, succ, loc)$ et $loc(p') \in \{\texttt{null}, \bot\}$ et il n'existe pas $p'' \in P \setminus \{p\}$ tel que $loc(p) \in$ **List**(GM, p''), et $loc(p) \notin \{\texttt{null}, \bot\}$ et ($p \neq p'$ ou $loc(p) \notin$ **List**$(GM, succ(loc(p)))$)	`LeakSeg`

FIGURE 4.7 – Sémantique de l'action $a = (p := p'.\texttt{succ})$

Hypothèses	$[\![a]\!]_P(GM)$
$GM = (N, P, succ, loc)$ et $loc(p) \notin \{\texttt{null}, \bot\}$ et soit il existe $p'' \in P \setminus \{p\}$ tel que $loc(p) \notin$ **List**(GM, p'') ou $succ(loc(p)) = loc(p'')$ et tel que $succ(loc(p)) \in$ **List**(GM, p''), soit $succ(loc(p)) \in \{\texttt{null}, \bot\}$	$(N, P, succ[loc(p) \mapsto loc(p')], loc)$
$GM = (N, P, succ, loc)$ et $loc(p) \in \{\texttt{null}, \bot\}$	`Seg`
$GM = (N, P, succ, loc)$ et $loc(p) \notin \{\texttt{null}, \bot\}$ et il n'existe pas $p'' \in P \setminus \{p\}$ tel que $loc(p) \notin$ **List**(GM, p'') ou $succ(loc(p)) = loc(p'')$ et tel que $succ(loc(p)) \in$ **List**(GM, p''), et $succ(loc(p)) \notin \{\texttt{null}, \bot\}$	`Leak`

FIGURE 4.8 – Sémantique de l'action $a = (p.\texttt{succ} := p')$

Hypothèses	$[\![a]\!]_P(GM)$
$GM = (N, P, succ, loc)$ et soit il existe $p'' \in P \setminus \{p\}$ tel que $loc(p) \in$ **List**(GM, p'), soit $loc(p) \in \{\texttt{null}, \bot\}$	$(N \cup \{n\}, P, succ[n \mapsto \bot], loc[p \mapsto n])$ avec $n \notin N$
$GM = (N, P, succ, loc)$ et il n'existe pas $p' \in P \setminus \{p\}$ tel que $loc(p) \in$ **List**(GM, p') et $loc(p) \notin \{\texttt{null}, \bot\}$	`Leak`

FIGURE 4.9 – Sémantique de l'action $a = (p := \texttt{malloc})$

Hypothèses	$[\![a]\!]_P(GM)$
$GM = (N, P, succ, loc)$ et $loc(p) \notin \{\texttt{null}, \bot\}$ et il existe $p' \in P \setminus \{p\}$ tel que $loc(p) \notin$ **List**(GM, p') ou $succ(loc(p)) = loc(p')$ et tel que $succ(loc(p)) \in$ **List**(GM, p')	$(N \setminus \{loc(p)\}, P, succ', loc')$ et pour tout $p' \in P$ si $loc(p') = loc(p)$ alors $loc'(p') = \bot$ si $loc(p') \neq loc(p)$ alors $loc'(p') = loc(p')$ et pout tout $n \in N \setminus \{loc(p)\}$, si $succ(n) = loc(p)$ alors $succ'(n) = \bot$ et si $succ(n) \neq loc(p)$ alors $succ'(n) = succ(n)$
$GM = (N, P, succ, loc)$ et $loc(p) \in \{\texttt{null}, \bot\}$	`Seg`
$GM = (N, P, succ, loc)$ et $loc(p) \notin \{\texttt{null}, \bot\}$ et il n'existe pas $p' \in P \setminus \{p\}$ tel que $loc(p) \notin$ **List**(GM, p') ou $succ(loc(p)) = loc(p')$ et tel que $succ(loc(p)) \in$ **List**(GM, p')	`Leak`

FIGURE 4.10 – Sémantique de l'action $a = (\texttt{free}(p))$

4.2.4 Problèmes d'accessibilité

Les systèmes à pointeurs sont un modèle pour la vérification de programmes manipulant des listes simplement chaînées. Nous présentons ici les différents problèmes auxquels nous allons nous intéresser. Ces problèmes correspondent à la traduction dans notre formalisme de la vérification des propriétés telles que l'absence d'erreur de segmentation ou l'absence de fuite mémoire. Comme nous l'avons vu, il est impératif que ces propriétés soient vérifiées pour garantir le bon fonctionnement d'un programme.

Soient $S = \langle Q, P, E\rangle$ un système à pointeurs, $TS(S) = \langle Q \times \mathcal{GM}^{err}, E, \rightarrow\rangle$ le système de transitions qui lui est associé et $c_0 \in Q \times \mathcal{GM}^{err}$ une configuration initiale. L'ensemble d'accessibilité du système à pointeurs initialisé (S, c_0) est $\mathbf{Reach}(S, c_0) = \{c \in Q \times \mathcal{GM}^{err} \mid c_0 \rightarrow^* c\}$.

Nous définissons différents problèmes pour les systèmes à pointeurs munis d'une configuration initiale. Nous donnerons par la suite, dans la section 4.3, une version symbolique de ces problèmes comme cela a été fait dans le cadre des systèmes à compteurs. Nous définissons le *problème d'accessibilité d'une erreur de segmentation* de la façon suivante :

Entrées : Un système à pointeurs $S = \langle Q, P, E\rangle$ et une configuration initiale $c_0 \in Q \times \mathcal{GM}_P^{err}$,
Question : Existe-t-il $q \in Q$ tel que $(q, \texttt{Seg}) \in \mathbf{Reach}(S, c_0)$ ou $(q, \texttt{LeakSeg}) \in \mathbf{Reach}(S, c_0)$?

Si la réponse est oui, nous dirons que (S, c_0) provoque une erreur de segmentation. De la même façon, nous définissons le *problème d'accessibilité d'une fuite mémoire* :

Entrées : Un système à pointeurs $S = \langle Q, P, E\rangle$ et une configuration initiale $c_0 \in Q \times \mathcal{GM}_P^{err}$,
Question : Existe-t-il $q \in Q$ tel que $(q, \texttt{Leak}) \in \mathbf{Reach}(S, c_0)$ ou $(q, \texttt{LeakSeg}) \in \mathbf{Reach}(S, c_0)$?

Si la réponse est oui, nous dirons que (S, c_0) provoque une fuite mémoire. Si le système à pointeurs ne provoque ni une erreur de segmentation, ni une fuite mémoire, nous considérons qu'il a un comportement que nous qualifions de sans erreur. Nous définissons de plus les problèmes suivants qui sont similaires à ceux introduits pour les systèmes à compteurs. Ainsi nous considérerons le *problème d'accessibilité d'un état de contrôle* :

Entrées : Un système à pointeurs $S = \langle Q, P, E\rangle$, une configuration initiale $c_0 \in Q \times \mathcal{GM}_P^{err}$ et un état de contrôle $q \in Q$,
Question : Existe-t-il $GM \in \mathcal{GM}^{err}$ tel que $(q, GM) \in \mathbf{Reach}(S, c_0)$?

Si la réponse est oui, nous dirons que q est accessible dans (S, c_0). Remarquons que l'élément GM tel que $(q, GM) \in \mathbf{Reach}(s, c_0)$ peut appartenir à $\mathcal{GM}^{err}$, c'est à dire qu'il peut s'agir d'un graphe mémoire mais aussi d'un élément de $\{\texttt{Leak}, \texttt{Seg}, \texttt{LeakSeg}\}$. Néanmoins une approche classique consistera à considérer ce problème uniquement pour les systèmes à pointeurs ayant un comportement sans erreur. Nous définissons finalement le *problème d'accessibilité d'une configuration* :

Entrées : Un système à pointeurs $S = \langle Q, P, E\rangle$ et deux configurations $c_0, c \in Q \times \mathcal{GM}_P^{err}$,
Question : Est-ce-que $c \in \mathbf{Reach}(S, c_0)$?

Si la réponse est oui, nous dirons que la configuration c est accessible dans (S, c_0).

4.2.5 Indécidabilité

Dans [BFN04], les auteurs prouvent que les différents problèmes que nous avons introduits sur les systèmes à pointeurs sont indécidables.

Théorème 4.9 *[BFN04, Bar05] Les problèmes d'accessibilité d'une erreur de segmentation et d'une fuite mémoire, tout comme les problèmes d'accessibilité d'un état de contrôle et d'une configuration sont indécidables pour les systèmes à pointeurs manipulant au moins trois variables de pointeurs.*

Nous rappelons l'idée de la preuve qui établit un lien entre les systèmes à compteurs et les systèmes à pointeurs.

Idée de la preuve : Nous réduisons le problème de l'accessibilité d'un état de contrôle pour les machines de Minsky déterministes à 2 compteurs qui est un problème indécidable (cf. théorème 1.37) à ce problème. Soient $S_C = \langle Q_C, \{x_1, x_2\}, E_C \rangle$ une machine de Minsky déterministe à deux compteurs munie d'une configuration initiale $c_0 = (q_0, (0, 0))$ (sans perte de généralités, nous pouvons supposer que les valeurs des compteurs valent initialement 0) et q_F un état de contrôle dans Q. Nous construisons un système à pointeurs $S = \langle Q, P, E \rangle$ avec :

- $Q = Q_C \cup Q_{aux}$ (nous ne détaillons pas l'ensemble Q_{aux} qui correspond aux états non étiquetés des figures 4.11, 4.12 et 4.13),
- $P = \{p_1, p_2, p_3\}$,
- $E \subseteq Q \times \mathcal{G} \times \mathcal{A} \times Q$ est la plus petite relation telle que, pour tout $i \in \{1, 2\}$:
  - pour chaque $(q, \textbf{inc}(x_i), q') \in E_C$, les transitions décrites par la figure 4.11 appartiennent à E,
  - pour chaque $(q, \textbf{dec}(x_i), q') \in E_C$, les transitions décrites par la figure 4.12 appartiennent à E,
  - pour chaque $(q, \textbf{ifzero}(x_i), q') \in E_C$, $(q, p_i == \texttt{null}?\texttt{skip}, q') \in E$,
  - les transitions de la figure 4.13 appartiennent aussi à E.

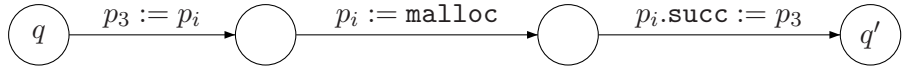


FIGURE 4.11 – Encodage des transitions de la forme $e = (q, \textbf{inc}(x_i), q')$

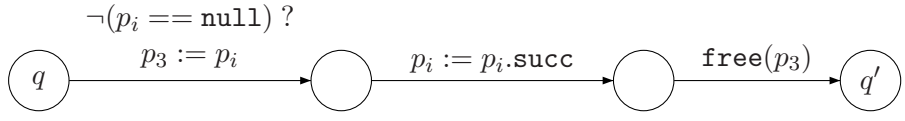


FIGURE 4.12 – Encodage des transitions de la forme $e = (q, \textbf{dec}(x_i), q')$

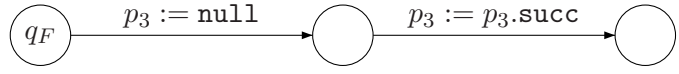


FIGURE 4.13 – Transitions ajoutées à l'état q_F

Nous construisons ensuite le graphe mémoire initiale GM_0 dans lequel les trois pointeurs p_1, p_2 et p_3 pointent tous les trois vers `null`. Intuitivement, la valeur de chaque compteur x_i au cours de l'exécution correspond à la longueur de la liste commençant par le noeud pointée par p_i. Ainsi pour incrémenter le compteur, on ajoute un élément dans la liste et pour le décrémenter, on retire un élément de la liste (sauf si la liste est vide). Pour réaliser le test à zéro, il suffit de tester si le pointeur p_i pointe vers `null`. Chacune de ces opérations ne réalise aucune erreur, et lorsque l'on atteint l'état q_F, on met la variable p_3 sur `null` et on effectue l'action $p_3 := p_3.\texttt{succ}$ qui réalise une erreur de segmentation. Ainsi le système à pointeurs S muni de la configuration initiale (q_0, GM_0) provoque une erreur de segmentation si et seulement si l'état de contrôle q_F est accessible dans (S_C, c_0). L'indécidabilité pour les autres problèmes d'accessibilité se prouvent exactement de la même façon. □

Malgré ce dernier résultat, nous verrons qu'il est possible de développer des méthodes permettant de résoudre dans certains cas ces différents problèmes, tout comme nous avons présenté des techniques permettant d'analyser des systèmes à compteurs, pour lesquels de nombreux problèmes sont aussi indécidables.

4.3 Représentation symbolique de graphes mémoire

4.3.1 États mémoire symboliques

Nous avons vu, en étudiant les systèmes à compteurs, que les formules de l'arithmétique de Presburger permettaient de représenter un ensemble infini de configurations. Nous souhaitons maintenant proposer un cadre analogue pour représenter des ensembles, possiblement infinis, de graphes mémoire. Idéalement nous souhaitons construire une représentation symbolique décidable et qui soit close par union, intersection et complément de façon à ce qu'elle puisse être facilement manipulée. Dans [BFN04] et [Bar05], les auteurs introduisirent une telle représentation symbolique pour les graphes mémoire, que nous présentons maintenant.

4.3.1.1 Formes mémoire

Dans un premier temps, nous définissons les formes mémoire.

Definition 4.10 (Forme mémoire) *Une forme mémoire est un sextuplet $FM = (N, P, X, succ, loc, c)$ tel que :*

- *$(N, P, succ, loc)$ est un graphe mémoire vérifiant :*
  - *pour tout noeud $n \in N$, $loc^{-1}(\{n\}) \neq \emptyset$ ou $|succ^{-1}(\{n\})| \geq 2$,*
- *X est un ensemble de compteurs,*
- *$c : N \to X$ est une fonction injective associant à chaque noeud un compteur différent.*

Autrement dit, une forme mémoire est un graphe mémoire dans lequel on associe à chaque noeud un compteur différent et dans lequel chaque noeud est pointé par un pointeur ou a au moins deux prédécesseurs. Étant donnés un ensemble P de pointeurs et un ensemble X de compteurs, nous notons $\mathcal{FM}_{P,X}$ l'ensemble des formes mémoire utilisant P et X comme ensembles de pointeurs et de compteurs. Si $FM = (N, P, X, loc, succ, c)$ est une forme mémoire, nous notons X_{FM} le sous-ensemble de X tel que $c(N) = X_{FM}$.

Étant données deux formes mémoire $FM_1 = (N_1, P, X, succ_1, loc_1, c_1)$ et $FM_2 = (N_2, P, X, succ_2, loc_2, c_2)$ dans $\mathcal{FM}_{P,X}$, une fonction $f : N_1 \to N_2$ est un isomorphisme de FM_1 vers FM_2 si et seulement si f est un isomorphisme du graphe mémoire $(N_1, P, succ_1, loc_1)$ vers le graphe mémoire $(N_2, P, succ_2, loc_2)$. Nous dirons que deux formes mémoire FM_1 et FM_2 sont isomorphes, noté $FM_1 \approx FM_2$, si et seulement si il existe un isomorphisme de FM_1 vers FM_2. De plus nous dirons que $FM_1 = (N_1, P, X, succ_1, loc_1, c_1)$ et $FM_2 = (N_2, P, X, succ_2, loc_2, c_2)$ sont égales, noté $FM_1 = FM_2$, si et seulement si il existe un isomorphisme $f : N_1 \to N_2$ de FM_1 vers FM_2 tel que pour tout $n \in N_1$, $c_1(n) = c_2(f(n))$.

Si $FM = (N, P, X, succ, loc, c)$ est une forme mémoire symbolique, nous notons $|FM|$ sa taille égale à $|N|$ son nombre de noeuds. Nous avons alors la propriété suivante.

Proposition 4.11 *[Bar05] Pour tout ensemble fini de pointeurs P et de compteurs X, si FM est une forme mémoire dans $\mathcal{FM}_{P,X}$ alors $|FM| \leq 2|P|$.*

Ainsi dans une forme mémoire symbolique, il y a au plus $2|P|$ noeuds, ceci est dû au fait que chaque noeud d'une forme mémoire symbolique est pointé par une variable de P ou a au moins deux prédécesseurs et est accessible par une variable de P. Comme de plus dans un graphe mémoire $FM = (N, P, X, succ, loc, c)$, la fonction $c : N \to X$ est totale et injective, nous avons nécessairement $|N| \leq |X|$. Ainsi de façon, à pouvoir décrire toutes les formes mémoire possibles sur un ensemble de pointeurs P sans être bloqué par le nombre de compteurs, nous supposerons dans la suite que nous aurons toujours $2|P| \leq |X|$. Une conséquence de la proposition suivante peut alors être exprimée ainsi :

Proposition 4.12 *[Bar05] Pour tout ensemble fini de pointeurs P et de compteurs X, nous avons :*

$$|\mathcal{FM}_{X,P}| \leq (2|P|)^{2|P|}|X|^{2|P|}$$

Nous utiliserons pleinement par la suite le fait que le nombre de formes mémoire sur un ensemble de pointeurs P et un ensemble de compteurs X est borné. Finalement, nous avons également cette dernière propriété concernant l'isomorphisme de formes mémoire :

Proposition 4.13 *[Bar05] Vérifier si deux formes mémoire de $\mathcal{FM}_{P,X}$ sont isomorphes est décidable en temps linéaire en $|P|$.*

4.3.1.2 Forme mémoire valuée

Nous montrons maintenant comment nous nous servons des compteurs présents dans une forme mémoire.

Definition 4.14 (Forme mémoire valuée) *Une forme mémoire valuée est une paire $(FM, \mathbf{v})$ telle que :*
- *FM est une forme mémoire,*
- *$\mathbf{v} : X_{FM} \to \mathbb{N}^*$ est une fonction qui associe à chaque compteur de FM une valeur entière strictement positive.*

En associant à chaque compteur d'une forme mémoire une valeur strictement positive, nous définissons en réalité un graphe mémoire. En effet, si $(FM, \mathbf{v})$ est une forme mémoire valuée avec $FM = (N, P, X, succ, loc, c)$, nous lui associons le graphe mémoire obtenu à partir de FM en insérant pour chaque noeud $n \in N$, $\mathbf{v}(c(n)) - 1$ noeuds intermédiaires entre n et $succ(n)$. Nous notons $FM(\mathbf{v})$ le graphe mémoire ainsi obtenu. Nous avons alors la propriété suivante :

Proposition 4.15 *Pour tout graphe mémoire $GM \in \mathcal{GM}_P$, pour tout ensemble de compteurs X avec $2|P| \leq |X|$, il existe une forme mémoire valuée $(FM, \mathbf{v})$ avec $FM \in \mathcal{FM}_{P,X}$ telle que $GM = FM(\mathbf{v})$.*

Idée de la preuve : Il suffit de retirer dans GM les noeuds qui ne sont pas pointés par une variable et ayant un unique prédécesseur. On obtient alors en ajoutant des compteurs la forme mémoire FM, et ensuite on construit la valuation **v** de façon à rajouter dans $FM(\mathbf{v})$ exactement le nombre de noeuds enlevés en construisant FM. □

Exemple 4.16 *La figure 4.14 nous montre comment à partir d'une forme mémoire valuée, on construit le graphe mémoire correspondant.*

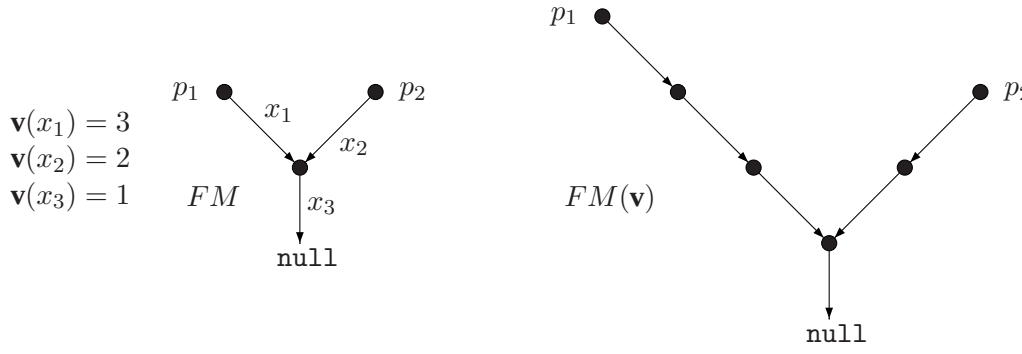


FIGURE 4.14 – Une forme mémoire valuée $(FM, \mathbf{v})$ et le graphe mémoire correspondant $FM(\mathbf{v})$

Il s'avère que si deux formes mémoire valuées correspondent à un même graphe mémoire alors leurs formes mémoire sous-jacentes sont nécessairement isomorphes, en effet :

Proposition 4.17 *[Bar05] Soit $GM \in \mathcal{GM}$ un graphe mémoire. Si $(FM_1, \mathbf{v}_1)$ et $(FM_2, \mathbf{v}_2)$ sont deux formes mémoire valuées telles que $GM = FM_1(\mathbf{v}_1) = FM_2(\mathbf{v}_2)$ alors nous avons $FM_1 \approx FM_2$.*

Autrement dit, étant donné un graphe mémoire, il existe une unique forme mémoire valuée (à isomorphisme près) correspondant à ce graphe mémoire.

4.3.1.3 Formes mémoire symboliques et états mémoire symboliques

Nous donnons maintenant une représentation symbolique pour les graphes mémoire utilisant les formes mémoire.

Definition 4.18 (Forme mémoire symbolique) *Une forme mémoire symbolique est une paire $FMS = (FM, \phi)$ telle que :*

- *FM est une forme mémoire,*
- *$\phi \in$ **Presb**(X_{FM}) est une formule de Presburger telle que la formule $\phi \Rightarrow \bigwedge_{x \in X_{FM}} x > 0$ est valide.*

Dans une forme mémoire symbolique, la formule de Presburger nous permet d'encoder un nombre possiblement infini de valuations pour les compteurs de FM et nous pouvons ainsi encoder un nombre possiblement infini de graphes mémoire. Étant donnée une forme mémoire symbolique (FM, ϕ) avec FM dans $\mathcal{FM}_{P,X}$, nous lui associons un ensemble de graphes mémoire $[\![(FM, \phi)]\!] = \{FM(\mathbf{v}) \in \mathcal{GM}_P \mid \mathbf{v} \in [\![\phi]\!]_{X_{FM}}\}$. Cependant, comme nous l'avons signalé, nous souhaitons avoir une représentation symbolique qui soit close par union, intersection et complément, ce qui n'est pas le cas des formes mémoire symboliques. C'est pourquoi nous définissons les états mémoire symboliques.

Definition 4.19 (État mémoire symbolique) *Soient P un ensemble fini de pointeurs et X un ensemble fini de compteurs. Un état mémoire symbolique EMS sur P et X est un ensemble fini de formes mémoire symboliques tel que pour tout $(FM, \phi) \in EMS$, $FM \in \mathcal{FM}_{P,X}$ et pour tout $(FM, \phi), (FM', \phi') \in EMS$, si $(FM, \phi) \neq (FM', \phi')$ alors $FM \not\approx FM'$.*

Un état mémoire symbolique correspond donc à un ensemble fini de formes mémoire symboliques dans lequel on ne trouve pas deux formes mémoire isomorphes. Nous notons $\mathcal{EMS}_{P,X}$ l'ensemble des états mémoire symboliques sur P et X. De la même façon que nous avons introduit une fonction de concrétisation pour les formes mémoire symboliques, nous étendons cette fonction aux états mémoire symboliques. Si $EMS = \{FMS_1, FMS_2, \ldots, FMS_m\}$ est un état mémoire symbolique, nous notons $[\![EMS]\!]$ sa concrétisation définie par $[\![EMS]\!] = [\![FMS_1]\!] \cup [\![FMS_2]\!] \cup \ldots \cup [\![FMS_m]\!]$.

4.3.2 Opérations ensemblistes sur les états mémoire symboliques

Nous définissons dans cette section différentes opérations symboliques sur les états mémoire symboliques. Soient P un ensemble fini de pointeurs et X un ensemble fini de compteurs tel que $2|P| \leq |X|$. Dans les définitions qui suivent, si il existe un isomorphisme f d'une forme mémoire symbolique (FM_1, ϕ_1) vers (FM_2, ϕ_2) et si $FM_1 = (N_1, P, X, succ_1, loc_1, c_1)$ et $FM_2 = (N_2, P, X, succ_2, loc_2, c_2)$, nous notons ϕ_1^f la formule obtenue en remplaçant dans ϕ_1 chaque instance de compteur x de X_{FM_1} avec $x = c_1(n)$ par le compteur de X_{FM_2} égale à $c_2(f(n))$. De plus, de façon à simplifier les notations, si FM_1 et FM_2 sont deux formes mémoire, nous noterons $FM_1 \approx_f FM_2$ le fait que FM_1 et FM_2 sont isomorphes et que f est un isomorphisme de FM_1 vers FM_2.

Nous considérons maintenant EMS_1 , EMS_2 et EMS trois états mémoire symboliques dans $\mathcal{EMS}_{P,X}$.

L'union de deux états mémoire symboliques est notée $\sqcup$. Nous avons $EMS = EMS_1 \sqcup EMS_2$ si et seulement si les conditions suivantes sont vérifiées :

- pour toute forme mémoire symbolique (FM_1, ϕ_1) dans EMS_1, il existe une forme mémoire symbolique (FM, ϕ) dans EMS telle que $FM \approx FM_1$,
- pour toute forme mémoire symbolique (FM_2, ϕ_2) dans EMS_2, il existe une forme mémoire symbolique (FM, ϕ) dans EMS telle que $FM \approx FM_2$,
- et pour toute forme mémoire symbolique (FM, ϕ) de EMS :
  - soit il existe (FM_1, ϕ_1) dans EMS_1 telle que $FM \approx FM_1$, soit il existe (FM_2, ϕ_2) dans EMS_2 telle que $FM \approx FM_2$,
  - si il existe (FM_1, ϕ_1) dans EMS_1 telle que $FM_1 \approx_f FM$ et si il existe (FM_2, ϕ_2) dans EMS_2 telle que $FM_2 \approx_g FM_1$, alors $\phi = \phi_1^f \vee \phi_2^g$,
  - si il existe (FM_1, ϕ_1) dans EMS_1 telle que $FM_1 \approx_f FM$ et si il n'existe pas (FM_2, ϕ_2) dans EMS_2 telle que $FM \approx FM_2$ alors $\phi = \phi_1^f$,

– si il n'existe pas (FM_1, ϕ_1) dans EMS_1 telle que $FM \approx FM_1$ et si il existe (FM_2, ϕ_2) dans EMS_2 telle que $FM_2 \approx_g FM$, alors $\phi = \phi_2^g$.

L'intersection de deux états mémoire symboliques est notée $\sqcap$. Nous avons $EMS = EMS_1 \sqcap EMS_2$ si et seulement si les conditions suivantes sont vérifiées pour toute forme mémoire symbolique (FM, ϕ) de EMS :

– si il existe (FM_1, ϕ_1) dans EMS_1 et (FM_2, ϕ_2) dans EMS_2 telle que $FM_1 \approx FM_2$, alors il existe (FM, ϕ) dans EMS tel que $FM \approx FM_1$,
– pour toute forme mémoire symbolique (FM, ϕ) de EMS :
  – il existe (FM_1, ϕ_1) dans EMS_1 telle que $FM \approx FM_1$ et (FM_2, ϕ_2) dans EMS_2 telle que $FM \approx FM_2$,
  – si (FM_1, ϕ_1) est la forme mémoire symbolique de EMS_1 telle que $FM_1 \approx_f FM$ et si (FM_2, ϕ_2) est la forme mémoire symbolique de EMS_2 telle que $FM_2 \approx_g FM$, alors $\phi = \phi_1^f \wedge \phi_2^g$.

Le complément d'un état mémoire symbolique EMS_1 est notée $\neg EMS_1$. Nous avons $EMS = \neg EMS_1$ si et seulement si les conditions suivantes sont vérifiées, pour toute forme mémoire $FM \in \mathcal{FM}_{P,X}$:

– si il n'existe pas de forme mémoire symbolique (FM_1, ϕ_1) dans EMS_1 telle que $FM \approx FM_1$ alors il existe une forme mémoire symbolique $(FM', \bigvee_{x \in X_{FM'}} x > 0)$ dans EMS telle que $FM \approx FM'$,
– si il existe une forme mémoire symbolique (FM_1, ϕ_1) dans EMS_1 telle $FM_1 \approx_f FM$, alors il existe une forme mémoire symbolique (FM', ϕ') dans EMS telle $FM \approx_g FM'$ et de plus $\phi' = \neg(\phi_1^f)^g$.

Notons que comme le nombre de formes mémoire dans $\mathcal{FM}_{P,X}$ est fini (cf. proposition 4.12), comme l'isomorphisme de formes mémoire est décidable (cf. proposition 4.13) et comme l'arithmétique de Presburger est close par disjonction, conjonction et négation nous sommes effectivement capable de construire l'union, l'intersection et le complément d'états mémoire symboliques. Nous avons alors la proposition suivante [Bar05], qui peut facilement être démontrée en utilisant les définitions de l'isomorphisme de formes mémoire et la définition de la concrétisation d'états mémoire symboliques ainsi que la proposition 4.17.

Proposition 4.20 *[Bar05] Soient EMS_1 et EMS_2 deux états mémoire symboliques dans $\mathcal{EMS}_{P,X}$ alors :*

1. $[\![EMS_1 \sqcup EMS_2]\!] = [\![EMS_1]\!] \cup [\![EMS_2]\!]$,
2. $[\![EMS_1 \sqcap EMS_2]\!] = [\![EMS_1]\!] \cap [\![EMS_2]\!]$,
3. $[\![\neg EMS_1]\!] = \mathcal{GM}_P \setminus [\![EMS_1]\!]$.

Finalement, l'inclusion de deux états mémoire symboliques est notée $\sqsubseteq$. Nous avons $EMS_1 \sqsubseteq EMS_2$ si et seulement si :

– pour tout forme mémoire symbolique (FM_1, ϕ_1) de EMS_1 telle que $\phi_1 \not\equiv false$, il existe une forme mémoire symbolique (FM_2, ϕ_2) de EMS_2 telle que $FM_1 \approx_f FM_2$ et telle que la formule $\phi_1^f \Rightarrow \phi_2$ soit valide.

Comme l'isomorphisme de formes mémoire est décidable (cf. proposition 4.13) et comme le problème de satisfiabilité de formules de l'arithmétique de Presburger est décidable, nous avons également le résultat suivant concernant l'inclusion d'états mémoire symboliques :

Proposition 4.21 *[Bar05]*

1. *Savoir si un état mémoire symbolique est inclus dans un autre est un problème décidable.*
2. *Soient EMS_1 et EMS_2 deux états mémoire symboliques dans $EMS_{P,X}$. Alors $EMS_1 \sqsubseteq EMS_2$ si et seulement si $[\![EMS_1]\!] \subseteq [\![EMS_2]\!]$.*

Comme de plus, l'état mémoire symbolique ne comportant aucune forme mémoire symbolique a une concrétisation égale à l'ensemble vide, nous déduisons le corollaire suivant :

Corollaire 4.22 *Le problème du vide est décidable pour les états mémoire symboliques.*

Autrement dit, il existe un algorithme qui, étant donné un état mémoire symbolique EMS dans $\mathcal{EMS}_{P,X}$, renvoie *vrai* si $[\![EMS]\!] = \emptyset$ et *faux* dans le cas contraire.

Ainsi les propositions 4.20 et 4.21 nous permettent de dire que les états mémoire symboliques ont toutes les qualités qu'une représentation symbolique doit avoir pour être facilement manipulée et utilisée. Notons de plus que les états mémoire symboliques permettent d'exprimer des propriétés à la fois qualititatives (c'est-à-dire sur la forme des graphes mémoire) et quantitatives (c'est-à-dire sur le nombre de cellules présentes).

4.3.3 Problèmes d'accessibilité symbolique

Tout comme nous l'avons fait pour les systèmes à compteurs, nous définissons maintenant des problèmes d'accessibilité symbolique en prenant comme représentation symbolique les états mémoire symboliques présentés auparavant.

Soient $S = \langle Q, P, E\rangle$ un système à pointeurs et $TS(S) = \langle Q \times \mathcal{GM}_P^{err}, E, \rightarrow\rangle$ le système de transitions qui lui est associé. Nous considérons de plus un ensemble X de compteurs, tel que $2|P| \leq |X|$, dont nous nous servons pour définir les états mémoire symboliques. Nous appellerons une configuration symbolique de $TS(S)$ une paire $(q, EMS) \in Q \times \mathcal{EMS}_{P,X}$.

Nous définissons alors le *problème d'accessibilité symbolique* de la façon suivante :

Entrées : Un système à pointeurs $S = \langle Q, P, E\rangle$, une configuration initiale $c_0 \in Q \times \mathcal{GM}_P^{err}$ et une configuration symbolique $(q, EMS) \in Q \times \mathcal{EMS}_{P,X}$,
Question : Existe-t-il $GM \in [\![EMS]\!]$ tel que $(q, GM) \in$ **Reach**(S, c_0) ?

Comme nous l'avons déjà signalé dans la partie sur les systèmes à compteurs, il est aussi important de pouvoir vérifier des propriétés d'un programme en considérant non pas une unique configuration initiale mais une infinité de configurations initiales possibles. Ainsi étant donnés un système à pointeurs $S = \langle Q, P, E\rangle$ et une configuration symbolique initiale (q_0, EMS_0) dans $Q \times \mathcal{EMS}_{P,X}$, nous étendons la définition d'ensemble d'accessibilité et définissons l'ensemble d'accessibilité symbolique **Reach**$(S, (q_0, EMS_0)) = \{(q, GM) \in Q \times \mathcal{GM}_P^{err} \mid \exists GM_0 \in [\![EMS_0]\!]$ tel que $(q, GM) \in$ **Reach**$(S, (q_0, GM_0))\}$. Ceci nous permet de définir *le problème d'accessibilité généralisé d'une erreur de segmentation* :

Entrées : Un système à pointeurs $S = \langle Q, P, E\rangle$ et une configuration symbolique initiale (q_0, EMS_0) dans $Q \times \mathcal{EMS}_{P,X}$,

Question : Existe-t-il $q \in Q$ tel que $(q, \texttt{Seg}) \in$ **Reach**$(S, (q_0, EMS_0))$ ou tel que $(q, \texttt{LeakSeg}) \in$ **Reach**$(S, (q_0, EMS_0))$?

De la même façon, nous définissons le *le problème d'accessibilité généralisé d'une fuite mémoire* :

Entrées : Un système à pointeurs $S = \langle Q, P, E \rangle$ et une configuration symbolique initiale (q_0, EMS_0) dans $Q \times \mathcal{EMS}_{P,X}$,
Question : Existe-t-il $q \in Q$ tel que $(q, \texttt{Leak}) \in$ **Reach**$(S, (q_0, EMS_0))$ ou tel que $(q, \texttt{LeakSeg}) \in$ **Reach**$(S, (q_0, EMS_0))$?

Nous définissons finalement *le problème d'accessibilité symbolique généralisé* :

Entrées : Un système à pointeurs $S = \langle Q, P, E \rangle$, deux configurations symboliques (q_0, EMS_0) et (q, EMS) dans $Q \times \mathcal{EMS}_{P,X}$,
Question : Existe-t-il $GM \in [\![EMS]\!]$ tel que $(q, GM) \in$ **Reach**$(S, (q_0, EMS_0))$?

Notons que le problème d'accessibilité symbolique est un cas particulier du problème d'accessibilité symbolique généralisé. De même, les problème d'accessibilité d'une erreur de segmentation et d'une fuite mémoire, que nous avons définis à la section 4.2.4, sont des cas particuliers des problèmes d'accessibilité généralisés d'une erreur de segmentation et d'une fuite mémoire. Ce qu'il est plus intéressant de remarquer est que les problèmes d'accessibilité généralisés d'une erreur de segmentation ou d'une fuite mémoire, peuvent eux aussi se ramener à des problèmes d'accessibilité symbolique. Ainsi, si nous avons des résultats de décidabilité pour ce dernier problème, ils impliqueront des résultats de décidabilité pour les problèmes d'accessibilité généralisés d'une erreur de segmentation ou d'une fuite mémoire

Proposition 4.23 *Les problèmes d'accessibilité généralisés d'une erreur de segmentation et d'une fuite mémoire peuvent se ramener à la vérification d'un nombre fini d'instances du problème d'accessibilité symbolique généralisé.*

Preuve : Nous considérons un système à pointeurs $S = (Q, P, E)$ et une configuration symbolique initiale (q_0, EMS_0) dans $Q \times \mathcal{EMS}$. Nous montrons le résultat pour le problème d'accessibilité généralisé d'une erreur de segmentation, qui consiste à savoir si il existe une configuration initiale $c_0 \in \{q_0\} \times [\![EMS_0]\!]$ et un état de contrôle $q \in Q$ tel que $(q, \texttt{Seg}) \in$ **Reach**(S, c_0) ou $(q, \texttt{LeakSeg}) \in$ **Reach**(S, c_0). Tout d'abord remarquons qu'étant donnée une garde de pointeurs $g \in \mathcal{G}$, on peut construire un état mémoire symbolique EMS_g tel que pour tout graphe mémoire $GM \in \mathcal{GM}_P$, $GM \models g$ si et seulement si $GM \in [\![EMS_g]\!]$. Avant de donner la construction de EMS_g, pour une forme mémoire FM, nous définissons la formule de Presburger $\phi_{X_{FM}}$ égale à $\bigwedge_{x \in X_{FM}} x > 0$. Nous construisons par induction sur g, l'état mémoire symbolique EMS_g :

- si $g = true$, alors $EMS_g = \{(FM_i, \phi_{X_{FM_i}}) \mid i \in [1..m]\}$ tel que pour tout $FM \in \mathcal{FM}_{P,X}$ il existe $i \in [1..m]$ tel que $FM \approx FM_i$,
- si $g = (p == \texttt{null})$, alors $EMS_g = \{(FM_i, \phi_{X_{FM_i}}) \mid i \in [1..m]\}$ tel que pour tout $FM \in \mathcal{FM}_{P,X}$ avec $FM = (N, P, X, succ, loc, c)$ si $loc(p) = \texttt{null}$ alors il existe $i \in [1..m]$ tel que $FM \approx FM_i$ et si $loc(p) \neq \texttt{null}$ alors il n'existe pas $i \in [1..m]$ tel que $FM \approx FM_i$,
- si $g = (p == p')$, alors $EMS_g = \{(FM_i, \phi_{X_{FM_i}}) \mid i \in [1..m]\}$ tel que pour tout $FM \in \mathcal{FM}_{P,X}$ avec $FM = (N, P, X, succ, loc, c)$ si $loc(p) = loc(p')$ alors il existe $i \in [1..m]$ tel que $FM \approx FM_i$ et si $loc(p) \neq loc(p')$ alors il n'existe pas $i \in [1..m]$ tel que $FM \approx FM_i$,

– si $g = \neg g'$ alors $EMS_g = \neg EMS'_g$,
– si $g = g' \wedge g''$ alors $EMS_g = EMS_{g'} \sqcap EMS_{g''}$.

De la même façon, si a est une action de pointeurs, nous construisons un état mémoire symbolique $EMS_{\texttt{Seg},a}$ tel que pour tout graphe mémoire symbolique $GM \in \mathcal{GM}_P$, $[\![a]\!]_P(GM) = \texttt{Seg}$ ou $[\![a]\!]_P(GM) = \texttt{LeakSeg}$ si et seulement si $GM \in [\![EMS_{\texttt{Seg},a}]\!]$. Nous définissons $EMS_{\texttt{Seg},a}$ de la façon suivante :

– si $a = (p := p')$ ou $a = (p := \texttt{null})$ ou $a = (\texttt{malloc}(p))$, alors nous définissons $EMS_{a,\texttt{Seg}} = \{(FM_i, \phi_{X_{FM_i}}) \mid i \in [1..m]\}$ tel que pour tout $FM \in \mathcal{FM}_{P,X}$ il existe $i \in [1..m]$ tel que $FM \approx FM_i$, (en effet les actions $p := p'$, $p := \texttt{null}$ et $\texttt{malloc}(p)$ ne réalisent jamais d'erreur de segmentation comme indiqué aux figures 4.5,4.6 et 4.9),
– si $a = (p := p'.\texttt{succ})$ ou $a = (p'.\texttt{succ} := p)$ ou $a = (\texttt{free}(p))$, alors nous définissons $EMS_{a,\texttt{Seg}} = \{(FM_i, \phi_{X_{FM_i}}) \mid i \in [1..m]\}$ tel que pour tout $FM \in \mathcal{FM}_{P,X}$ avec $FM = (N, P, X, succ, loc, c)$, si $loc(p') = \texttt{null}$ ou $loc(p') = \perp$ alors il existe $i \in [1..m]$ tel que $FM \approx FM_i$ et si $loc(p') \neq \texttt{null}$ et $loc(p') \neq \perp$ alors il n'existe pas $i \in [1..m]$ tel que $FM \approx FM_i$ (en effet, ces actions réalisent une erreur de segmentation uniquement lorsque la variable p' pointe sur le noeud $\texttt{null}$ ou le noeud $\perp$ comme indiqué sur les figures 4.7, 4.8 et 4.10).

De la même façon, nous pouvons construire pour toute action de pointeurs a, un état mémoire symbolique $EMS_{\texttt{Leak},a}$ tel que pour tout graphe mémoire symbolique $GM \in \mathcal{GM}_P$, $[\![a]\!]_P(GM) = \texttt{Leak}$ ou $[\![a]\!]_P(GM) = \texttt{LeakSeg}$ si et seulement si $GM \in [\![EMS_{\texttt{Leak},a}]\!]$ (nous ne détaillons pas la construction qui ceci dit est un peu plus compliquée que pour l'erreur de segmentation, car il faut tester parfois des conditions sur les compteurs, en particulier pour l'action $p.\texttt{succ} = p'$ qui réalise une fuite mémoire si la valeur du compteur associé au noeud pointé par p est strictement supérieure à 1, ceci dit mise à part ce point la construction se fait de manière tout à fait similaire au cas de l'erreur de segmentation).
Soit q un état de contrôle de Q, nous construisons l'état mémoire symbolique

$$EMS_{\texttt{Seg},q} = \bigsqcup_{(q,g,a,q') \in E} EMS_g \sqcap EMS_{a,\texttt{Seg}}$$

Ainsi nous pouvons conclure qu'il y aura une erreur de segmentation dans S muni de la configuration symbolique initiale (q_0, EMS_0) si et seulement si il existe une configuration initiale c_0 dans $\{q_0\} \times [\![EMS_0]\!]$ et un état de contrôle q tel que la configuration symbolique $(q, EMS_{\texttt{Seg},q})$ est accessible dans (S, c_0) (c'est-à-dire si le problème d'accessibilité symbolique généralisé retourne *vrai* avec les instances S, (q_0, EMS_0) et $(q, EMS_{\texttt{Seg},q})$). Ainsi le nombre d'états de contrôle d'un système à pointeurs étant fini, le problème d'accessibilité d'une erreur de segmentation se ramène à un nombre fini d'instances du problème d'accessibilité symbolique généralisé. Il en va de même pour le problème d'accessibilité généralisé d'une fuite mémoire. □

De façon identique, les problèmes d'accessibilité d'un état de contrôle et d'une configuration sont des cas particuliers de problèmes d'accessibilité symbolique. Par conséquent, le théorème 4.9 nous indique que ces deux nouveaux problèmes d'accessibilité sont eux aussi indécidables pour les systèmes à pointeurs manipulant au moins 3 variables de pointeurs.

Chapitre 5

Une traduction vers les systèmes à compteurs

Après avoir introduit les systèmes à pointeurs et une possible représentation symbolique pour caractériser leurs configurations, nous présentons dans ce chapitre une méthode pour les analyser. Cette méthode est basée sur la construction d'un système à compteurs bisimilaire. Nous pouvons ensuite utiliser les méthodes déjà existantes pour l'analyse de systèmes à compteurs pour vérifier des propriétés sur le système à pointeurs considéré. Les résultats présentés ici sont issus de [BFLS06]. Signalons au lecteur que la méthode présentée est similaire à celle proposée dans [BBH$^+$06], mais que ces deux travaux ont été réalisés de façon indépendante.

5.1 Traduction vers un système à compteurs bisimilaire

5.1.1 Présentation de la traduction

Nous présentons ici une traduction qui, étant donné un système à pointeurs le transforme en un système à compteurs linéaire bisimilaire. Cette traduction nous permettra d'utiliser l'outil FAST pour résoudre des problèmes tels que l'accessibilité d'une erreur de segmentation ou d'une fuite mémoire pour des systèmes à pointeurs. Dans ce chapitre, nous considérons un ensemble P de pointeurs et un ensemble X de compteurs tel que $2|P| \leq |X|$ de façon à pouvoir représenter tout graphe mémoire grâce à une forme mémoire (cf. proposition 4.15).

L'idée de la traduction est la suivante, étant donnés une forme mémoire et une action de pointeurs, il est possible d'exécuter symboliquement cette action sur la forme mémoire. On obtient alors un ensemble des formes mémoire ainsi qu'un ensemble d'opérations sur les compteurs. Les formes mémoire obtenues correspondent aux formes mémoire possibles après exécution de l'action, quant aux opérations sur les compteurs elles permettent de tester les valeurs des différents compteurs et de les mettre ensuite à jour. Formellement, nous définissons la fonction **POST** : $\mathcal{A}_P \times \mathcal{FM}_{P,X} \rightarrow \mathcal{P}(\textbf{Presb}(X, X') \times \mathcal{FM}^{err}_{P,X})$ avec $\mathcal{FM}^{err}_{P,X} = \mathcal{FM}_{P,X} \cup \{\texttt{Seg}, \texttt{Leak}, \texttt{LeakSeg}\}$. Ainsi étant donnés une forme mémoire FM et une action de pointeurs $a \in \mathcal{A}_P$, **POST**(a, FM) est un ensemble de paires (ϕ, FM'), chacune étant constituée d'une formule de Presburger et d'une forme mémoire ou d'un élément de l'ensemble $\{\texttt{Seg}, \texttt{Leak}, \texttt{LeakSeg}\}$ caractérisant une erreur. Nous souhaitons de plus que la propriété suivante soit vérifiée : pour toute paire $(\phi, FM') \in$ **POST**(a, FM), avec $FM' \in \mathcal{FM}_{P,X}$,

pour toutes valuations $\mathbf{v}, \mathbf{v}' : X \to \mathbb{N}$ telles que $\mathbf{v}(X_{FM}) \subseteq \mathbb{N}^*$ et $\mathbf{v}'(X_{FM'}) \subseteq \mathbb{N}^*$, nous avons :

$$[\![a]\!]_P(FM(\mathbf{v}_{FM})) = FM'(\mathbf{v}'_{FM'}) \text{ si et seulement si } (\mathbf{v}, \mathbf{v}') \models \phi$$

où $\mathbf{v}_{FM}$ représente la restriction de $\mathbf{v}$ aux compteurs dans X_{FM}, c'est-à-dire aux compteurs étiquetant la forme mémoire FM et $\mathbf{v}'_{FM'}$ représente la restriction de $\mathbf{v}'$ aux compteurs dans $X_{FM'}$. Rappelons de plus que comme $\mathbf{v}(X_{FM}) \subseteq \mathbb{N}^*$, la paire $(FM, \mathbf{v}_{FM})$ est une forme mémoire valuée à laquelle on associe l'unique graphe mémoire $FM(\mathbf{v}_{FM})$, comme cela est expliqué dans la section 4.3 du chapitre précédent.

Les figures 5.1 à 5.7 donnent la définition de la fonction **POST** en fonction des différentes actions de pointeurs possibles. Sur chacune de ces figures, pour chaque forme mémoire possible, on donne les paires (ϕ, FM') appartenant à l'ensemble **POST**(a, FM). Pour l'action `skip`, nous avons **POST**$(\texttt{skip}, FM) = \{(\bigwedge_{x \in X} x' = x, FM)\}$.

Hypothèses	**POST**(a, FM)	
	ϕ	FM'
$FM = (N, P, X, succ, loc, c)$ et $p \neq p'$ et soit $\lvert succ^{-1}(\{loc(p)\})\rvert \geq 2$, soit $\lvert loc^{-1}(\{loc(p)\})\rvert \geq 2$, soit $loc(p) \in \{\texttt{null}, \bot\}$	$\bigwedge_{x \in X} x' = x$	$(N, P, X, succ, loc[p \mapsto loc(p')], c)$
$p = p'$	$\bigwedge_{x \in X} x' = x$	FM
$FM = (N, P, X, succ, loc, c)$ et $p \neq p'$ et $succ^{-1}(\{loc(p)\}) = \{n\}$ et $n \neq loc(p)$ et $y = c(n)$ et $z = c(loc(p))$ et $\lvert loc^{-1}(\{loc(p)\})\rvert = 1$, et $loc(p) \notin \{\texttt{null}, \bot\}$	$y' = y + z \wedge$ $z' = 0 \wedge$ $\bigwedge_{x \in X \setminus \{y,z\}} x' = x$	$(N \setminus \{loc(p)\}, P, X, succ', loc[p \mapsto loc(p')], c')$ avec $succ'(n) = succ(loc(p))$ $succ'_{\lvert N \setminus \{loc(p), n\}} = succ_{\lvert N \setminus \{loc(p), n\}}$ $c' = c_{\lvert N \setminus \{loc(p)\}}$
$FM = (N, P, X, succ, loc, c)$ et $p \neq p'$ et $\lvert loc^{-1}(\{loc(p)\})\rvert = 1$ et $loc(p) \notin \{\texttt{null}, \bot\}$ et soit $\lvert succ^{-1}(\{loc(p)\})\rvert = 0$ soit $succ^{-1}(\{loc(p)\}) = \{loc(p)\}$	$\bigwedge_{x \in X} x' = x$	`Leak`

FIGURE 5.1 – Définition de **POST** pour l'action $a = (p := p')$

Hypothèses	**POST**(a, FM)	
	ϕ	FM'
$FM = (N, P, X, succ, loc, c)$ et soit $\lvert succ^{-1}(\{loc(p)\})\rvert \geq 2$, soit $\lvert loc^{-1}(\{loc(p)\})\rvert \geq 2$, soit $loc(p) \in \{\texttt{null}, \bot\}$	$\bigwedge_{x \in X} x' = x$	$(N, P, X, succ, loc[p \mapsto \texttt{null}], c)$
$FM = (N, P, X, succ, loc, c)$ et $succ^{-1}(\{loc(p)\}) = \{n\}$ et $n \neq loc(p)$ et $y = c(n)$ et $z = c(loc(p))$ et $\lvert loc^{-1}(\{loc(p)\})\rvert = 1$, et $loc(p) \notin \{\texttt{null}, \bot\}$	$y' = y + z \wedge$ $z' = 0 \wedge$ $\bigwedge_{x \in X \setminus \{y,z\}} x' = x$	$(N \setminus \{loc(p)\}, P, X, succ', loc[p \mapsto \texttt{null}], c')$ avec $succ'(n) = succ(loc(p))$ $succ'_{\lvert N \setminus \{loc(p), n\}} = succ_{\lvert N \setminus \{loc(p), n\}}$ $c' = c_{\lvert N \setminus \{loc(p)\}}$
$FM = (N, P, X, succ, loc, c)$ et $\lvert loc^{-1}(\{loc(p)\})\rvert = 1$ et $loc(p) \notin \{\texttt{null}, \bot\}$ et soit $\lvert succ^{-1}(\{loc(p)\})\rvert = 0$ soit $succ^{-1}(\{loc(p)\}) = \{loc(p)\}$	$\bigwedge_{x \in X} x' = x$	`Leak`

FIGURE 5.2 – Définition de **POST** pour l'action $a = (p := \texttt{null})$

Hypothèses	**POST**(a, FM)	
	ϕ	FM'
$FM = (N, P, X, succ, loc, c)$ et $loc(p') \notin \{\texttt{null}, \bot\}$ et $y = c(loc(p'))$ et $z \in X \setminus X_{FM}$ et soit $\lvert succ^{-1}(\{loc(p)\})\rvert \geq 2$ soit $\lvert loc^{-1}(\{loc(p)\})\rvert \geq 2$ soit $loc(p) \in \{\texttt{null}, \bot\}$	$y = 1 \wedge$ $\bigwedge_{x \in X} x' = x$	$(N, P, X, succ, loc', c)$ avec $loc' = loc[p \mapsto succ(loc(p'))]$
	$y > 1 \wedge$ $y' = 1 \wedge z' = y - 1 \wedge$ $\bigwedge_{x \in X \setminus \{y,z\}} x' = x$	$(N \uplus \{newn\}, P, X, succ', loc', c')$ avec $c' = c[newn \mapsto z]$ $succ'(loc(p')) = newn$ $succ'(newn)) = succ(loc(p')$ $succ'_{\lvert N \setminus \{loc(p')\}} = succ_{\lvert N \setminus \{loc(p')\}}$ $loc' = loc[p \mapsto newn]$
$FM = (N, P, X, succ, loc, c)$ et $loc(p') \notin \{\texttt{null}, \bot\}$ et $loc(p) \notin \{\texttt{null}, \bot\}$ et $y = c(loc(p'))$ et $z = c(loc(p))$ et $succ^{-1}(\{loc(p)\}) = \{n\}$ et $n \neq loc(p)$ et $c(n) = w$ et $\lvert loc^{-1}(\{loc(p)\})\rvert = 1$ et $succ(loc(p')) \neq loc(p)$	$y = 1 \wedge z' = 0 \wedge$ $w' = w + z \wedge$ $\bigwedge_{x \in X \setminus \{w,z\}} x' = x$	$(N \setminus \{loc(p)\}, P, X, succ', loc', c')$ avec $loc' = loc[p \mapsto succ(loc(p'))]$ $c' = c_{\lvert N \setminus \{loc(p)\}}$ $succ'(n) = succ(loc(p))$ $succ'_{\lvert N \setminus \{n, loc(p)\}} = succ_{\lvert N \setminus \{n, loc(p)\}}$
	$y > 1 \wedge$ $y' = 1 \wedge z' = y - 1 \wedge$ $w' = w + z \wedge$ $\bigwedge_{x \in X \setminus \{w,y,z\}} x' = x$	$(N, P, X, succ', loc, c)$ avec $succ'(n) = succ(loc(p))$ $succ'(loc(p')) = loc(p)$ $succ'(loc(p)) = succ(loc(p'))$ $succ'_{\lvert N \setminus \{n, loc(p), loc(p')\}} = succ_{\lvert N \setminus \{n, loc(p), loc(p')\}}$
$FM = (N, P, X, succ, loc, c)$ et $loc(p') \notin \{\texttt{null}, \bot\}$ et $loc(p) \notin \{\texttt{null}, \bot\}$ et $y = c(loc(p'))$ et $z = c(loc(p))$ et $\lvert succ^{-1}(\{loc(p)\})\rvert = 1$ et $\lvert loc^{-1}(\{loc(p)\})\rvert = 1$ et $succ(loc(p')) = loc(p)$ et $p \neq p'$	$y' = 1 \wedge$ $z' = z + y - 1 \wedge$ $\bigwedge_{x \in X \setminus \{y,z\}} x' = x$	FM

FIGURE 5.3 – Définition de **POST** pour l'action $a = (p := p'.\texttt{succ})$ *(début)*

Hypothèses	**POST**(a, FM)	
	ϕ	FM'
$FM = (N, P, X, succ, loc, c)$ et $loc(p') \notin \{\texttt{null}, \bot\}$ et $loc(p) \notin \{\texttt{null}, \bot\}$ et $\lvert succ^{-1}(\{loc(p)\})\rvert = 1$ et $\lvert loc^{-1}(\{loc(p)\})\rvert = 1$ et $succ(loc(p')) = loc(p)$ et $p = p'$	$\bigwedge_{x \in X} x' = x$	FM
$FM = (N, P, X, succ, loc, c)$ et $loc(p') \notin \{\texttt{null}, \bot\}$ et $loc(p) \notin \{\texttt{null}, \bot\}$ et $\lvert loc^{-1}(\{loc(p)\})\rvert = 1$ et soit $\lvert succ^{-1}(\{loc(p)\})\rvert = 0$ soit $succ^{-1}(\{loc(p)\}) = \{loc(p)\}$ avec $p \neq p'$	$\bigwedge_{x \in X} x' = x$	`Leak`
$FM = (N, P, X, succ, loc, c)$ et $loc(p') \in \{\texttt{null}, \bot\}$ et $loc(p) \notin \{\texttt{null}, \bot\}$ et $\lvert loc^{-1}(\{loc(p)\})\rvert = 1$ et soit $\lvert succ^{-1}(\{loc(p)\})\rvert = 0$ soit $succ^{-1}(\{loc(p)\}) = \{loc(p)\}$ avec $p \neq p'$	$\bigwedge_{x \in X} x' = x$	`LeakSeg`
$FM = (N, P, X, succ, loc, c)$ et $loc(p') \in \{\texttt{null}, \bot\}$ et soit $\lvert loc^{-1}(\{loc(p)\})\rvert > 1$ soit $loc(p) \in \{\texttt{null}, \bot\}$ soit $\lvert succ^{-1}(\{loc(p)\})\rvert \geq 2$ soit $succ^{-1}(\{loc(p)\}) = \{n\}$ avec $n \neq loc(p)$	$\bigwedge_{x \in X} x' = x$	`Seg`

FIGURE 5.4 – Définition de **POST** pour l'action $a = (p := p'.\texttt{succ})$ *(suite)*

Hypothèses	**POST**(a, FM)	
	ϕ	FM'
$FM = (N, P, X, succ, loc, c)$ et $y = c(loc(p))$ et $loc(p) \notin \{\texttt{null}, \bot\}$ et soit $\lvert succ^{-1}(\{succ(loc(p))\})\rvert > 2$ soit $\lvert loc^{-1}(\{succ(loc(p))\})\rvert \geq 1$ soit $succ(loc(p)) \in \{\texttt{null}, \bot\}$	$y = 1 \wedge$ $\bigwedge_{x \in X} x' = x$	$(N, P, X, succ[loc(p) \mapsto loc(p')], loc, c)$
	$y > 1 \wedge$ $\bigwedge_{x \in X} x' = x$	`Leak`
$FM = (N, P, X, succ, loc, c)$ et $y = c(loc(p))$ et $loc(p) \notin \{\texttt{null}, \bot\}$ et $succ(loc(p)) \notin \{\texttt{null}, \bot\}$ et $succ^{-1}(\{succ(loc(p))\}) = \{m, loc(p)\}$ et $m \neq succ(loc(p))$ et $m \neq loc(p)$ et $\lvert loc^{-1}(\{succ(loc(p))\})\rvert = 0$ et $n = succ(loc(p)$ et $w = c(n)$ et $z = c(m)$	$y = 1 \wedge w' = 0 \wedge$ $z' = w + z \wedge$ $\bigwedge_{x \in X} x' = x$	$(N \setminus \{n\}, P, X, succ', loc, c_{\lvert N \setminus \{n\}})$ avec $succ'(loc(p)) = loc(p')$ $succ'(m) = succ(n)$ $succ'_{\lvert N \setminus \{n, loc(p), m\}} = succ_{\lvert N \setminus \{n, loc(p), m\}}$
	$y > 1 \wedge$ $\bigwedge_{x \in X} x' = x$	`Leak`
$FM = (N, P, X, succ, loc, c)$ et $loc(p) \in \{\texttt{null}, \bot\}$	$\bigwedge_{x \in X} x' = x$	`Seg`
$FM = (N, P, X, succ, loc, c)$ et $loc(p) \notin \{\texttt{null}, \bot\}$ et $succ(loc(p)) \notin \{\texttt{null}, \bot\}$ et $loc(p) \neq loc(p')$ et $succ^{-1}(\{succ(loc(p))\}) = \{m, loc(p)\}$ et $m = succ(loc(p))$ et $m \neq loc(p)$ et $\lvert loc^{-1}(\{succ(loc(p))\})\rvert = 0$	$\bigwedge_{x \in X} x' = x$	`Leak`

FIGURE 5.5 – Définition de **POST** pour l'action $a = (p.\texttt{succ} := p')$

Hypothèses	**POST**(a, FM)	
	ϕ	FM'
$FM = (N, P, X, succ, loc, c)$ et $y \in X \setminus X_{FM}$ et soit $\lvert succ^{-1}(\{loc(p)\})\rvert \geq 2$, soit $\lvert loc^{-1}(\{loc(p)\})\rvert \geq 2$, soit $loc(p) \in \{\texttt{null}, \bot\}$	$y' = 1 \wedge$ $\bigwedge_{x \in X \setminus \{y\}} x' = x$	$(N \uplus n, P, X, succ', loc[p \mapsto n], c')$ avec $succ' = succ[n \mapsto \bot]$ $c' = c[n \mapsto y]$
$FM = (N, P, X, succ, loc, c)$ et $succ^{-1}(\{loc(p)\}) = \{n\}$ et $n \neq loc(p)$ et $y = c(n)$ et $z = c(loc(p))$ et $\lvert loc^{-1}(\{loc(p)\})\rvert = 1$, et $loc(p) \notin \{\texttt{null}, \bot\}$	$y' = y + z \wedge$ $z' = 1 \wedge$ $\bigwedge_{x \in X \setminus \{y,z\}} x' = x$	$(N, P, X, succ'[loc(p) \mapsto \bot], c)$ avec $succ'(loc(p)) = \bot$ $succ'(n) = succ(loc(p))$ $succ'_{\lvert N \setminus \{loc(p)), n\}} = succ_{\lvert N \setminus \{loc(p)), n\}}$
$FM = (N, P, X, succ, loc, c)$ et $loc(p) \notin \{\texttt{null}, \bot\}$ et $\lvert loc^{-1}(\{loc(p)\})\rvert = 1$ et soit $\lvert succ^{-1}(\{loc(p)\})\rvert = 0$ soit $succ^{-1}(\{loc(p)\}) = \{loc(p)\}$	$\bigwedge_{x \in X} x' = x$	`Leak`

FIGURE 5.6 – Définition de **POST** pour l'action $a = (p := \texttt{malloc})$

Hypothèses	**POST**(a, FM)	
	ϕ	FM'
$FM = (N, P, X, succ, loc, c)$ et $y = c(loc(p))$ et $loc(p) \notin \{\texttt{null}, \bot\}$ et soit $\lvert succ^{-1}(succ(loc(p)))\rvert > 2$ soit $\lvert loc^{-1}(succ(loc(p)))\rvert \geq 1$ soit $succ(loc(p)) \in \{\texttt{null}, \bot\}$	$y = 1 \wedge y' = 0 \wedge$ $\bigwedge_{x \in X} x' = x$	$(N \setminus \{loc(p)\}, P, X, succ', loc', c_{N \setminus \{loc(p)\}})$ et pour tout $p' \in P$ si $loc(p') = loc(p)$ alors $loc'(p') = \bot$ si $loc(p') \neq loc(p)$ alors $loc'(p') = loc(p')$ pour tout $n \in N \setminus \{loc(p)\}$ si $succ(n) = loc(p)$ alors $succ'(n) = \bot$ si $succ(n) \neq loc(p)$ alors $succ'(n) = succ(n)$
	$y > 1 \wedge$ $\bigwedge_{x \in X} x' = x$	`Leak`
$FM = (N, P, X, succ, loc, c)$ et $y = c(loc(p))$ et $loc(p) \notin \{\texttt{null}, \bot\}$ et $succ(loc(p)) \notin \{\texttt{null}, \bot\}$ et $succ^{-1}(succ(loc(p))) = \{m, loc(p)\}$ et $m \neq succ(loc(p))$ et $m \neq loc(p)$ et $\lvert loc^{-1}(succ(loc(p)))\rvert = 0$ et $n = succ(loc(p)$ et $w = c(n)$ et $z = c(m)$	$y = 1 \wedge y' = 0 \wedge$ $w' = 0 \wedge$ $z' = w + z \wedge$ $\bigwedge_{x \in X} x' = x$	$(N \setminus \{loc(p), n\}, P, X, succ', loc', c_{\lvert N \setminus \{n, loc(p)\}})$ et pour tout $p' \in P$ si $loc(p') = loc(p)$ alors $loc'(p') = \bot$ si $loc(p') \neq loc(p)$ alors $loc'(p') = loc(p')$ $succ'(m) = succ(n)$ et pour tout $l \in N \setminus \{m, loc(p)\}$ si $succ(l) = loc(p)$ alors $succ'(l) = \bot$ si $succ(l) \neq loc(p)$ alors $succ'(l) = succ(l)$
	$y > 1 \wedge$ $\bigwedge_{x \in X} x' = x$	`Leak`
$FM = (N, P, X, succ, loc, c)$ et $loc(p) \in \{\texttt{null}, \bot\}$	$\bigwedge_{x \in X} x' = x$	`Seg`
$FM = (N, P, X, succ, loc, c)$ et $loc(p) \notin \{\texttt{null}, \bot\}$ et $succ(loc(p)) \notin \{\texttt{null}, \bot\}$ et $succ^{-1}(succ(loc(p))) = \{m, loc(p)\}$ et $m = succ(loc(p))$ et $m \neq loc(p)$ et $\lvert loc^{-1}(succ(loc(p)))\rvert = 0$	$\bigwedge_{x \in X} x' = x$	`Leak`

FIGURE 5.7 – Définition de **POST** pour l'action $a = (free(p))$

Exemple 5.1 *La figure 5.8 donne un exemple de l'application de la fonction* **POST** *avec comme arguments l'action de pointeurs* $p_1.\texttt{succ} := p_2$ *et la forme mémoire* FM*. Ainsi si nous considérons une valuation* $\mathbf{v} : X_{FM} \to \mathbb{N}^*$ *et le graphe mémoire correspondant* $FM(\mathbf{v})$*, nous voyons que si la valeur du compteur* x_1 *est strictement plus grande que* 1*, alors l'action* $p_1.\texttt{succ} := p_2$ *conduira à une fuite mémoire, ceci car nous savons alors que dans le graphe mémoire* $FM(\mathbf{v})$ *il existe au moins un noeud accessible par* p_1 *et pas par* p_2*. En revanche, si la valeur associée au compteur* x_1 *est égale à* 1*, l'action* $p_1.\texttt{succ} := p_2$ *ne réalise pas d'erreur, le noeud pointé par* p_2 *devient alors le successeur du noeud pointé par* p_1*, et comme en faisant cela, le noeud qui était auparavant le successeur du noeud pointé par* p_1 *, devient un noeud avec un seul prédécesseur et qui n'est pas étiqueté par une variable, on le supprime pour obtenir de nouveau une forme mémoire, et les différents compteurs sont mis à jour en tenant compte de cette modification du graphe.*

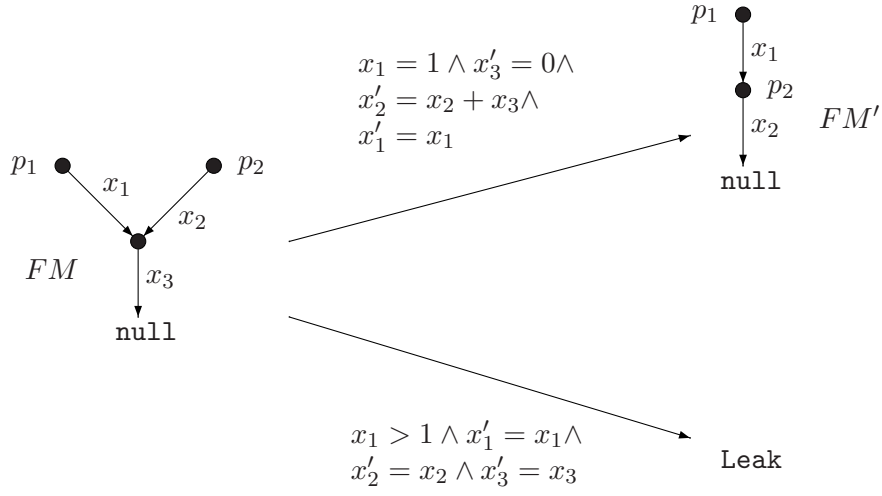


FIGURE 5.8 – La fonction **POST** appliquée à l'action $p_1.\texttt{succ} := p_2$ et à FM

Exemple 5.2 *La figure 5.9 donne un autre exemple de l'application de la fonction* **POST** *avec cette fois-ci comme arguments l'action de pointeurs* $p_1 := p_2.\texttt{succ}$ *et la forme mémoire* FM*. Nous constatons que l'application de l'action* **POST** *fournit deux formes mémoire distinctes. En effet, si dans un cas,* `null` *est le successeur direct de* p_2*, c'est-à-dire si la valeur du compteur* x_2 *vaut* 1*, alors* p_1 *doit pointer sur le noeud* `null` *si en revanche ce n'est pas le cas,* p_1 *doit pointer vers un noeud entre* $loc(p_2)$ *et* `null`*, et il est nécessaire d'inclure ce noeud et d'ajouter un compteur (ici* x_1*) à la forme mémoire symbolique.*

Nous définissons maintenant la relation $\sim \subseteq \mathcal{GM} \times (\mathcal{FM} \times \mathbb{N}^X)$ entre des graphes mémoire et des paires constituées d'une forme mémoire et d'une valuation de compteurs. Pour tout $GM \in \mathcal{GM}$ et tout $(FM, \mathbf{v}) \in \mathcal{FM} \times \mathbb{N}^X$, nous avons $GM \sim (FM, \mathbf{v})$ si et seulement si :

$$\mathbf{v}(X_{FM}) \subseteq \mathbb{N}^* \text{ et } GM = FM(\mathbf{v}_{|X_{FM}})$$

Nous étendons ensuite la relation $\sim$ aux paires dans $\mathcal{GM}^{err} \times (\mathcal{FM}^{err} \times \mathbb{N}^X)$ de telle façon que pour toute valuation $\mathbf{v} \in \mathbb{N}^X$, $\texttt{Seg} \sim (\texttt{Seg}, \mathbf{v})$, $\texttt{Leak} \sim (\texttt{Leak}, \mathbf{v})$ et $\texttt{LeakSeg} \sim (\texttt{LeakSeg}, \mathbf{v})$. Nous avons alors la propriété suivante :

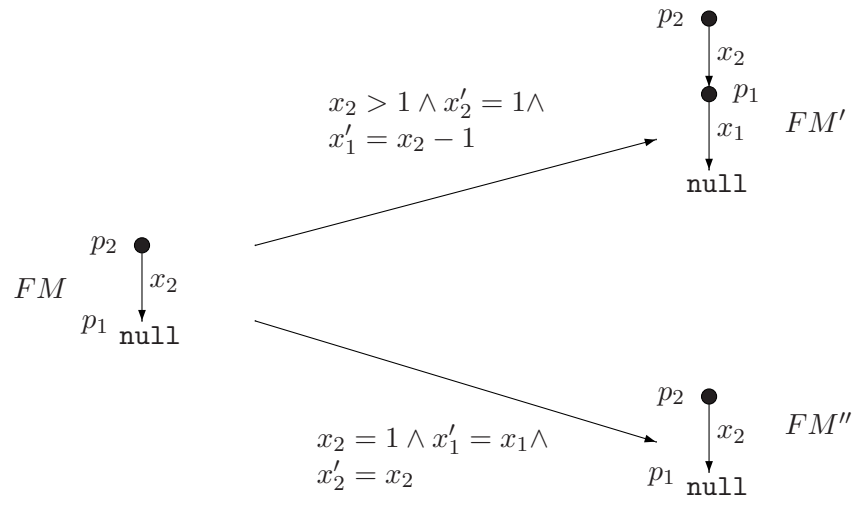


FIGURE 5.9 – La fonction **POST** appliquée à l'action $p_1 := p_2.\texttt{succ}$ et à FM

Lemme 5.3 *Soient $GM \in \mathcal{GM}$, $a \in \mathcal{A}$ une action de pointeurs et $(FM, \mathbf{v}) \in \mathcal{FM} \times \mathbb{N}^X$ tel que $GM \sim (FM, \mathbf{v})$. Pour tout $GM' \in \mathcal{GM}^{err}$, nous avons $GM' = [\![a]\!]_P(GM)$ si et seulement si il existe $(FM', \mathbf{v}') \in \mathcal{FM}^{err} \times \mathbb{N}^X$ et $\phi \in$ **Presb**(X, X') tels que :*

- *$GM' \sim (FM', \mathbf{v}')$, et,*
- *$(\phi, FM') \in$ **POST**(a, FM), et,*
- *$(\mathbf{v}, \mathbf{v}') \models \phi$.*

Idée de la preuve : Ce lemme est en fait une conséquence directe de la construction de la fonction **POST**(a, FM) et de l'application des définitions de $[\![a]\!]_P(GM)$ et de $GM \sim (FM, \mathbf{v})$. Nous montrons sur des cas particuliers les principales étapes à suivre pour obtenir le résultat énoncé par ce lemme.

Soient $GM \in \mathcal{GM}$, $a \in \mathcal{A}$ et $(FM, \mathbf{v}) \in \mathcal{FM} \times \mathbb{N}^X$ tel que $GM \sim (FM, \mathbf{v})$. Nous posons $FM = (N, P, X, succ, loc, c)$. Par définition de $\sim$, nous avons $GM = FM(\mathbf{v}_{|X_{FM}})$.

Nous considérons dans un premier temps l'action $p := p'$ (avec $p \neq p'$). Soit $GM' \in \mathcal{GM}^{err}$ tel que $GM' = [\![a]\!]_P(GM)$.

- Supposons que $GM' \neq \texttt{Leak}$. Nous sommes alors dans le cas présenté à la première ligne du tableau de la figure 4.5. En accord avec la façon dont nous construisons le graphe mémoire $GM = FM(\mathbf{v}_{|X_{FM}})$, en insérant des noeuds seulement entre les noeuds pointés par une variable ou ayant au moins deux prédécesseurs, nous en déduisons que dans FM, il existe $p'' \in P \setminus \{p\}$ tel que $loc(p) \in$ **List**(FM, p'') (on étend la fonction **List** aux formes mémoire en l'appliquant au graphe mémoire sous-jacent). Nous nous trouvons alors soit dans le cas de la ligne 1 soit dans celui de la ligne 3 du tableau de la figure 5.1, la ligne 4 étant exclue par le fait que comme $loc(p) \in$ **List**(FM, p''), nous avons nécessairement $|loc^{-1}(\{loc(p)\})| \neq 1$ (dans le cas où $loc(p'') = loc(p)$) ou $|succ^{-1}(\{loc(p)\})| \neq 1$ (dans le cas contraire).

  - Dans le cas de la ligne 1 comme dans celui de la ligne 3, pour construire FM', nous déplaçons le pointeur p en le faisant pointé vers le noeud $loc(p')$. Si $loc(p)$ a au moins deux prédécesseurs ou est pointé par au moins deux variables en incluant p, nous nous trouvons dans le cas 1. et mise à part le déplacement de variable, cette opération ne modifie pas le graphe ni le nombre de noeud, nous avons alors $GM' = FM'(\mathbf{v}_{|X_{FM}})$ (donc $GM' \sim (FM', \mathbf{v})$) et la paire $(\mathbf{v}, \mathbf{v})$ satisfait bien la formule $\bigwedge_{x \in X} x' = x$.
  - Le cas de la ligne 3 est un peu plus complexe car le fait de déplacer la variable p pour construire FM', fait que le noeud pointé par $loc(p)$ devient un noeud avec un unique prédécesseur qui n'est pas pointé par une variable, il faut donc l'enlever pour obtenir une forme mémoire, toutefois les opérations sur les compteurs garantissent qu'en supprimant ce noeud les longueurs des différentes listes sont préservées. Donc en considérant une valuation $\mathbf{v}'$ telle que $(\mathbf{v}, \mathbf{v}')$ satisfait ϕ, nous avons $GM' = FM'(\mathbf{v}')$.

- Si maintenant $GM' = \texttt{Leak}$. Alors nous sommes dans le cas de la ligne 3 du tableau de la figure 4.5. Comme précédemment, cela signifie que dans FM, nous avons également qu'il n'existe pas de pointeur $p'' \in P \setminus \{p\}$ tel que $loc(p) \in \mathbf{List}(FM, p'')$, par conséquent p est l'unique pointeur vers $loc(p)$ et soit $loc(p)$ n'a pas de prédécesseur, soit il appartient à une liste cyclique dont p est l'unique variable permettant d'y accéder. Ce cas correspond bien à la ligne 4 du tableau de la figure 5.1. Nous ne changeons pas la valeur des compteurs et nous avons bien $\texttt{Leak} \sim (\texttt{Leak}, \mathbf{v})$ (par définition de $\sim$).

Nous considérons l'action $p := p.\texttt{succ}$. Soit $GM' \in \mathcal{GM}^{err}$ tel que $GM' = [\![a]\!]_P(GM)$.

- Supposons que $GM' \notin \{\texttt{Seg}, \texttt{Leak}, \texttt{LeakSeg}\}$. Nous sommes alors dans un des cas présentés dans les deux premières lignes du tableau de la figure 4.7.
  - Si nous sommes dans le cas de la première ligne alors, dans FM, nous avons $succ(loc(p')) \neq loc(p)$. Si le noeud pointé par p dans FM a au moins deux prédécesseurs, ou est étiqueté par une autre variable que p, ou est égal à $\texttt{null}$ ou à $\perp$, nous sommes dans le cas évoqué par la ligne 1 du tableau de la figure 5.3. Deux cas se posent alors :

    1. soit le noeud successeur de p dans FM est aussi le noeud successeur de p dans GM, dans ce cas la valeur du compteur associé à $loc(p)$ vaut 1 et alors il suffit de déplacer p et de le faire pointer vers son successeur,

    2. soit le noeud successeur de p dans FM n'est pas le noeud successeur de p dans GM (c'est-à-dire que l'on a inséré des noeuds au moment de la concrétisation), il faut alors inclure un nouveau noeud pointé par p entre $loc(p)$ et $succ(loc(p))$; la valeur associée au compteur du noeud pointé auparavant par p vaut alors 1, et la valeur du compteur associé au nouveau noeud vaut la valeur du compteur du noeud pointé auparavant par p à laquelle on retire 1.

  - La ligne 2 du tableau de la figure 5.3 correspond à la variante de ce cas, mise à part que cette fois, en déplaçant p, le noeud pointé jusqu'à alors par p devient un noeud à un prédécesseur et qui n'est pas étiqueté par une variable, il faut donc le supprimer (ou le réutiliser comme nous le faisons lorsque nous insérons un noeud entre $loc(p)$ et $succ(loc(p))$). Finalement si p pointe dans un cycle dans FM qui n'est accessible par aucune autre variable, alors nous avons $GM' = GM = FM(\mathbf{v}_{X_{FM}})$, ce cas correspond au cas de la ligne 1 du tableau de la figure 5.4.

- Si maintenant nous avons $GM' = \texttt{Seg}$ cela signifie que $loc(p') \in \{\texttt{null}, \perp\}$ et nous sommes dans le cas de la ligne 3 du tableau 4.7. Nous vérifions alors que les conditions de la ligne 4 du tableau de la figure 5.4 sont respectées. Nous considérons les différentes hypothèses de la ligne 3 du tableau de la figure 4.7. Si il existe un pointeur $p'' \in P \setminus \{p\}$ tel que le noeud pointé par p dans GM appartient

à **List**$(GM, loc(p''))$, alors dans FM nous avons :

- soit $|loc^{-1}((\{loc(p)\}))| > 1$ (il s'agit du cas où p'' pointe vers le même noeud que p dans GM),
- soit $succ^{-1}(\{loc(p)\}) = \{n\}$ avec $n \neq loc(p)$ (il s'agit du cas où le noeud pointé par p a unique prédécesseur et celui-ci est nécessairement accessible par une autre variable que p, en l'occurrence p''),
- soit $|succ^{-1}(\{loc(p)\})| \geq 2$ (il s'agit du cas où $loc(p)$ a deux prédécesseurs et dans ce cas un des deux est nécessairement accessible par une autre variable que p, en l'occurrence p'').

Les conditions de la ligne 4 du tableau sont donc bien respectées, et nous avons $\texttt{Seg} \sim (\texttt{Seg}, \mathbf{v})$ et $(\mathbf{v}, \mathbf{v}) \models \phi$.

– Les cas où $GM \in \{\texttt{Leak}, \texttt{LeakSeg}\}$ se prouvent de manière identique.

Nous considérons l'action $p.\texttt{succ} := p'$.

– Nous supposons que $GM \notin \{\texttt{Seg}, \texttt{Leak}\}$. Nous sommes dans le cas présenté dans la première ligne du tableau de la figure 4.8. Montrons alors que nécessairement FM vérifie les hypothèses d'une des deux premières lignes du tableau de la figure 5.5. Comme $GM \notin \{\texttt{Seg}, \texttt{Leak}\}$, nous avons nécessairement $loc(p) \notin \{\texttt{null}, \bot\}$. De plus le successeur de $loc(p)$ a soit deux prédécesseurs ou est soit étiqueté par une variable (par définition de FM). L'unique cas qui pose problème est le cas où ce successeur n'est pas étiqueté par une variable et s'avère être l'unique noeud d'une liste cyclique, ce cas n'est pas possible car à ce moment il n'existe pas de variable p'' tel que dans GM le successeur de ce noeud soit accessible par p'' sans que p soit accessible par p''. De plus, comme $GM' \neq \texttt{Leak}$, le compteur associé à $loc(p)$ dans FM vaut nécessairement 1 car sinon cela signifierait que dans GM le noeud successeur du noeud pointé par p est accessible uniquement par la variable p est l'action $p.\texttt{succ} := p'$ réalise alors une fuite mémoire. Nous sommes donc soit dans le premier cas de la ligne 1 soit dans dans le premier cas de la ligne 2 du tableau de la figure 5.5. L'unique différence entre ces deux cas vient du fait qu'après avoir changé le successeur de $loc(p)$, le noeud qui était précédemment successeur de $loc(p)$ peut devenir un noeud ayant un unique prédécesseur et qui n'est pas étiqueté par une variable, c'est-à-dire un noeud à supprimer en modifiant la valeur des compteurs correspondants (c'est ce qui arrive dans le premier cas de la ligne 2). Pour chacun de ces deux cas nous avons de plus que dans FM' le successeur du noeud pointé par p est bien le noeud pointé par p', comme c'est le cas dans GM'. Le lemme est donc vérifié pour ces cas là.
– Si $GM' = \texttt{Seg}$, cela signifie que p pointe vers $\texttt{null}$ ou $\bot$ dans GM et par conséquent dans FM et nous sommes alors dans le cas de la ligne 3 du tableau de la figure 5.5, et là encore le lemme est vérifié. Il en va de même lorsque $GM' = \texttt{Leak}$, qui est facilement déduit en reprenant les explications ci-dessus.

Nous ne détaillons pas la preuve pour les autres actions, car la méthode utilisée est toujours la même et de plus les cas présentés ci-dessus peuvent être facilement adaptés. Par exemple, dans le cas de l'action $p := \texttt{malloc}$, la procédure est tout à fait similaire à $p := p'$ car il s'agit là aussi de déplacer la variable p est de s'assurer que les cas où une fuite mémoire a lieu correspondent bien. □

Si $g \in \mathcal{G}$ est une garde sur les pointeurs de P, et si FM est une forme mémoire telle que $FM = (N, P, X, succ, loc, c)$, nous dirons que $FM \models g$ si et seulement si cette propriété est vraie pour le graphe mémoire sous-jacent de FM, c'est-à-dire si et seulement si nous avons $(N, P, succ, loc) \models g$. Nous avons alors la propriété suivante :

Lemme 5.4 *Soient $(FM, \mathbf{v})$ une forme mémoire valuée et $g \in \mathcal{G}$. Nous avons $FM \models g$ si et seulement si $FM(\mathbf{v}) \models g$.*

Preuve : Le résultat de ce lemme est dû au fait que lorsque l'on crée le graphe mémoire $FM(\mathbf{v})$, on ne déplace pas les variables pointant sur `null` et deux variables pointant vers le même noeud dans la forme mémoire FM pointent aussi vers le même noeud dans le graphe mémoire $FM(\mathbf{v})$. Comme de plus les gardes sur les pointeurs permettent seulement de tester si une variable pointe vers `null` ou si deux variables pointent vers le même noeud, nous obtenons facilement la propriété énoncée. □

La fonction **POST** et la définition de la relation de satisfiabilité $\models$ étendue aux formes mémoire nous permettent de construire un système à compteurs à partir d'un système à pointeurs. Soit $S = \langle Q, P, E \rangle$ un système à pointeurs. Nous lui associons le système à compteurs $S_C = \langle Q_C, X, E_C \rangle$ avec :

- $Q_C = Q \times \mathcal{FM}^{err}_{P,X}$,
- $E_C \subseteq (Q \times \mathcal{FM}) \times \mathbf{Presb}(X, X') \times Q_C$ est la relation de transitions qui vérifie la condition suivante :

$$((q, FM), \phi, (q', FM')) \in E_C$$
si et seulement si
il existe $(q, (g, a), q') \in E$ tel que $FM \models g$ et $(\phi, FM') \in \mathbf{POST}(a, FM)$

Remarquons que comme, d'après la proposition 4.12, le nombre de formes mémoire dans $\mathcal{FM}_{P,X}$ est fini, S_C a bien un nombre fini d'états de contrôle. Il est de plus possible de construire de façon effective S_C en utilisant la définition de la fonction **POST** donnée auparavant. Nous allons voir dans la suite que le système à compteurs S_C est bisimilaire au système à compteurs S et qu'il est ainsi possible d'utiliser S_C pour vérifier des propriétés sur S.

5.1.2 Bisimulation et vérification

Soient $S = \langle Q, P, E \rangle$ un système à pointeurs et $S_C = \langle Q_C, X, E_C \rangle$ le système à compteurs construit à partir de S. Nous étudions dans cette section les différentes propriétés du système à compteurs S_C ainsi que le lien qu'il existe entre les systèmes de transitions $TS(S)$ et $TS(S_C)$.

Tout d'abord, de part la définition de la fonction **POST** que nous utilisons pour définir les formules étiquetant les actions de S_C, nous avons :

Proposition 5.5 *Le système à compteurs S_C est un système à compteurs linéaire à monoïde fini.*

Preuve : D'après la définition 1.15, pour que S_C soit un système à compteurs linéaire, il faut que pour tout $((q, FM), \phi, (q', FM')) \in E_C$, la formule ϕ corresponde à une fonction Presburger-linéaire. Or si $((q, FM), \phi, (q', FM')) \in E_C$, cela signifie qu'il existe une action de pointeurs a tel que $(\phi, FM') \in \mathbf{POST}(a, FM)$, et en regardant la définition de la fonction **POST** fournie par les tableaux des figures 5.1 à 5.7, nous constatons que ϕ définit bien une fonction Presburger-linéaire.
De plus S_C est à monoïde fini car pour toute fonction Presburger-linéaire $(\psi, A, \mathbf{b})$ tel qu'il existe $((q, FM), (\psi, A, b), (q', FM')) \in E_C$, nous remarquons que la matrice A est une matrice telle que chaque colonne de A ne contient que des 0 excepté un élément qui est égal à 1, or le monoïde multiplicatif engendré par un tel ensemble de matrices est nécessairement fini. En effet, le produit de deux matrices qui contiennent dans chaque colonne uniquement des 0 sauf à une ligne où l'on trouve

un 1 est encore une matrice vérifiant la même propriété, et comme il n'y a qu'un nombre borné de telles matrices dans $\mathcal{M}_n(\mathbb{N})$ pour un n fixé, nous en déduisons que le monoïde engendré par un ensemble de matrices vérifiant cette propriété est fini. Dans la plupart des cas, la matrice de la fonction Presburger-linéaire étiquetant une transition de E_C est la matrice identité, qui vérifie la propriété énoncée, cependant lorsque nous réalisons une action $y' = y + z$, nous prenons soin de réaliser également l'action $z' = 1$ ou $z' = 0$ de façon à ce que les matrices utilisées vérifient toujours cette propriété. □

Le fait que le système à compteurs soit linéaire à monoïde fini est un point important car cela va nous permettre d'utiliser les méthodes développées pour analyser cette classe de systèmes à compteurs, en particulier les résultats des théorèmes 1.47 et 1.55 énoncés au chapitre 1. Nous pourrons de plus nous servir de l'outil FAST pour analyser ces systèmes. Mais avant nous explicitons le lien qui existe entre les systèmes de transitions $TS(S) = \langle Q \times \mathcal{GM}^{err}, E, \rightarrow \rangle$ et $TS(S_C) = \langle (Q \times \mathcal{FM}^{err}) \times \mathbb{N}^X, E_C, \rightarrow_C \rangle$ associés respectivement à S et S_C. Pour ce faire, nous étendons la relation $\sim$, définie dans la section précédente, aux configurations de $TS(S)$ et de $TS(S_C)$ de telle sorte que $\sim \subseteq \big(Q \times \mathcal{GM}^{err}\big) \times \big((Q \times \mathcal{FM}^{err}) \times \mathbb{N}^X\big)$ et que $(q, GM) \sim ((q', FM'), \mathbf{v})$ si et seulement si les conditions suivantes sont vérifiées :

– $q = q'$, et,
– $GM \sim (FM, \mathbf{v})$.

Nous avons alors le lemme suivant, montrant que les systèmes de transitions $TS(S)$ et $TS(S_C)$ sont bisimilaires.

Lemme 5.6 *Soient $(q_1, GM_1) \in Q \times \mathcal{GM}^{err}$ et $((q_1, FM_1), \mathbf{v}_1) \in Q_C \times \mathbb{N}^X$ tels que $(q_1, GM_1) \sim ((q_1, FM_1), \mathbf{v}_1)$. Alors pour tout $(q_2, GM_2) \in Q \times \mathcal{GM}^{err}$, $(q_1, GM_1) \rightarrow (q_2, GM_2)$ si et seulement si il existe $((q_2, FM_2), \mathbf{v}_2) \in Q_C \times \mathbb{N}^X$ tel que :*

– $((q_1, FM_1), \mathbf{v}_1) \rightarrow_C ((q_2, FM_2), \mathbf{v}_2)$*, et,*
– $(q_2, GM_2) \sim ((q_2, FM_2), \mathbf{v}_2)$.

Preuve : Supposons que $(q_1, GM_1) \xrightarrow{e} (q_2, GM_2)$ avec $e = (q_1, (g, a), q_2)$. Comme $GM_2 = [\![a]\!]_P(GM_1)$, en utilisant le résultat du lemme 5.3, nous en déduisons qu'il existe $(FM_2, \mathbf{v}_2) \in \mathcal{FM}^{err} \times \mathbb{N}^X$ et $\phi \in \textbf{Presb}(X, X')$ tels que $GM_2 \sim (FM_2, \mathbf{v}_2)$ et $(\phi, FM_2) \in \textbf{POST}(a, FM_1)$ et $(\mathbf{v}_1, \mathbf{v}_2) \models \phi$. De plus comme $GM_1 \models g$, et comme $GM_1 \sim (FM_1, \mathbf{v}_1)$, nous avons d'après le lemme 5.4 et la définition de $\sim$, $FM_1 \models g$. Par conséquent, par définition de E_C, la transition $e' = ((q_1, FM_1), \phi, (q_2, FM_2)))$ appartient à E_C. Comme $(\mathbf{v}_1, \mathbf{v}_2) \models \phi$, nous avons bien $((q_1, FM_1), \mathbf{v}_1) \rightarrow_C ((q_2, FM_2), \mathbf{v}_2)$ et comme de plus $GM_2 \sim (FM_2, \mathbf{v}_2)$, nous avons également $(q_2, GM_2) \sim ((q_2, FM_2), \mathbf{v}_2)$. La réciproque se prouve de la même façon. □

Ce lemme nous permet alors d'établir une propriété sur les ensembles d'accessibilité de S et S_C, à savoir :

Lemme 5.7 *Soient $c_0 \in Q \times \mathcal{GM}^{err}$ une configuration initiale de $TS(S)$ et $c'_0 \in Q_C \times \mathbb{N}^X$ tel que $c_0 \sim c'_0$. Nous avons* $\textbf{Reach}(S, c_0) = \{c \in Q \times \mathcal{GM}^{err} \mid \exists c' \in \textbf{Reach}(S_C, c'_0)$ *tel que* $c \sim c'\}$.

Preuve : Soit $c \in \textbf{Reach}(S, c_0)$. Montrons qu'il existe $c' \in \textbf{Reach}(S_C, c'_0)$ tel que $c \sim c'$. Comme $c \in \textbf{Reach}(S, c_0)$, il existe une exécution dans $TS(S)$ de la forme $c_0 \rightarrow c_1 \dots c_f$ avec $f \in \mathbb{N}$ et $c_f = c$. Nous raisonnons par induction sur la taille de cette exécution, c'est-à-dire sur f, en utilisant le lemme 5.6. Si $f = 0$, alors nous avons bien $c_0 \sim c'_0$ (par hypothèse) et $c'_0 \in \textbf{Reach}(S_C, c'_0)$.

Supposons que $f > 0$ et que pour tout $j \in [1..f-1]$, il existe $c'_j \in \mathbf{Reach}(S_C, c'_0)$ tel que $c_j \sim c'_j$. En particulier, nous avons que $c_{f-1} \sim c'_{f-1}$ et comme $c_{f-1} \rightarrow c_f$, d'après le lemme 5.6, il existe une configuration c' de $TS(S_C)$ telle que $c'_{f-1} \rightarrow_C c'$ et $c_f \sim c'$. Comme $c'_{f-1} \in \mathbf{Reach}(S_C, c'_0)$, nous avons bien que $c' \in \mathbf{Reach}(S_C, c'_0)$.
La réciproque se prouve de la même façon. □

Le lemme précédent concerne le cas où nous avons une unique configuration initiale, mais il s'avère qu'il est également possible de considérer un ensemble infini de graphes mémoire initiaux décrits par un état mémoire symbolique. Tout d'abord remarquons que si q_0 est un état de contrôle dans Q et (FM_0, ϕ_0) est une forme mémoire symbolique, alors $((q_0, FM_0), \phi_0)$ est une configuration symbolique de $TS(S_C)$ et de plus pour tout $GM \in \mathcal{GM}$, nous avons $GM \in [\![(FM_0, \phi_0)]\!]$ si et seulement si il existe $\mathbf{v} \in \mathbb{N}^X$ tel que $GM \sim (FM_0, \mathbf{v})$ et $\mathbf{v} \models \phi_0$. En effet, comme $\phi_0 \in \mathbf{Presb}(X_{FM})$, nous avons $\mathbf{v} \models \phi_0$ si et seulement si $\mathbf{v}_{|X_{FM_0}} \models \phi_0$ et nous pouvons alors facilement conclure en utilisant la définition de $[\![(FM_0, \phi_0)]\!]$ et celle de $\sim$. Par application immédiate du lemme précédent, nous obtenons :

$$\mathbf{Reach}(S, (q_0, \{(FM_0, \phi_0)\})) = \{c \in Q \times \mathcal{GM}^{err} \mid \exists c' \in \mathbf{Reach}(S, ((q_0, FM_0), \phi_0)) \text{ tel que } c \sim c'\}$$

Ainsi pour vérifier si un système à pointeurs muni d'une configuration symbolique initiale (q_0, EMS_0) avec $EMS_0 = \{(FM_0, \phi_0)\}$ provoque une erreur de segmentation, il faut chercher un $q \in Q$ et $\mathbf{v} \in \mathbb{N}^X$ tel que la configuration $((q, \mathtt{Seg}), \mathbf{v})$ appartient à l'ensemble $\mathbf{Reach}(S, ((q_0, FM_0), \phi_0))$, c'est-à-dire que ce problème se réduit à savoir si un ensemble d'états de contrôle est accessible dans la système à compteurs S_C muni de la configuration symbolique initiale $((q_0, FM_0), \phi_0)$.

De plus, lorsque le système à compteurs correspondant a un ensemble d'accessibilité effectivement définissable dans Presburger, alors l'ensemble d'accessibilité du système à pointeurs considéré peut être également représenté de façon symbolique, comme cela est présenté par le lemme suivant :

Lemme 5.8 *Soit $q_0 \in Q$ un état de contrôle de S et $EMS_0 = \{(FM_0, \phi_0)\}$ un état mémoire symbolique (composé d'une unique forme mémoire symbolique). Alors si $\mathbf{Reach}(S_C, ((q_0, FM_0), \phi_0))$ est effectivement définissable dans Presburger, pour chaque état de contrôle $q \in Q$, il est possible de construire un état mémoire symbolique EMS_q tel que :*

$$\mathbf{Reach}(S, EMS_0) \cap (Q \times \mathcal{GM}) = \bigcup_{q \in Q} \{q\} \times [\![EMS_q]\!]$$

Preuve : Supposons que $\mathbf{Reach}(S_C, ((q_0, FM_0), \phi_0))$ soit effectivement définissable dans Presburger. Alors pour chaque paire $(q, FM) \in Q_C$, il existe une formule de Presburger $\phi_{q,FM} \in \mathbf{Presb}(X)$ tel que $\mathbf{Reach}(S_C, ((q_0, FM_0), \phi_0)) = \bigcup_{(q,FM) \in Q_C} \{(q, FM)\} \times [\![\phi_{q,FM}]\!]_X$. Pour chaque paire $(q, FM) \in Q \times \mathcal{GM}$, nous construisons la formule $\phi'_{q,FM} = \exists x_1 \ldots \exists x_m . \phi_{q,FM}$ avec $X \setminus X_{FM} = \{x_1, \ldots, x_m\}$. La paire $(FM, \phi'_{q,FM})$ est alors une forme mémoire symbolique. Nous définissons ensuite pour chaque état de contrôle $q \in Q$, l'état mémoire symbolique $EMS_q = \bigsqcup_{FM' \in \mathcal{FM}} \{(FM', \phi'_{q,FM'})\}$ et nous montrons maintenant que $\mathbf{Reach}(S, EMS_0) \cap (Q \times \mathcal{GM}) = \bigcup_{q \in Q} \{q\} \times [\![EMS_q]\!]$.
Soit $(q, GM) \in \mathbf{Reach}(S, EMS_0) \cap (Q \times \mathcal{GM})$. En utilisant le lemme 5.7, nous en déduisons qu'il existe $((q, FM), \mathbf{v}) \in \mathbf{Reach}(S_C, ((q_0, FM_0), \phi_0))$ tel que $GM \sim (FM, \mathbf{v})$. De plus, par définition de $\phi_{q,FM}$, nous avons $\mathbf{v} \models \phi_{q,FM}$ d'où nous déduisons que $\mathbf{v}_{|X_{FM}} \models \phi'_{q,FM}$. Comme

$GM = FM(\mathbf{v}_{|X_{FM}})$, nous avons $GM \in [\![(FM, \phi'_{q,FM})]\!]$ et du coup $GM \in [\![EMS_q]\!]$, ceci car d'après la proposition 4.20, $[\![\bigsqcup_{FM' \in \mathcal{FM}} \{(FM', \phi'_{q,FM'})\}]\!] = \bigcup_{FM' \in \mathcal{FM}} [\![(FM', \phi'_{q,FM'})]\!]$. Soit $(q, GM) \in \bigcup_{q \in Q} \{q\} \times [\![EMS_q]\!]$. Alors $GM \in [\![\bigsqcup_{FM \in \mathcal{FM}} \{(FM, \phi'_{q,FM})\}]\!]$, par conséquent, il existe $(FM, \phi'_{q,FM})$ tel que $GM \in [\![(FM, \phi'_{q,FM}]\!]$. Il existe donc une valuation $\mathbf{v} \in \mathbb{N}^X$ telle que $GM = FM(\mathbf{v}_{|X_{FM}})$ et telle que $\mathbf{v} \models \phi_{q,FM}$ (par définition de $\phi'_{q,FM}$). Nous avons donc que $((q, FM), \mathbf{v})) \in \mathbf{Reach}(S_C, ((q_0, FM_0), \phi_0))$, en utilisant le lemme 5.7, on en déduit que $GM \in \mathbf{Reach}(S, EMS_0)$. □

Pour énoncer les théorèmes qui suivent nous appellerons S_C le système à compteurs associé à S. Si S est muni d'une configuration symbolique initiale (q_0, EMS_0) telle que $EMS_0 = \{(FMS_0, \phi_0)\}$ est un singleton, nous dirons que S est muni d'une configuration symbolique initiale simple ; comme nous venons de le voir, nous associons à un tel système à pointeurs, le système à compteurs S_C muni de la configuration symbolique initiale $((q_0, FM_0), \phi_0)$. Nous avons alors les deux théorèmes suivants :

Théorème 5.9 *Le problème d'accessibilité symbolique généralisé est décidable pour les systèmes à pointeurs dont le système à compteurs associé est plat.*

Preuve : Soient (q_0, EMS_0) et (q, EMS) deux configurations symboliques dans $Q \times \mathcal{EMS}$. Nous supposons que $EMS_0 = \{(FM_1, \phi_1), \ldots, \ldots, (FM_m, \phi_m)\}$. Comme S_C est un système à compteurs linéaire plat et à monoïde fini, en utilisant le résultat du théorème 1.47, pour chaque $i \in [1..m]$, nous savons que l'ensemble $\mathbf{Reach}(S_C, ((q, FM_i), \phi_i))$ est effectivement définissable dans Presburger. En utilisant le résultat du lemme 5.8, on en déduit que pour chaque $i \in [1..m]$, nous avons $\mathbf{Reach}(S, (q, (FM_i, \phi_i))) = \bigcup_{q \in Q} \{q\} \times [\![EMS_{q,i}]\!]$. Ainsi le problème d'accessibilité symbolique généralisé revient à tester si il existe $i \in [1..m]$ tel que $[\![EMS]\!] \cap [\![EMS_{q,i}]\!] = \emptyset$. Comme $[\![EMS]\!] \cap [\![EMS_{q,i}]\!] = [\![EMS \sqcap EMS_{q,i}]\!]$ et que, d'après le corollaire 4.22, le problème du vide est décidable pour les états mémoire symboliques, nous obtenons le résultat souhaité. □

Théorème 5.10 *Le problème d'accessibilité symbolique généralisé est décidable pour les systèmes à pointeurs muni d'une configuration symbolique initiale simple dont les système à compteurs associés sont symboliquement aplatissables.*

Preuve : La preuve de ce théorème est identique à celle du théorème précédent excepté le fait que l'on utilise ici le résultat de la proposition 1.54 qui nous dit que si un système à pointeurs muni d'une configuration symbolique initiale est symboliquement aplatissable, alors l'ensemble d'accessibilité correspondant est effectivement définissable dans Presburger. □

Cependant comme nous l'avons vu avec le corollaire 1.56, savoir si un système à compteurs muni d'une configuration symbolique initiale est aplatissable est un problème indécidable. Toutefois nous savons que si l'algorithme implanté dans l'outil FAST termine alors le système à compteurs muni d'une configuration symbolique initiale donné en entrée est aplatissable et de plus l'ensemble retourné par cet algorithme correspond à l'ensemble d'accessibilité du système à compteurs considéré. Soit $(q_0, \{EMS_0, \phi_0\})$ une configuration symbolique initiale du système à pointeurs S. La méthode que nous proposons pour analyser ce système à pointeurs se déroule donc en deux phases :

1. Construction du système à compteurs S_C muni de la configuration initiale $((q_0, FM_0).\phi_0)$,
2. Analyse du système à compteurs construit (avec l'outil FAST par exemple).

Remarquons qu'en pratique lors de la phase de construction du système à compteurs S_C nous ne construisons que la composante connexe du graphe sous-jacent contenant l'état de contrôle (q_0, FM_0) et ainsi cela nous donne déjà une information sur le système à pointeurs. En effet, si il n'existe pas d'état de contrôle $q \in Q$ tel que $(q, \texttt{Seg})$ est un état de contrôle appartenant à la composante connexe construite, alors nous savons que le système à pointeurs S muni de la configuration initiale symbolique considérée ne provoque pas d'erreur de segmentation.

Exemple 5.11 *La figure 5.10 donne un exemple de système à compteurs construit à partir du système à pointeurs de la figure 4.3 correspondant au programme* `deleteAll`*. Comme nous l'avons précisé précédemment nous avons considéré dans cette exemple une configuration symbolique initiale contenant la forme mémoire FM_0 représentée dans l'état de contrôle situé en haut à gauche de la figure, et nous avons seulement construit la composante connexe liée à cet état de contrôle. De plus, nous n'avons pas représenté dans les différentes formes mémoire, le pointeur q lorsque celui ci pointait vers $\perp$. Comme il n'y a pas d'état de contrôle connecté à $(1, FM_0)$ contenant l'élément* `Seg` *ou l'élément* `LeakSeg`*, nous en déduisons que le système à pointeurs muni de la configuration symbolique initiale utilisant $(FM_0, \bigwedge_{x \in X}$ ne provoque pas d'erreur de segmentation et ceci sans avoir besoin de calculer l'ensemble d'accessibilité du système à compteurs construit. Nous signalons que dans ce système à compteurs, nous avons regroupé ensemble les états de contrôle dont les formes mémoire étaient isomorphes, ceci pour limiter le nombre d'état de contrôle.*

5.1.3 Des systèmes à pointeurs plats donnent un système à compteurs non plat

Nous avons vu précédemment, avec le théorème 5.9 que lorsque le système à compteurs associé à un système à pointeurs est plat alors il est possible de résoudre le problème d'accessibilité symbolique généralisé. Aussi, si les systèmes à pointeurs plats produisaient un système à compteurs plats, nous aurions une classe intéressante de systèmes à pointeurs pour laquelle ce dernier problème est décidable. Nous adaptons alors la définition 1.44 de systèmes à compteurs plats aux systèmes à pointeurs.

Definition 5.12 (Système à pointeurs plat) *Un système à pointeurs $S = \langle Q, P, E \rangle$ est plat, si pour tout état de contrôle $q \in Q$, q apparaît dans au plus un cycle élémentaire.*

Remarquons que si dans un programme, il n'y a pas de boucles **while** imbriquées et il n'y a pas de conditions **if** ... **then** ... **else** ... dans le corps des boucles **while**, alors le système à pointeurs associé est plat. Nous considérons ainsi le système à pointeurs S_1 représentée à la figure 5.11 ainsi que la configuration symbolique initiale $(1, (FM_0, \bigwedge_{x \in X} x > 0))$ (FM_0 étant également représenté sur la figure 5.11). Remarquons que S est un système à pointeurs plat.
La figure 5.12 nous donne une représentation de la composante connexe du système à compteurs S_{1_C} contenant l'état de contrôle $(1, FM_0)$. Nous remarquons que cette composante connexe n'est pas plate, d'où nous pouvons conclure que le système à compteurs associé à un système à pointeurs plat n'est pas nécessairement plat.

Proposition 5.13 *Il existe des systèmes à pointeurs plats S tels que le système à compteurs associé S_C n'est pas plat.*

Toutefois, en analysant la fonction **POST** définie à la section précédente, on s'aperçoit que pour toute action $a \in A$, pour toute forme mémoire $FM \in \mathcal{FM}$, nous avons $|\mathbf{POST}(a, FM)| \leq 2$ et de plus il existe $\phi_1, \phi_2 \in \mathbf{Presb}(X, X')$ et $FM_1, FM_2 \in \mathcal{FM}$ tels que $\mathbf{POST}(a, FM) = \{(\phi_1, FM_1), (\phi_2, FM_2)\}$ et $FM_1 \neq FM_2$ si et seulement si a est une action de la forme $p := p'.\texttt{succ}$

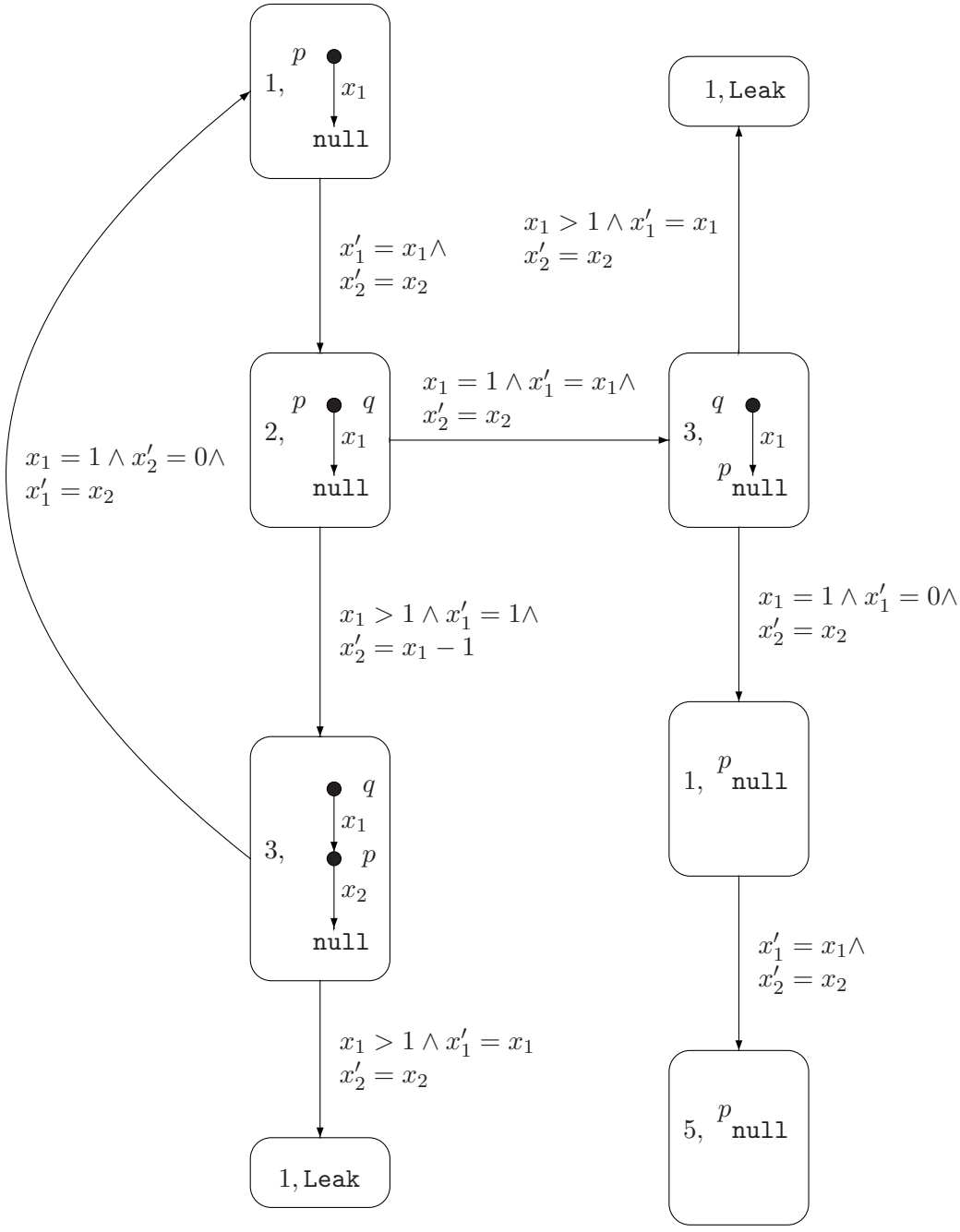


FIGURE 5.10 – Exemple de système à compteurs obtenu à partir du programme `deleteAll`

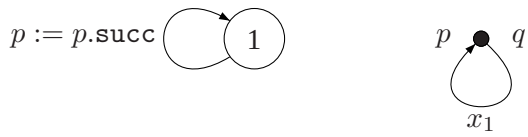


FIGURE 5.11 – Un système à pointeurs plat S_1 et une forme mémoire symbolique FM_0

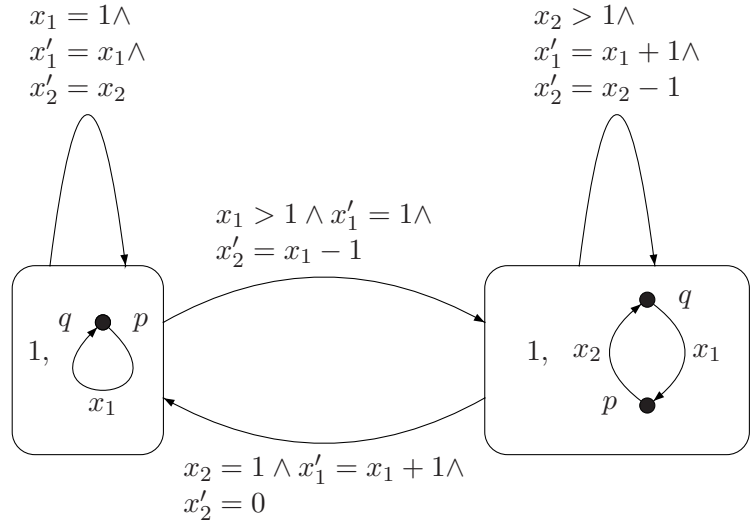


FIGURE 5.12 – Exemple de système à compteurs obtenu à partir du système à pointeurs S_1

avec $p, p' \in P$. En effet, il est possible que pour une autre action, l'ensemble obtenu par application de la fonction **POST** contienne deux éléments différents, mais à ce moment un des deux éléments est de la forme (FM, ϕ) avec FM dans $\{\texttt{Seg}, \texttt{Leak}, \texttt{LeakSeg}\}$. Nous appelons déplacement au successeur les actions de la forme $p := p'.\texttt{succ}$ avec $p, p' \in P$. Nous dirons alors qu'un système à pointeurs $S = \langle Q, P, E\rangle$ est sans déplacement au successeur si et seulement si pour toute transition $(q, (g, a), q') \in E$, a n'est pas un déplacement au successeur. Nous avons la proposition suivante :

Proposition 5.14 *Si S est un système à pointeurs plat sans déplacement au successeur, alors S_C est un système à compteurs plat.*

Preuve : Soient $S = \langle Q, P, E\rangle$ un système à pointeurs plat sans déplacement au successeur et $S_C = \langle Q_C, X, E_C\rangle$ le système à compteurs associé. Nous raisonnons par l'absurde et supposons que S_C n'est pas plat. Alors il existe un état de contrôle $(q, FM) \in Q_C$ et deux transitions $e_1 = ((q, FM), \phi_1, (q_1, FM_1))$ et $e_2 = ((q, FM), \phi_2, (q_2, FM_2))$ dans E_C telle que $e_1 \neq e_2$ et e_1 et e_2 appartiennent chacune à un cycle élémentaire. Supposons que $q_1 \neq q_2$. Cela n'est pas possible sinon par construction de S_C cela signifierait qu'il existe deux transitions $(q, (g, a), q_1)$ et $(q, (g, a), q_2)$ dans E qui appartiennent chacune à un cycle élémentaire, et par conséquent S ne serait pas plat. Nous avons donc nécessairement $q_1 = q_2$. Par définition de S_C, il existe une transition $(q, (g, a), q_1)$ dans E et comme S est plat cette transition est unique. Nous avons alors nécessairement $(\phi_1, FM_1) \in$ **POST**(a, FM) et $(\phi_2, FM_2) \in$ **POST**(a, FM) et $(\phi_1, FM_1) \neq (\phi_2, FM_2)$. Comme e_1 et e_2 appartiennent chacune à un cycle élémentaire et qu'il n'y a pas de transition dans E_C partant d'un état de contrôle contenant un élément de $\{\texttt{Seg}, \texttt{Leak}, \texttt{LeakSeg}\}$, nous avons nécessairement $\{FM_1, FM_2\} \cap \{\texttt{Seg}, \texttt{Leak}, \texttt{LeakSeg}\} = \emptyset$. Or comme S est sans déplacement au successeur, cette situation est impossible, nous en déduisons que S_C est plat. □

Nous déduisons alors de façon immédiate, en appliquant le théorème 5.9, le résultat suivant :

Théorème 5.15 *Le problème d'accessibilité symbolique généralisé est décidable pour les systèmes à pointeurs plats sans déplacement au successeur.*

Nous avons ainsi, de par la façon dont nous construisons S_C, obtenu une sous classe de systèmes à pointeurs plats pour laquelle le problème d'accessibilité symbolique généralisé est décidable. Remarquons, que le fait qu'en général un système à pointeurs plat ne produisent pas un système à compteurs plat avec notre traduction, n'implique pas que le problème d'accessibilité symbolique généralisé soit indécidable pour les systèmes à pointeurs plats. Cependant il l'est comme nous le verrons dans le chapitre suivant, dans lequel nous étudierons entre autre des questions de décidabilité pour différents problèmes d'accessibilité en considérant des systèmes à pointeurs plats.

Conclusion

Nous avons présenté dans ce chapitre une méthode générale pour analyser les programmes manipulant des listes simplement chaînées basée sur une traduction vers un système à compteurs. Cette méthode nous permet en particulier de réutiliser les outils déjà existants pour l'analyse de systèmes à compteurs comme par exemple l'outil FAST . La méthode proposée permet de vérifier des propriétés sur la forme des structures de données mais aussi des propriétés quantitatives sur le nombre de cellules présentes dans les listes. Notons de plus que de par la bisimulation qui existe entre le système à pointeurs et le système à compteurs, cette méthode peut aussi être utilisée pour vérifier si les exécutions

du système à pointeurs terminent en vérifiant si les exécutions du système à compteurs terminent. Ceci est un avantage car il est plus facile de vérifier cette propriété sur un système à compteurs, où l'on peut vérifier par exemple que les valeurs d'un compteur décroissent strictement, alors que sur un système à pointeurs des conditions de terminaison sont plus difficiles à trouver. Finalement, signalons que récemment d'autres méthodes manipulant des variables représentant la longueur des listes ont été développées. Ainsi, dans [MBCC07], les auteurs proposent une méthode combinant un calcul utilisant la logique de séparation et une analyse arithmétique sur la longueur des listes. Les formules arithmétiques leur servent alors à raffiner un calcul surapproximatif des différentes configurations possibles. Une autre méthode a été développée dans [PRW08] pour prouver la terminaison de programmes manipulant des listes simplement chaînées en construisant des préconditions manipulant la longueur des listes. Il apparaît ainsi que dans le cadre de l'analyse de programmes manipulant des pointeurs, les techniques combinant la shape analysis et une analyse numérique permettent de produire un résultat plus précis que les techniques basées uniquement sur la shape analysis.

Chapitre 6

Model-checking de systèmes à pointeurs

Après avoir proposé différents problèmes d'accessibilité pour les systèmes à pointeurs et une méthode de vérification pour ces problèmes basée sur une traduction vers un système à compteurs bisimilaire, nous nous attaquons dans ce chapitre au model-checking. Dans un premier temps, nous proposons une extension de la logique arborescente CTL* , obtenue en ajoutant des états mémoire symboliques à l'ensemble des propositions atomiques. Nous étudions ensuite la décidabilité de ces problèmes d'accessibilité et de model-checking pour des systèmes à pointeurs plats. Les travaux que nous présentons ici sont issus de [FLS07].

6.1 Une logique temporelle pour la vérification de systèmes à pointeurs

6.1.1 La logique CTL^*_{mem}

Nous avons introduit au chapitre 3 la logique temporelle FOCTL*(Pr) qui permet de décrire des propriétés temporelles d'un système à compteurs. Nous avons aussi vu que l'avantage de cette logique temporelle réside dans le fait qu'elle permet de décrire non seulement l'évolution des différents états de contrôle mais aussi des différentes valeurs possibles des compteurs, ceci en utilisant des formules de Presburger comme propositions atomiques. De la même façon, nous introduisons ici la logique temporelle CTL^*_{mem} qui va nous servir à spécifier des propriétés temporelles d'un système à pointeurs. Dans le chapitre précédent, nous avons défini les états mémoire symboliques pour représenter des ensembles infinis de graphes mémoire, tout comme les formules de l'arithmétique de Presburger permettent de représenter des ensembles infinis de vecteurs. Nous étendons ici la logique CTL* en utilisant des états mémoire symboliques comme propositions atomiques. Remarquons déjà que comme CTL^*_{mem} étend CTL* , en posant des restrictions sur les quantificateurs de chemins et sur les opérateurs temporels, nous pourrions définir une extension similaire pour la logique temporelle linéaire LTL et pour la logique temporelle branchante CTL .

Nous donnons la syntaxe de CTL^*_{mem}. Soient Q un ensemble fini d'éléments (pratiquement l'ensemble des états de contrôle du système à pointeurs considéré), P un ensemble fini de pointeurs et X un ensemble fini de compteurs. Nous supposons de plus que $2|P| \leq |X|$ de façon à ce que l'ensemble des états mémoire symboliques $\mathcal{EMS}_{P,X}$ soit clos par complément (cf proposition 4.15). Les formules de la logique $\mathrm{CTL}^*_{mem}[Q, P, X]$ sont définies par :

$$\Phi \ ::= q \mid EMS \mid \Phi \wedge \Phi \mid \mathtt{X}\Phi \mid \Phi \mathtt{U} \Phi \mid \mathtt{A}\Phi$$

où $q \in Q$ et EMS est un état mémoire symbolique dans $\mathcal{EMS}_{P,X}$. Nous utilisons les raccourcis habituels suivants pour $\Phi \in \text{CTL}^*_{mem}[Q, P, X]$:

– $\texttt{F}\Phi$ équivaut à $true\texttt{U}\Phi$,
– $\texttt{G}\Phi$ équivaut à $\neg\texttt{F}\neg\Phi$,
– $\texttt{E}\Phi$ équivaut à $\neg\texttt{A}\neg\Phi$.

Notation 6.1 *Lorsque les ensemble Q, P et X seront explicites, nous ne les préciserons pas dans* $\text{CTL}^*_{mem}[Q, P, X]$ *en écrivant seulement* CTL^*_{mem}.

Tout comme pour la logique FOCTL*(Pr) , nous ne nous intéressons pas ici à des questions de satisfiabilité pour la logique CTL^*_{mem}, et nous nous servons de cette logique uniquement pour résoudre des problèmes de model-checking dans le cadre de la vérification de systèmes à pointeurs. Ceci dit, il pourrait être intéressant d'étudier le problème de satisfiabilité suivant : étant donnée une spécification écrite avec une formule de la logique CTL^*_{mem}, existe-t-il un système à pointeurs dont les exécutions satisfont cette formule ?

6.1.2 Sémantique de CTL^*_{mem} et problèmes de model checking

Nous donnons dans cette section la sémantique de la logique CTL^*_{mem} introduite auparavant. Soit $S = \langle Q, P, E\rangle$ un système à pointeurs et $TS(S) = \langle Q \times \mathcal{GM}^{err}, E, \rightarrow\rangle$ le système de transitions qui lui est associé. Comme pour les systèmes à compteurs, nous utiliserons les notations suivantes sur les exécutions de $TS(S)$. Soient π une exécution finie de $TS(S)$ et $i \in \mathbb{N}$, nous notons :

– $\pi(i) \in Q \times \mathcal{GM}^{err}$ la i-ème configuration de π,
– $\pi_{\leq i}$ la partie initiale de π jusqu'à la position i,
– $|\pi|$ la longueur de π.

Pour $i \in \mathbb{N}$ et une exécution π de $TS(S)$, la relation de satisfaction $\models$ pour les formules de la logique $\text{CTL}^*_{mem}[Q, P, X]$ est définie par induction à la position i de l'exécution π de la façon suivante :

– $\pi, i \models q$ avec $q \in Q$ si et seulement si $\pi(i) = (q, GM)$ avec $GM \in \mathcal{GM}^{err}$,
– $\pi, i \models EMS$ si et seulement si $\pi(i) = (q, GM)$ avec $q \in Q$ et $GM \in [\![EMS]\!]$,
– $\pi, i \models \neg\Phi$ si et seulement si $\pi, i \not\models \Phi$,
– $\pi, i \models \Phi \wedge \Phi'$ si et seulement si $\pi, i \models \Phi$ et $\pi, i \models \Phi'$,
– $\pi, i \models \texttt{X}\Phi$ si et seulement si $i < |\pi| - 1$ et $\pi, i + 1 \models \Phi$,
– $\pi, i \models \Phi\texttt{U}\Phi'$ si et seulement si il existe $j \in \mathbb{N}$ tel que $i \leq j \leq |\pi|$ et $\pi, j \models \Phi'$ et pour tout $k \in \mathbb{N}$ tel que $i \leq k < j$, $\pi, k \models \Phi$,
– $\pi, i \models \texttt{A}\Phi$ si et seulement si pour toute exécution π' de $TS(S)$ telle que $\pi_{\leq i} = \pi'_{\leq i}$, nous avons $\pi', i \models \Phi$.

Nous pouvons alors définir le *problème de model-checking* :

Entrées : Un système à pointeurs $S = \langle Q, P, E\rangle$, une configuration symbolique initiale $(q_0, EMS_0) \in Q \times \mathcal{EMS}_{P,X}$ et une formule Φ de $\text{CTL}^*_{mem}[Q, P, X]$.
Question : Est-il vrai que pour toutes les exécutions π de $TS(S)$ vérifiant $\pi(0) \in \{q_0\} \times [\![EMS_0]\!]$, nous avons $\pi, 0 \models \Phi$?

Tous les problèmes d'accessibilité pour les systèmes à pointeurs que nous avons définis au chapitre 4 sont des cas particuliers du problème de model-checking. En effet, si nous considérons le problème

d'accessibilité symbolique généralisé (défini à la section 4.3.3) avec S comme système à pointeurs, (q_0, EMS_0) comme configuration symbolique initiale et (q, EMS) comme deuxième configuration symbolique, ce problème se réduit à un problème de model-checking avec le même système à pointeurs et la même configuration symbolique initiale et la formule $\Phi = \texttt{EF}(q \wedge EMS)$. Comme d'après la proposition 4.23, les problèmes d'accessibilité généralisés d'une erreur de segmentation et d'une fuite mémoire peuvent se réduire à un nombre fini d'instances du problème d'accessibilité symbolique généralisé, et comme ces deux problèmes sont indécidables pour les systèmes à pointeurs manipulant au moins 3 pointeurs (cf. théorème 4.9), nous en déduisons le théorème suivant :

Théorème 6.2 *Le problème de model-checking est indécidable pour les systèmes à pointeurs manipulant au moins 3 pointeurs.*

Dans la suite de ce chapitre, nous allons étudier la décidabilité de ce problème en nous focalisant principalement sur les systèmes à pointeurs plats et nous verrons que déjà pour cette classe restreinte le problème de model-checking est indécidable.

6.2 Analyse des problèmes d'accessibilité et de model-checking

6.2.1 Un premier résultat de décidabilité

Dans le chapitre précédent, nous avons montré qu'étant donné un système à pointeurs S, il était possible de construire un système à compteurs S_C bisimilaire. Nous avons de plus vu, avec le théorème 5.9, que dans les cas où ce système à compteurs S_C était plat, nous disposions d'un algorithme pour résoudre le problème d'accessibilité symbolique généralisé. Nous montrons maintenant qu'il est possible de traduire une formule de CTL$^*_{mem}$ sur S en une formule de FOCTL*(Pr) sur S_C et d'étendre ainsi le résultat du théorème 5.9 au problème du model-checking.

Soient P un ensemble de pointeurs, X un ensemble de compteurs tels que $2|P| \leq |X|$ et $S = \langle Q, P, E\rangle$ un système à pointeurs. Nous considérons le système à compteurs $S_C = \langle Q_C, X, E_C\rangle$ associé à S dont la construction est décrite dans le chapitre précédent. Nous notons $TS(S) = \langle Q \times \mathcal{GM}, E, \rightarrow\rangle$ et $TS(S_C) = \langle Q_C \times \mathbb{N}^X, E_C, \rightarrow_C\rangle$ les systèmes de transitions associés à S et à S_C. Nous construisons maintenant à partir d'une formule Φ de CTL$^*_{mem}[Q, P, X]$ une formule $T(\Phi)$ de FOCTLX*(Pr)$[X_0]$. Pour rappel $X_0 = \{x_0\} \uplus X$ et la variable x_0 est utilisée pour encoder les états de contrôle. Ainsi nous considérons que l'ensemble des états de contrôle $Q_C = Q \times \mathcal{FM}^{err}$ est isomorphe à l'ensemble d'entiers $[1..|Q \times \mathcal{FM}^{err}|]$ et chaque paire $(q, FM) \in Q \times \mathcal{FM}^{err}$ est donc identifiée à un entier. Avant de donner la construction de la formule $T(\Phi)$, nous introduisons quelques notations supplémentaires. Si il existe un isomorphisme f d'une forme mémoire symbolique (FM_1, ϕ_1) vers (FM_2, ϕ_2) et si $FM_1 = (N_1, P, X, succ_1, loc_1, c_1)$ et $FM_2 = (N_2, P, X, succ_2, loc_2, c_2)$, nous notons ϕ_1^f la formule obtenue en remplaçant dans ϕ_1 chaque instance de compteur x de X_{FM_1} avec $x = c_1(n)$ par le compteur $c_2(f(n))$ de X_{FM_2}. De plus, de façon à simplifier les notations, si FM_1 et FM_2 sont deux formes mémoire, nous noterons $FM_1 \approx_f FM_2$ le fait que FM_1 et FM_2 sont isomorphes et que f est un isomorphisme de FM_1 vers FM_2. Nous construisons alors la formule $T(\Phi)$ par induction de la façon suivante :

– $T(q) = \bigvee_{FM \in \mathcal{FM}} x_0 = (q, FM)$,
– si $\Phi = EMS$ avec $EMS = \{(FM_1, \phi_1), \ldots, (FM_t, \phi_t)\}$ alors :

$$T(\Phi) = \bigvee_{q \in Q} \bigvee_{i \in [1..t]} \bigvee_{FM_i \approx_f FM} x_0 = (q, FM) \wedge \phi_i^f$$

- $T(\neg\Phi) = \neg T(\Phi)$,
- $T(\Phi \wedge \Phi') = T(\Phi) \wedge T(\Phi')$,
- $T(\mathtt{X}\Phi) = \mathtt{X}T(\Phi)$,
- $T(\Phi \mathtt{U} \Phi') = T(\Phi)\mathtt{U}T(\Phi')$,
- $T(\mathtt{A}\Phi) = \mathtt{A}T(\Phi)$.

Nous avons le lemme suivant :

Lemme 6.3 *Soit Φ une formule de* $\mathrm{CTL}^*_{mem}[Q, P, X]$. *Pour toutes exécutions π de $TS(S)$ et π_C de $TS(S_C)$ telles que $|\pi| = |\pi_C|$ et telles que pour tout $i \in [0..|\pi| - 1]$, $\pi(i) \sim \pi_C(i)$ et pour tout entier $k \in [1..|\pi| - 1]$, nous avons $\pi, k \models \Phi$ si et seulement si $\pi_C, k \models T(\Phi)$.*

Preuve : La preuve se fait par induction structurelle sur la formule Φ. Tout d'abord montrons que si $\Phi = q$ ou si $\Phi = EMS$ le lemme est vrai.
Supposons que $\Phi = q$. Soient π une exécution de $TS(S)$ et π_C une exécution de $TS(S_C)$ vérifiant les conditions énoncées. Soit $k \in [0..|\pi| - 1]$. Nous avons $\pi(k) = (q', GM)$ et $\pi_C(k) = ((q'', FM), \mathbf{v})$. Comme $\pi(k) \sim \pi_C(k)$, nous avons nécessairement $q' = q''$ et par conséquent $\pi, k \models q$ si et seulement si $\pi_C, k \models \bigvee_{FM \in \mathcal{FM}} x_0 = (q, FM)$ et ceci si et seulement si $q' = q$.
Supposons que $\Phi = EMS$ avec $EMS = \{(FM_1, \phi_1), \ldots, (FM_t, \phi_t)\}$. Soient π une exécution de $TS(S)$ et π_C une exécution de $TS(S_C)$ vérifiant les conditions énoncées. Soit $k \in [0..|\pi| - 1]$. Nous notons $\pi(k) = (q, GM)$ et $\pi_C(k) = ((q, FM), \mathbf{v})$. Supposons que $\pi, k \models EMS$. Alors il existe $i \in [1..t]$ tel que $GM \in [\![(FM_i, \phi_i)]\!]$. Comme de plus $GM \sim (FM, \mathbf{v})$ (ceci car $\pi(k) \sim \pi_C(k)$), nous avons $GM = FM(\mathbf{v}_{|X_{FM}})$. Par conséquent, d'après la proposition 4.17, nous avons $FM_i \approx_f FM$ et comme $GM \in [\![(FM_i, \phi_i)]\!]$, nous avons également que $\mathbf{v}_{|X_{FM}} \models \phi_i^f$ et par conséquent $\mathbf{v} \models \phi_i^f$. Nous en déduisons que $\pi_C, k \models T(EMS)$. La réciproque disant que si $\pi_C, k \models T(EMS)$ alors $\pi, k \models EMS$ se prouve de manière identique.
Nous supposons maintenant le lemme vrai pour toutes les sous-formules de Φ et montrons que le lemme reste vrai pour Φ. Soient π une exécution de $TS(S)$ et π_C une exécution de $TS(S_C)$ vérifiant les conditions énoncées. Soit $k \in [0..|\pi| - 1]$.

- Si $\Phi = \neg\Phi'$, comme par hypothèse d'induction nous avons $\pi, k \models \Phi'$ si et seulement si $\pi_C, k \models T(\Phi')$ et comme $T(\neg\Phi') = \neg T(\Phi')$, nous avons bien $\pi, k \models \Phi$ si et seulement si $\pi_C, k \models T(\Phi)$.
- Si $\Phi = \Phi' \wedge \Phi''$, comme par hypothèse d'induction nous avons $\pi, k \models \Phi'$ si et seulement si $\pi_C, k \models T(\Phi')$ et $\pi, k \models \Phi''$ si et seulement si $\pi_C, k \models T(\Phi'')$ et comme $T(\Phi' \wedge \Phi') = T(\Phi') \wedge T(\Phi'')$, nous avons bien $\pi, k \models \Phi$ si et seulement si $\pi_C, k \models T(\Phi)$.
- Si $\Phi = \mathtt{X}\Phi'$. Si $k \geq |\pi| - 1$, nous avons $\pi, k \not\models \mathtt{X}\Phi$ et $\pi_C, k \not\models \mathtt{X}T(\Phi)$. Supposons que $k < |\pi| - 1$, alors par hypothèse d'induction nous avons $\pi, k + 1 \models \Phi'$ si et seulement si $\pi_C, k + 1 \models T(\Phi')$ et par conséquent nous avons bien que $\pi, k \models \mathtt{X}\Phi'$ si et seulement si $\pi_C, k \models \mathtt{X}T(\Phi')$.
- Si $\Phi = \Phi' \mathtt{U} \Phi''$. Si il existe $j \in \mathbb{N}$ tel que $i \leq j \leq |\pi|$ et tel que $\pi, j \models \Phi''$ et tel que pour tout $l \in \mathbb{N}$ vérifiant $i \leq l < j$, $\pi, l \models \Phi'$ alors par hypothèse d'induction $\pi_C, j \models T(\Phi'')$ et pour tout $l \in \mathbb{N}$ vérifiant $i \leq l < j$, $\pi_C, l \models T(\Phi')$. Par conséquent $\pi_C, k \models T(\Phi')\mathtt{U}T(\Phi'')$. Si il existe $j \in \mathbb{N}$ tel que $i \leq j \leq |\pi|$ et tel que $\pi_C, j \models T(\Phi'')$ et tel que pour tout $l \in \mathbb{N}$ vérifiant $i \leq l < j$, $\pi_C, l \models T(\Phi')$ alors par hypothèse d'induction $\pi, j \models \Phi''$ et pour tout $l \in \mathbb{N}$ vérifiant $i \leq l < j$, $\pi, l \models \Phi'$. Par conséquent $\pi, k \models \Phi' \mathtt{U} \Phi''$.
- Si $\Phi = \mathtt{A}\Phi'$. Supposons que pour toute exécution π' de $TS(S)$ telle que $\pi_{\leq k} = \pi'_{\leq k}$, nous avons $\pi', k \models \Phi'$. Soit π'_C une exécution de $TS(S_C)$ telle que $\pi_{C_{\leq k}} = \pi'_{C_{\leq k}}$. Alors en utilisant le lemme 5.6, comme $\pi(k) \sim \pi'_C(k)$, nous en déduisons qu'il existe une exécution de π' de $TS(S)$ telle que $\pi_{\leq k} = \pi'_{\leq k}$ et telle que $|\pi'| = |\pi'_C|$ et telle que pour tout $i \in [0..|\pi'| - 1]$, $\pi'(i) \sim \pi'_C(i)$.

En utilisant l'hypothèse d'induction, comme $\pi', k \models \Phi'$, nous avons $\pi'_C, k \models T(\Phi')$. Nous en déduisons que $\pi_C, k \models \mathtt{A}T(\Phi')$. De la même façon, nous pouvons montrer que si $\pi_C, k \models \mathtt{A}T(\Phi')$ alors $\pi, k \models \mathtt{A}\Phi'$.

□

En utilisant le résultat du théorème 3.3, qui nous dit que le problème de model-checking symbolique généralisé est décidable pour les systèmes à compteurs linéaires plats et à monoïde fini, nous obtenons alors le résultat suivant :

Théorème 6.4 *Le problème de model-checking est décidable pour les systèmes à pointeurs dont le système à compteurs associé est plat.*

Preuve : Nous considérons un ensemble P de pointeurs et un ensemble X de compteurs tel que $2|P| \leq |X|$. Soient $S = \langle Q, P, E\rangle$ un système à pointeurs, (q_0, EMS_0) une configuration symbolique initiale de $TS(S)$ avec $EMS_0 = \{(FM_1, \phi_1), \ldots, (FM_t, \phi_t)\}$ et Φ une formule de la logique $\mathrm{CTL}^*_{mem}[Q, P, X]$. Soit $S_C = \langle Q_C, X, E_C\rangle$ le système à compteurs associé à S dont la construction est décrite dans le chapitre précédent. Nous prenons un entier $i \in [1..t]$. Nous allons montrer que pour toutes les exécutions π de $TS(S)$ vérifiant $\pi(0) \in \{q_0\} \times [\![(FM_i, \phi_i)]\!]$, nous avons $\pi, 0 \models \Phi$ si et seulement si pour toutes les exécutions π_C de $TS(S_C)$ vérifiant $\pi_C(0) \in \{q_0, FM_i\} \times [\![\phi_i]\!]$ nous avons $\pi_C, 0 \models T(\Phi)$. Remarquons que une fois que nous aurons montré cela, comme le système à compteurs linéaire S_C est plat (par hypothèse) et à monoïde fini (d'après le lemme 5.5) en utilisant le théorème 3.3, nous pourrons déduire le résultat énoncé en utilisant le fait que le problème du model-checking symbolique généralisé est décidable pour les systèmes à compteurs linéaires plats et à monoïde fini.

Supposons que pour toutes les exécutions π de $TS(S)$ vérifiant $\pi(0) \in \{q_0\} \times [\![(FM_i, \phi_i)]\!]$ nous avons $\pi, 0 \models \Phi$. Soit π_C une exécution de $TS(S_C)$ vérifiant $\pi_C(0) \in \{q_0, FM_i\} \times [\![\phi_i]\!]$. Supposons que $\pi_C(0) = ((q_0, FM_i), \mathbf{v})$ alors nous avons $\mathbf{v} \models \phi_i$ et comme $\phi \in \mathbf{Presb}(X_{FM_i})$, nous avons $\mathbf{v}_{|X_{FM_i}} \models \phi_i$ et par conséquent $FM_i(\mathbf{v}_{|X_{FM_i}}) \in [\![(FM_i, \phi_i)]\!]$. Nous posons $c_0 = (q_0, FM_i(\mathbf{v}_{|X_{FM_i}}))$, par définition de $\sim$, nous avons $c_0 \sim \pi_C(0)$. En utilisant le lemme 5.6, nous montrons facilement par induction qu'il existe alors une exécution π de $TS(S)$ tel que $|\pi| = |\pi_C|$ et $\pi(0) = c_0$ et pour tout $j \in [0..|\pi| - 1]$, $\pi(j) \sim \pi_C(j)$. Comme $\pi, 0 \models \Phi$, en utilisant le lemme 6.3, nous déduisons que $\pi_C, 0 \models T(\Phi)$. De la même façon, nous pouvons montrer que si pour toutes les exécutions π_C de $TS(S_C)$ vérifiant $\pi_C(0) \in \{q_0, FM_i\} \times [\![\phi_i]\!]$ nous avons $\pi_C, 0 \models T(\Phi)$, alors pour toutes les exécutions π de $TS(S)$ vérifiant $\pi(0) \in \{q_0\} \times [\![(FM_i, \phi_i)]\!]$, nous avons $\pi, 0 \models \Phi$. □

Nous avons vu avec la proposition 5.14 que pour les systèmes à pointeurs plats sans déplacement au successeur (c'est-à-dire sans action de la forme $p' := p.\mathtt{succ}$), le système à compteurs associé était plat, nous avons donc le corollaire suivant :

Corollaire 6.5 *Le problème de model-checking est décidable pour les systèmes à pointeurs plats sans déplacement au successeur.*

Dans les sections qui suivent nous allons nous intéresser au même problème pour d'autres classes de systèmes à pointeurs plats.

6.2.2 Une nouvelle traduction vers des systèmes à compteurs

Nous présentons ici une nouvelle traduction vers les systèmes à compteurs dont nous nous servirons pour montrer que certains des problèmes considérés précédemment sont décidables pour les systèmes à pointeurs plats sans mise à jour destructive. Nous appelons mise à jour destructive ("destructive update" en anglais) les actions de la forme $p.\texttt{succ} := p'$ ou $\texttt{free}(p)$ avec $p, p' \in P$. Remarquons que ces actions ont la particularité de modifier les successeurs d'un noeud dans un graphe mémoire, contrairement aux autres actions qui ne font que déplacer les variables dans le graphe ou créer un nouveau noeud. L'idée principale consiste à proposer une nouvelle traduction qui conserve la platitude pour les systèmes sans mise à jour destructive. Comme nous l'avons vu dans la section 5.1.3, ceci n'est pas le cas de la première traduction que nous avons donnée. De plus, il s'avère que nous devrons aussi faire une distinction entre les systèmes à pointeurs utilisant des tests d'alias et ceux n'en utilisant pas. Une garde de pointeurs utilise un test d'alias si et seulement si elle utilise des gardes de la forme $p == p'$ avec $p, p' \in P$. Ainsi nous dirons qu'un système à pointeurs S est sans test d'alias si aucune des gardes apparaissant dans les transitions de S n'utilise de test d'alias. Les systèmes à pointeurs sans test d'alias peuvent ainsi seulement tester si des variables pointent ou non vers le noeud `null`.

6.2.2.1 Formes mémoire restreintes

Tout comme la traduction précédente utilisait les formes mémoire pour caractériser des graphes mémoire, nous introduisons maintenant une nouvelle façon de représenter des graphes mémoire, les formes mémoire restreintes, dont nous nous servirons pour construire un système à compteurs.

Definition 6.6 (Forme mémoire restreinte) *Une forme mémoire restreinte est une forme mémoire $FMR = (N, P, X, succ, loc, c)$ telle que pour tout $n \in N$, si $loc^{-1}(\{n\}) \neq \emptyset$ alors soit $succ^{-1}(\{n\}) = \emptyset$, soit $succ^{-1}(\{n\}) = \{n\}$.*

En d'autres termes, les formes mémoire restreintes sont des formes mémoire pour lesquelles chaque noeud pointé par une variable est soit un noeud à la tête d'une liste, c'est-à-dire un noeud sans prédécesseur, soit l'unique noeud d'une liste cyclique qui n'est accessible par aucun autre noeud. Nous notons $\mathcal{FMR}_{P,X}$ l'ensemble des formes mémoire restreintes utilisant P et X comme ensembles de pointeurs et de compteurs.

Notation 6.7 *Comme pour les formes mémoire, nous ne noterons pas P et X dans $\mathcal{FMR}_{P,X}$ en écrivant seulement $\mathcal{FMR}$ lorsque ces ensembles seront explicites.*

Nous introduisons quelques notations sur les formes mémoire restreintes qui nous seront utiles par la suite. Soit $FMR = (N, P, X, succ, loc, c)$ une forme mémoire restreinte. Comme nous l'avons déjà fait pour les formes mémoire, nous étendons la fonction **List** aux formes mémoire restreintes. Pour rappel, si n est un noeud de N, alors $\textbf{List}(FMR, n) = \{m \in N \cup \{\texttt{null}, \bot\} \mid \exists i \in \mathbb{N} \text{ tel que } m = succ^i(n)\}$. Rappelons de plus que $\textbf{List}(FMR, \texttt{null}) = \{\texttt{null}\}$ et $\textbf{List}(FMR, \bot) = \{\bot\}$. Et si p est un pointeur dans P, nous notons $\textbf{List}(FMR, p)$ l'ensemble $\textbf{List}(FMR, loc(p))$. Nous introduisons également une notation pour décrire l'ensemble des noeuds se trouvant entre deux noeuds donnés. Ainsi pour $n, n' \in N$, nous définissons l'ensemble $\textbf{Entre}(FMR, n, n')$ par :

- si $n' \notin \textbf{List}(FMR, n)$ alors $\textbf{List}(FMR, n, n') = \emptyset$,
- si $n' = n$ alors $\textbf{List}(FMR, n, n') = \emptyset$,
- si $n' \neq n$ et $n' \in \textbf{List}(FMR, n)$, alors $\textbf{Entre}(FMR, n, n') = \{n_1, \ldots, n_r\}$ tel que $n_1 = succ(n)$ et $n' = succ(n_r)$ et pour tout $i \in [1..r-1]$, $succ(n_i) = n_{i+1}$ et pour tout $i \in [1..r]$, $n_i \notin \{n, n'\}$.

Étant donnés deux noeuds $n, n' \in N$, nous définissons aussi l'ensemble **Partage**$(FMR, n, n') =$ **List**$(FMR, n) \cap$ **List**(FMR, n') des noeuds partagés par les listes partant des noeuds n et n'. De plus, nous dirons qu'un pointeur p est seul dans FMR si et seulement si $loc^{-1}(\{loc(p)\}) = \{loc(p)\}$ c'est-à-dire que p est la seule variable pointant vers le noeud $loc(p)$. Finalement, nous dirons qu'un noeud $n \in N$ est sur une liste cyclique dans FMR si et seulement si $n \in$ **List**$(FMR, \texttt{succ}(n))$. Une forme mémoire restreinte $FMR = (N, P, X, succ, loc, c)$ est acyclique si et seulement si pour tout noeud $n \in N$, n n'est pas sur une liste cyclique dans FMR. Ces notions peuvent également être appliquées à des formes mémoire et à des graphes mémoire.

Nous considérerons par la suite $P = \{p_1, \ldots, p_n\}$ un ensemble de pointeurs et X un ensemble de compteurs. Nous définissons alors l'ensemble de compteurs $Y_P = \{y_1, \ldots, y_n\}$ tel que $X \cap Y_P = \emptyset$, obtenu en associant à chaque pointeur p_i de P un compteur y_i. Comme pour les formes mémoire, si $FMR = (N, P, X, succ, loc, c)$ est une forme mémoire restreinte, X_{FMR} représente le sous-ensemble de X tel que $c(N) = X_{FMR}$. Si $(FMR, \mathbf{v})$ est une paire telle que $FMR = (N, P, X, succ, loc)$ est une forme mémoire restreinte et $\mathbf{v}$ est une fonction de $X_{FMR} \to \mathbb{N}^*$ et si $\mathbf{u} : Y_P \to \mathbb{N}$ est une fonction associant une valeur entière à chaque variable de Y_P, nous dirons que $\mathbf{u}$ est admissible pour $(FMR, \mathbf{v})$ si et seulement si pour tout $p_i \in P$ tel que $loc(p_i) \notin \{\texttt{null}, \bot\}$, les conditions suivantes sont respectées :

1. si $loc(p_i) \in \{\texttt{null}, \bot\}$ alors $\mathbf{u}(p_i) = 0$,
2. pour tout noeud $n \in N$ tel que $\texttt{succ}^{-1}(\{n\}) = \emptyset$, il existe $p_i \in P$ tel que $\mathbf{u}(y_i) = 0$, et,
3. si $succ^{-1}(loc(p_i)) = \emptyset$ et si **List**$(FMR, p_i) \cap \{\bot, \texttt{null}\} \neq \emptyset$ alors :

$$\mathbf{u}(y_i) \leq \sum_{x \in \textbf{List}(FMR, p_i) \setminus \{\bot, \texttt{null}\}} \mathbf{v}(x)$$

Remarquons que pour le cas 3, comme **List**$(FMR, p_i) \cap \{\texttt{null}, \bot\} \neq \emptyset$, l'ensemble des noeuds appartenant à **List**(FMR, p_i) est nécessairement fini, ceci car les noeuds $\bot$ et `null` n'ont pas de successeur. Intuitivement, nous posons ces conditions car la valuation $\mathbf{u} : Y_P \to \mathbb{N}$ va nous servir à positionner dans le graphe chaque pointeur p à une distance $\mathbf{u}(p)$ du noeud où il pointe dans FMR, et comme nous le verrons, si $\mathbf{u}$ n'est pas admissible, nous ne pourrons pas construire le graphe mémoire correspondant.

De la même façon que nous avons associé une valuation de compteurs à une forme mémoire, nous définissons maintenant les formes mémoires restreintes valuées.

Definition 6.8 (Forme mémoire restreinte valuée) *Une forme mémoire restreinte valuée est un triplet $(FMR, \boldsymbol{v}, \boldsymbol{u})$ telle que :*

- *FMR est une forme mémoire restreinte de $\mathcal{FMR}_{P,X}$,*
- *$\boldsymbol{v} : X_{FMR} \to \mathbb{N}^*$ est une fonction qui associe à chaque compteur de FMR une valeur strictement positive,*
- *$\boldsymbol{u} : Y_P \to \mathbb{N}$ est une fonction qui associe à chaque compteur de Y_P une valeur entière, de telle sorte que $\boldsymbol{u}$ est admissible pour $(FMR, \boldsymbol{v})$.*

Comme nous avons associé un graphe mémoire à une forme mémoire valuée (cf. section 4.3.1.2), nous pouvons construire un unique graphe mémoire à partir d'une forme mémoire restreinte valuée. Soit $(FMR, \mathbf{v}, \mathbf{u})$ une forme mémoire restreinte valuée, nous lui associons le graphe mémoire

$FMR[\mathbf{v},\mathbf{u}]$ construit de la façon suivante. Nous commençons par construire le graphe mémoire $GM = (N, P, succ, loc)$ tel que $GM = FMR(\mathbf{v})$, c'est-à-dire en considérant $(FMR, \mathbf{v})$ comme une forme mémoire valuée et en appliquant la construction vue à la section 4.3.1.2 du chapitre 4. Nous avons ensuite $FMR[\mathbf{v},\mathbf{u}] = (N, P, succ, loc')$ où loc' est tel que pour tout p vérifiant $loc(p) \in \{\texttt{null}, \bot\}$, nous avons $loc'(p) = loc(p)$ et pour tout $p_i \in P$ vérifiant $loc(p_i) \notin \{\texttt{null}, \bot\}$, $loc'(p_i) = succ^{\mathbf{u}(y_i)}(loc(p_i))$. Ainsi les conditions 1, 2 et 3 écrites pour définir les valuations admissibles $\mathbf{u}$ prennent ici tout leur sens. En effet, on voit que si pour un noeud $n \in N$ tel que $\texttt{succ}^{-1}(\{n\}) = \emptyset$, il n'existe pas de $p_i \in P$ telle que $\mathbf{u}(y_i) = 0$, alors $FMR[\mathbf{v},\mathbf{u}]$ n'est plus un graphe mémoire, car il existe des noeuds qui sont accessibles par aucune variable. Quant à la condition 3, elle sert à s'assurer que le noeud $succ^{\mathbf{u}(y_i)}(loc(p))$ est bien définie. Remarquons juste que si $\mathbf{List}(FMR, p_i) \cap \{\bot, \texttt{null}\} = \emptyset$, alors pour tout $j \in \mathbb{N}$, le noeud $succ^j(loc(p))$ est bien défini, car le fait que $\mathbf{List}(FMR, p_i) \cap \{\bot, \texttt{null}\} = \emptyset$ implique qu'il existe un noeud $n \in \mathbf{List}(FMR, p_i)$ tel que n est sur une liste cyclique dans FMR.

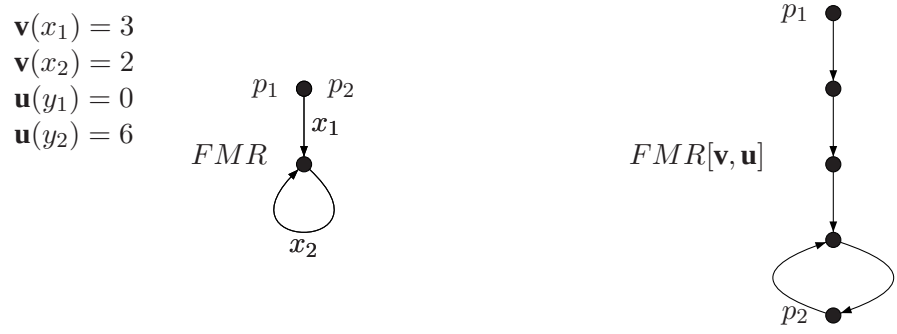


FIGURE 6.1 – Une forme mémoire restreinte valuée $(FMR, \mathbf{v}, \mathbf{u})$ et le graphe mémoire correspondant

Exemple 6.9 *La figure 6.1 montre comment à partir d'une forme mémoire restreinte valuée, on construit le graphe mémoire correspondant. Ainsi nous voyons que dans* $\mathbf{List}(FMR, p_2)$*, il y a un noeud* n *appartenant à une liste cyclique, et donc la valeur de* $\mathbf{u}(y_2)$ *peut-être aussi grande que l'on veut, et en appliquant* $\mathbf{u}(y_2)$ *fois la fonction* $succ$*, on "tourne" dans cette liste cyclique.*

Nous avons de plus la proposition suivante :

Proposition 6.10 *Pour tout graphe mémoire* $GM \in \mathcal{GM}_P$*, pour tout ensemble de compteurs* X *avec* $2|P| \leq |X|$*, il existe une forme mémoire restreinte valuée* $(FMR, \boldsymbol{v}, \boldsymbol{u})$ *avec* $FMR \in \mathcal{FMR}_{P,X}$ *telle que* $GM = FMR[\boldsymbol{v}, \boldsymbol{u}]$*.*

Idée de la preuve : À partir du graphe mémoire GM, on commence par construire la forme mémoire valuée $(FM, \mathbf{v})$ telle que $GM = FM(\mathbf{v})$. Une telle forme mémoire valuée existe d'après la proposition 4.15. On modifie ensuite la forme mémoire valuée et la valuation $\mathbf{v}$ de façon à mettre les variables qui sont accessibles par un noeud sans prédécesseur sur ce noeud, et les variables qui pointent vers un noeud appartenant à une liste cyclique qui n'est pas accessible par un noeud sans prédécesseur sont toutes rattachées à un unique noeud. Ensuite, on supprime les noeuds non étiquetés par une variable et ayant un unique prédécesseur, et on met à jour les valeurs des compteurs de façon à obtenir une forme mémoire restreinte valuée telle que $GM = FMR[\mathbf{v},\mathbf{u}]$ □

6.2.2.2 Définition de la traduction

Soit $P = \{p_1, \ldots, p_n\}$ un ensemble de compteurs et X un ensemble de compteurs tel que $2|P| \leq |X|$. Nous définissons X_P l'ensemble de compteurs égal à $X \uplus Y_P$. Pour rappel $Y_P = \{y_1, \ldots, y_n\}$ est un ensemble de compteurs associé aux pointeurs de P.

Notation 6.11 *Dans la suite de ce chapitre, lorsque nous considérerons une valuation* $\mathbf{v} : X_P \to \mathbb{N}$*, nous noterons* $\mathbf{v}_P$ *la restriction de* $\mathbf{v}$ *aux variables dans* Y_P *et si* FMR *est une forme mémoire restreinte, nous noterons* $\mathbf{v}_{FMR}$ *la restriction de* $\mathbf{v}$ *aux variables dans* X_{FMR}*.*

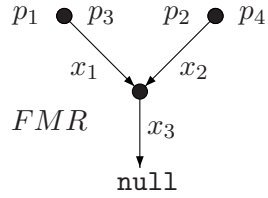
Nous allons commencer par définir une fonction **Test** : $\mathcal{G}_P \times \mathcal{FMR}_{P,X} \to$ **Presb**(X_P) qui étant données une garde de pointeurs et une forme mémoire restreinte retourne une formule de Presburger. Ainsi pour $g \in \mathcal{G}_P$ et $FMR \in \mathcal{FMR}$, nous souhaitons construire une formule **Test**(g, FMR) de telle façon que la propriété suivante soit vérifiée : pour toute valuation $\mathbf{v} : X_P \to \mathbb{N}$ telle que $(FMR, \mathbf{v}_{FMR}, \mathbf{v}_P)$ est une forme mémoire restreinte valuée, nous avons :

$$FMR[\mathbf{v}_{FMR}, \mathbf{v}_P] \models g \text{ si et seulement si } \mathbf{v} \models \textbf{Test}(g, FMR)$$

Toutefois, remarquons que si FMR est une forme mémoire contenant une liste cyclique, et si g est une garde sur les pointeurs de la forme $p_i == p_j$ alors la formule arithmétique engendrée devrait utiliser des propositions de la forme x divise y qui ne peuvent pas être décrites dans l'arithmétique de Presburger. Ainsi si p_i et p_j sont deux variables de pointeurs pointant dans FMR sur l'unique noeud d'une liste cyclique dont l'arc est étiqueté par le compteur x, tester si $p_i == p_j$ se ferait grâce à la formule $(y_i \geq y_j \Rightarrow x | y_i - y_j) \vee (y_j \geq y_i \Rightarrow x | y_j - y_i)$ (où le symbole $|$ veut dire ici "est un diviseur entier de"). Pour éviter cette situation, nous restreignons le domaine de **Test** de telle sorte que **dom**(**Test**) $= \{(g, FMR) \in \mathcal{G}_P \times \mathcal{FMR}_{P,X} \mid g$ n'est pas un test d'alias ou FMR est acyclique$\}$. Soient $g \in \mathcal{G}_P$ une garde sur les pointeurs et $FMR = (N, P, X, succ, loc, c)$ une forme mémoire restreinte telles que $(g, FMR) \in$ **dom**(**Test**), la figure 6.3 donne la définition de **Test**(g, FMR) par induction sur g.

Exemple 6.12 *La figure 6.2 nous montre le résultat de la fonction* **Test** *appliquée à la garde* $p_1 == p_2$ *et à la forme mémoire restreinte* FMR *dessinée. Si nous considérons une valuation* $\mathbf{v} : X_P \to \mathbb{N}$ *telle que* $(FMR, \mathbf{v}_{FMR}, \mathbf{v}_P)$ *est une forme mémoire restreinte valuée, alors pour que la garde* $p_1 == p_2$ *soit satisfaite par* $FMR[\mathbf{v}_{FMR}, \mathbf{v}_P]$*, il faut déjà que les variables* p_1 *et* p_2 *appartiennent à la même liste simplement chaînée, c'est à cela que servent les conditions* $y_1 \geq x_1$ *et* $y_2 \geq x_2$ *et il faut que la position du noeud pointée par chacune de ces deux variables soit la même dans cette liste chaînée, d'où la condition* $y_1 - x_1 = y_2 - x_2$*.*

Avant de donner les propriétés vérifiées par la fonction **Test**, nous construisons, comme nous l'avons fait pour la traduction présentée au chapitre précédent, une fonction $\textbf{POST}_2 : \mathcal{A}_P \times \mathcal{FMR}_{P,X} \to \mathcal{P}(\textbf{Presb}(X_P, X'_P) \times \mathcal{FMR}^{err}_{P,X})$ avec $\mathcal{FMR}^{err}_{P,X} = \mathcal{FMR}_{P,X} \cup \{\texttt{Seg}, \texttt{Leak}, \texttt{Seg}\}$. Cette fois-ci au lieu de prendre comme argument une forme mémoire (comme c'est le cas de la fonction **POST**), la fonction $\textbf{POST}_2$ prend comme argument une forme mémoire restreinte. Nous restreignons là encore le domaine de la fonction $\textbf{POST}_2$ de telle façon que **dom**$(\textbf{POST}_2) = \{(a, FMR) \in \mathcal{A}_P \times \mathcal{FMR}_{P,X} \mid$ a n'est pas une mise à jour destructive$\}$, ceci car nous nous servons de cette traduction pour établir des résultats de décidabilité sur les systèmes à pointeurs sans mise à jour destructive. Le principe de la fonction $\textbf{POST}_2$ que nous définissons ici est le même que celui de la fonction **POST** définie



$$\textbf{Test}(g, FMR) = \begin{array}{l} y_1 \geq x_1 \wedge \\ y_2 \geq x_2 \wedge \\ y_1 - x_1 = y_2 - x_2 \end{array}$$

FIGURE 6.2 – Exemple de l'application de la fonction **Test** avec la garde $g = \ p_1 == p_2$

au chapitre précédent. Nous construisons ainsi la fonction $\textbf{POST}_2$ de telle sorte que pour toute paire $(\phi, FMR') \in \textbf{POST}_2(a, FMR)$, où a n'est pas une mise à jour destructive, pour toutes valuations $\mathbf{v}, \mathbf{v'} : X_P \rightarrow \mathbb{N}$ telles que $(FMR, \mathbf{v}_{FMR}, \mathbf{v}_P)$ et $(FMR', \mathbf{v'}_{FMR}, \mathbf{v'}_P)$ sont des formes mémoire restreintes valuées, nous avons :

$$[\![a]\!]_P(FMR[\mathbf{v}_{FMR}, \mathbf{v}_P]) = FMR'[\mathbf{v'}_{FMR}, \mathbf{v'}_P] \text{ si et seulement si } (\mathbf{v}, \mathbf{v'}) \models \phi$$

Les figures 6.4 à 6.9 donnent la définition de la fonction $\textbf{POST}_2$ en fonction des différentes actions sur les pointeurs possibles. Pour chaque forme mémoire restreinte possible FMR, nous fournissons les paires (ϕ, FMR') appartenant à l'ensemble $\textbf{POST}_2(a, FMR)$. Pour l'action `skip`, nous avons $\textbf{POST}_2(\texttt{skip}, FMR) = \{(\bigwedge_{x \in X_P} x' = x, FMR)\}$.

Hypothèses	**Test**(g, FMR)
$g = true$	$true$
$g = p_i == \texttt{null}$ et $loc(p_i) = \texttt{null}$	$true$
$g = p_i == \texttt{null}$ et soit $loc(p_i) \notin \{\texttt{null}, \bot\}$ et $\texttt{null} \notin$ **List**(FMR, p_i) soit $loc(p_i) = \bot$	$false$
$g = p_i == \texttt{null}$ et $loc(p_i) \notin \{\texttt{null}, \bot\}$ et $\texttt{null} \in$ **List**(FMR, p_i)	$y_i = \sum_{n \in \textbf{List}(FMR, p_i) \cap N} c(n)$
$g = p_i == p_j$ et **Partage**$(FMR, p_i, p_j) = \emptyset$	$false$
$g = p_i == p_j$ et **Partage**$(FMR, p_i, p_j) \neq \emptyset$	$y_i \geq \sum_{n \in \textbf{List}(FMR, p_i) \setminus \textbf{Partage}(FMR, p_i, p_j)} c(n) \wedge$ $y_j \geq \sum_{n \in \textbf{List}(FMR, p_j) \setminus \textbf{Partage}(FMR, p_i, p_j)} c(n) \wedge$ $y_i - \sum_{n \in \textbf{List}(FMR, p_i) \setminus \textbf{Partage}(FMR, p_i, p_j)} c(n) =$ $y_j - \sum_{n' \in \textbf{List}(FMR, p_j) \setminus \textbf{Partage}(FMR, p_i, p_j)} c(n')$
$g = \neg g'$	$\neg T(g', FMR)$
$g = g' \wedge g''$	$T(g', FMR) \wedge T(g'', FMR)$

FIGURE 6.3 – Définition de **Test**(g, FMR)

Hypothèses	**POST**$_2(a, FMR)$	
	ϕ	FMR'
$FMR = (N, P, X, succ, loc, c)$ et $loc(p_i) \in \{\texttt{null}, \bot\}$ et $p_i \neq p_j$	$y'_i = y_j \wedge$ $\bigwedge_{x \in X_p \setminus \{y_i\}} x' = x$	$(N, P, X, succ, loc[p_i \mapsto loc(p_j)], c)$
$p_i = p_j$	$\bigwedge_{x \in X_p} x' = x$	FMR
$FMR = (N, P, X, succ, loc, c)$ et $loc(p_i) \notin \{\texttt{null}, \bot\}$ et p_i n'est pas seul dans FMR et $p_i \neq p_j$ et $loc(p_i)$ n'est pas sur une liste cyclique	$\bigvee_{p_k \in loc^{-1}(\{loc(p_i)\}) \setminus \{p_i\}} y_k = 0 \wedge$ $y'_i = y_j \wedge$ $\bigwedge_{x \in X_p \setminus \{y_i\}} x' = x$	$(N, P, X, succ, loc[p_i \mapsto loc(p_j)], c)$
	$\bigwedge_{p_k \in loc^{-1}(\{loc(p_i)\}) \setminus \{p_i\}} y_k \neq 0 \wedge$ $\bigwedge_{x \in X_p} x' = x$	`Leak`
$FMR = (N, P, X, succ, loc, c)$ et $loc(p_i) \notin \{\texttt{null}, \bot\}$ et p_i n'est pas seul dans FMR et $p_i \neq p_j$ et $loc(p_i)$ est sur une liste cyclique	$y'_i = y_j \wedge$ $\bigwedge_{x \in X_p \setminus \{y_i\}} x' = x$	$(N, P, X, succ, loc[p_i \mapsto loc(p_j)], c)$
$FMR = (N, P, X, succ, loc, c)$ et p_i est seul dans FMR et $loc(p_i) \notin \{\texttt{null}, \bot\}$ et $p_i \neq p_j$	$\bigwedge_{x \in X_p} x' = x$	`Leak`

FIGURE 6.4 – Définition de **POST**$_2$ pour l'action $a = (p_i := p_j)$

Hypothèses	**POST**$_2(a, FMR)$	
	ϕ	FMR'
$FMR = (N, P, X, succ, loc, c)$ et $loc(p_i) \in \{\texttt{null}, \bot\}$	$\bigwedge_{x \in X_p} x' = x$	$(N, P, X, succ, loc[p_i \mapsto \texttt{null}], c)$
$FMR = (N, P, X, succ, loc, c)$ et $loc(p_i) \notin \{\texttt{null}, \bot\}$ et p_i n'est pas seul dans FMR et $loc(p_i)$ n'est pas sur une liste cyclique	$\bigvee_{p_k \in loc^{-1}(\{loc(p_i)\}) \setminus \{p_i\}} y_k = 0 \wedge$ $y'_i = 0 \wedge$ $\bigwedge_{x \in X_p \setminus \{y_i\}} x' = x$	$(N, P, X, succ, loc[p_i \mapsto \texttt{null}], c)$
	$\bigwedge_{p_k \in loc^{-1}(\{loc(p_i)\}) \setminus \{p_i\}} y_k \neq 0 \wedge$ $\bigwedge_{x \in X_p} x' = x$	`Leak`
$FMR = (N, P, X, succ, loc, c)$ et $loc(p_i) \notin \{\texttt{null}, \bot\}$ et p_i n'est pas seul dans FMR et $loc(p_i)$ est sur une liste cyclique	$y'_i = 0 \wedge$ $\bigwedge_{x \in X_p \setminus \{y_i\}} x' = x$	$(N, P, X, succ, loc[p_i \mapsto \texttt{null}], c)$
$FMR = (N, P, X, succ, loc, c)$ et p_i est seul dans FMR et $loc(p_i) \notin \{\texttt{null}, \bot\}$	$\bigwedge_{x \in X_p} x' = x$	`Leak`

FIGURE 6.5 – Définition de **POST**$_2$ pour l'action $a = (p_i := \texttt{null})$

Hypothèses	**POST**$_2(a, FMR)$	
	ϕ	FMR'
$FMR = (N, P, X, succ, loc, c)$ et $loc(p_i) \in \{\texttt{null}, \bot\}$ et $loc(p_j) \notin \{nullp, \bot\}$ et $\exists n \in$ **List**(FMR, p_j) tel que n est sur une liste cyclique	$y'_i = y_j + 1 \wedge$ $\bigwedge_{x \in X_p \setminus \{y_i\}} x' = x$	$(N, P, X, succ, loc[p_i \mapsto loc(p_j)], c)$
$FMR = (N, P, X, succ, loc, c)$ et $loc(p_i) \in \{\texttt{null}, \bot\}$ et $loc(p_j) \notin \{\texttt{null}, \bot\}$ et $\nexists n \in$ **List**(FMR, p_j) tel que n est sur une liste cyclique	$y_j < \sum_{m \in \textbf{List}(FMR, p_j) \cap N} c(m) \wedge$ $y'_i = y_j + 1 \wedge$ $\bigwedge_{x \in X_p \setminus \{y_i\}} x' = x$	$(N, P, X, succ, loc[p_i \mapsto loc(p_j)], c)$
	$y_j \geq \sum_{m \in \textbf{List}(FMR, p_j) \cap N} c(m) \wedge$ $\bigwedge_{x \in X_p} x' = x$	`Seg`
$FMR = (N, P, X, succ, loc, c)$ et $loc(p_i) \notin \{\texttt{null}, \bot\}$ et $loc(p_j) \notin \{\texttt{null}, \bot\}$ et p_i n'est pas seul dans FMR et $loc(p_i)$ n'est pas sur une liste cyclique et $\exists n \in$ **List**(FMR, p_j) tel que n est sur une liste cyclique	$y_j < \sum_{m \in \textbf{List}(FMR, p_j) \cap N} c(m) \wedge$ $\bigvee_{p_k \in loc^{-1}(\{loc(p_i)\}) \setminus \{p_i\}} y_k = 0 \wedge$ $y'_i = y_j + 1 \wedge$ $\bigwedge_{x \in X_p \setminus \{y_i\}} x' = x$	$(N, P, X, succ, loc[p_i \mapsto loc(p_j)], c)$
	$y_j < \sum_{m \in \textbf{List}(FMR, p_j) \cap N} c(m) \wedge$ $\bigwedge_{p_k \in loc^{-1}(\{loc(p_i)\}) \setminus \{p_i\}} y_k \neq 0 \wedge$ $\bigwedge_{x \in X_p} x' = x$	`Leak`
	$y_j \geq \sum_{m \in \textbf{List}(FMR, p_j) \cap N} c(m) \wedge$ $\bigvee_{p_k \in loc^{-1}(\{loc(p_i)\}) \setminus \{p_i\}} y_k = 0 \wedge$ $\bigwedge_{x \in X_p} x' = x$	`Seg`
	$y_j \geq \sum_{m \in \textbf{List}(FMR, p_j) \cap N} c(m) \wedge$ $\bigwedge_{p_k \in loc^{-1}(\{loc(p_i)\}) \setminus \{p_i\}} y_k \neq 0 \wedge$ $\bigwedge_{x \in X_p} x' = x$	`LeakSeg`
$FMR = (N, P, X, succ, loc, c)$ et $loc(p_i) \notin \{\texttt{null}, \bot\}$ et $loc(p_j) \notin \{\texttt{null}, \bot\}$ et p_i n'est pas seul dans FMR et $loc(p_i)$ n'est pas sur une liste cyclique et $\exists n \in$ **List**(FMR, p_j) tel que n est sur une liste cyclique	$\bigvee_{p_k \in loc^{-1}(\{loc(p_i)\}) \setminus \{p_i\}} y_k = 0 \wedge$ $y'_i = y_j + 1 \wedge$ $\bigwedge_{x \in X_p \setminus \{y_i\}} x' = x$	$(N, P, X, succ, loc[p_i \mapsto loc(p_j)], c)$
	$\bigwedge_{p_k \in loc^{-1}(\{loc(p_i)\}) \setminus \{p_i\}} y_k \neq 0 \wedge$ $\bigwedge_{x \in X_p} x' = x$	`Leak`

FIGURE 6.6 – Définition de **POST**$_2$ pour l'action $a = (p_i := p_j.\texttt{succ})$ (i)

Hypothèses	$\mathbf{POST}_2(a, FMR)$	
	ϕ	FMR'
$FMR = (N, P, X, succ, loc, c)$ et $loc(p_i) \notin \{\texttt{null}, \bot\}$ et $loc(p_j) \notin \{\texttt{null}, \bot\}$ et p_i n'est pas seul dans FMR et $loc(p_i)$ est sur une liste cyclique et $\nexists n \in$ **List**(FMR, p_j) tel que n est sur une liste cyclique	$y_j < \sum_{m \in \textbf{List}(FMR,p_j) \cap N} c(m) \wedge$ $y_i' = y_j + 1 \wedge$ $\bigwedge_{x \in X_p \setminus \{y_i\}} x' = x$	$(N, P, X, succ, loc[p_i \mapsto loc(p_j)], c)$
	$y_j \geq \sum_{m \in \textbf{List}(FMR,p_j) \cap N} c(m) \wedge$ $\bigwedge_{x \in X_p} x' = x$	`Seg`
$FMR = (N, P, X, succ, loc, c)$ et $loc(p_i) \notin \{\texttt{null}, \bot\}$ et $loc(p_j) \notin \{\texttt{null}, \bot\}$ et p_i n'est pas seul dans FMR et $loc(p_i)$ est sur une liste cyclique et $\exists n \in$ **List**(FMR, p_j) tel que n est sur une liste cyclique	$y_i' = y_j + 1 \wedge$ $\bigwedge_{x \in X_p \setminus \{y_i\}} x' = x$	$(N, P, X, succ, loc[p_i \mapsto loc(p_j)], c)$
$FMR = (N, P, X, succ, loc, c)$ et $loc(p_i) \notin \{\texttt{null}, \bot\}$ et $loc(p_j) \notin \{\texttt{null}, \bot\}$ et p_i est seul dans FMR et $loc(p_i)$ n'est pas sur une liste cyclique et $\exists n \in$ **List**(FMR, p_j) tel que n est sur une liste cyclique	$\bigwedge_{x \in X_P} x' = x$	`Leak`
$FMR = (N, P, X, succ, loc, c)$ et $loc(p_i) \notin \{\texttt{null}, \bot\}$ et $loc(p_j) \notin \{\texttt{null}, \bot\}$ et p_i est seul dans FMR et $loc(p_i)$ n'est pas sur une liste cyclique et $\nexists n \in$ **List**(FMR, p_j) tel que n est sur une liste cyclique	$y_j < \sum_{m \in \textbf{List}(FMR,p_j) \cap N} c(m) \wedge$ $\bigwedge_{x \in X_P} x' = x$	`Leak`
	$y_j \geq \sum_{m \in \textbf{List}(FMR,p_j) \cap N} c(m) \wedge$ $\bigwedge_{x \in X_P} x' = x$	`LeakSeg`
$FMR = (N, P, X, succ, loc, c)$ et $loc(p_i) \notin \{\texttt{null}, \bot\}$ et $loc(p_j) \notin \{\texttt{null}, \bot\}$ et p_i est seul dans FMR et $loc(p_i)$ est sur une liste cyclique et $p_i = p_j$	$y_i' = y_j + 1 \wedge$ $\bigwedge_{x \in X_P} x' = x$	FMR

FIGURE 6.7 – Définition de $\mathbf{POST}_2$ pour l'action $a = (p_i := p_j.\texttt{succ})$ (ii)

Hypothèses	**POST**$_2(a, FMR)$	
	ϕ	FMR'
$FMR = (N, P, X, succ, loc, c)$ et $loc(p_i) \notin \{\texttt{null}, \bot\}$ et $loc(p_j) \notin \{\texttt{null}, \bot\}$ et p_i est seul dans FMR et $loc(p_i)$ est sur une liste cyclique et $p_i \neq p_j$ et $\exists n \in$ **List**(FMR, p_j) tel que n est sur une liste cyclique	$\bigwedge_{x \in X_P} x' = x$	`Leak`
$FMR = (N, P, X, succ, loc, c)$ et $loc(p_i) \notin \{\texttt{null}, \bot\}$ et $loc(p_j) \notin \{\texttt{null}, \bot\}$ et p_i est seul dans FMR et $loc(p_i)$ est sur une liste cyclique et $p_i \neq p_j$ et $\nexists n \in$ **List**(FMR, p_j) tel que n est sur une liste cyclique	$y_j < \sum_{m \in \textbf{List}(FMR, p_j) \cap N} c(m) \wedge$ $\bigwedge_{x \in X_P} x' = x$	`Leak`
	$y_j \geq \sum_{m \in \textbf{List}(FMR, p_j) \cap N} c(m) \wedge$ $\bigwedge_{x \in X_P} x' = x$	`LeakSeg`
$FMR = (N, P, X, succ, loc, c)$ et $loc(p_i) \in \{\texttt{null}, \bot\}$ et $loc(p_j) \in \{\texttt{null}, \bot\}$	$\bigwedge_{x \in X_p} x' = x$	`Seg`
$FMR = (N, P, X, succ, loc, c)$ et $loc(p_i) \notin \{\texttt{null}, \bot\}$ et $loc(p_j) \in \{\texttt{null}, \bot\}$ et p_i n'est pas seul dans FMR et $loc(p_i)$ n'est pas sur une liste cyclique	$\bigvee_{p_k \in loc^{-1}(\{loc(p_i)\}) \setminus \{p_i\}} y_k = 0 \wedge$ $\bigwedge_{x \in X_p} x' = x$	`Seg`
	$\bigwedge_{p_k \in loc^{-1}(\{loc(p_i)\}) \setminus \{p_i\}} y_k \neq 0 \wedge$ $\bigwedge_{x \in X_p} x' = x$	`LeakSeg`
$FMR = (N, P, X, succ, loc, c)$ et $loc(p_i) \notin \{\texttt{null}, \bot\}$ et $loc(p_j) \in \{\texttt{null}, \bot\}$ et p_i n'est pas seul dans FMR et $loc(p_i)$ est sur une liste cyclique	$\bigwedge_{x \in X_p} x' = x$	`Seg`
$FMR = (N, P, X, succ, loc, c)$ et $loc(p_i) \notin \{\texttt{null}, \bot\}$ et $loc(p_j) \in \{\texttt{null}, \bot\}$ et p_i est seul dans FMR	$\bigwedge_{x \in X_p} x' = x$	`LeakSeg`

FIGURE 6.8 – Définition de **POST**$_2$ pour l'action $a = (p_i := p_j.\texttt{succ})$ (iii)

Hypothèses	**POST**$_2(a, FMR)$	
	ϕ	FMR'
$FMR = (N, P, X, succ, loc, c)$ et $loc(p_i) \in \{\texttt{null}, \bot\}$ et $z \in X \setminus X_{FMR}$	$y'_i = 0 \wedge$ $z' = 1 \wedge$ $\bigwedge_{x \in X_p \setminus \{y_i, z\}} x' = x$	$(N', P, X, succ', loc', c')$ avec $N' = N \uplus \{n\}$ $succ' = succ[n \mapsto \bot]$ $loc' = loc[p_i \mapsto n]$ et $c' = c[n \mapsto z]$
$FMR = (N, P, X, succ, loc, c)$ et $loc(p_i) \notin \{\texttt{null}, \bot\}$ et p_i n'est pas seul dans FMR et $loc(p_i)$ n'est pas sur une liste cyclique et $z \in X \setminus X_{FMR}$	$\bigvee_{p_k \in loc^{-1}(\{loc(p_i)\}) \setminus \{p_i\}} y_k = 0 \wedge$ $\wedge y'_1 = 0 \wedge z' = 1 \wedge$ $\bigwedge_{x \in X_p} x' = x$	$(N', P, X, succ', loc', c')$ avec $N' = N \uplus \{n\}$ $succ' = succ[n \mapsto \bot]$ $loc' = loc[p_i \mapsto n]$ et $c' = c[n \mapsto z]$
	$\bigwedge_{p_k \in loc^{-1}(\{loc(p_i)\}) \setminus \{p_i\}} y_k \neq 0 \wedge$ $\bigwedge_{x \in X_p} x' = x$	`Leak`
$FMR = (N, P, X, succ, loc, c)$ et $loc(p_i) \notin \{\texttt{null}, \bot\}$ et p_i n'est pas seul dans FMR et $loc(p_i)$ est sur une liste cyclique et $z \in X \setminus X_{FMR}$	$y'_i = 0 \wedge$ $z' = 1 \wedge$ $\bigwedge_{x \in X_p \setminus \{y_i, z\}} x' = x$	$(N', P, X, succ', loc', c')$ avec $N' = N \uplus \{n\}$ $succ' = succ[n \mapsto \bot]$ $loc' = loc[p_i \mapsto n]$ et $c' = c[n \mapsto z]$
$FMR = (N, P, X, succ, loc, c)$ et p_i est seul dans FMR et $loc(p_i) \notin \{\texttt{null}, \bot\}$	$\bigwedge_{x \in X_p} x' = x$	`Leak`

FIGURE 6.9 – Définition de **POST**$_2$ pour l'action $a = (p_i := \texttt{malloc})$

Exemple 6.13 *La figure 6.10 donne un exemple de l'application de la fonction* $\mathbf{POST}_2$ *avec comme arguments l'action de pointeurs* $p_1 := p_2.\texttt{succ}$ *et la forme mémoire restreinte* FMR*. Ainsi si nous considérons une valuation* $\mathbf{v} : X_P \to \mathbb{N}$ *telle que* $(FMR, \mathbf{v}_{FMR}, \mathbf{v}_P)$ *est une forme mémoire restreinte valuée, et* $GM = FM[\mathbf{v}_{FMR}, \mathbf{v}_P]$ *est le graphe mémoire correspondant, nous voyons que si la valeur du compteur* y_2 *est supérieure ou égale à la valeur de* x_1 *(en réalité elle ne peut au plus qu'être égale à la valeur de* x_1 *pour satisfaire la définition des formes mémoire restreintes valuées), alors l'action* $p_1 := p_2.\texttt{succ}$ *conduira à une erreur de segmentation, ceci car nous savons alors que dans le graphe mémoire* GM *la variable* p_2 *pointe sur le noeud* `null`*. En revanche, si la valeur associée au compteur* y_2 *est strictement plus petite que la valeur de* x_1*, l'action* $p_1 := p_2.\texttt{succ}$ *ne réalise pas d'erreur et on peut alors insérer le noeud* p_1*, pour se faire on le fait pointer sur le noeud pointé par* p_2 *et on met le compteur* y_1 *à jour de façon à ce que le noeud pointé par* p_1 *dans le graphe mémoire obtenu soit à une distance égale à* 1 *du noeud pointé par* p_2*.*

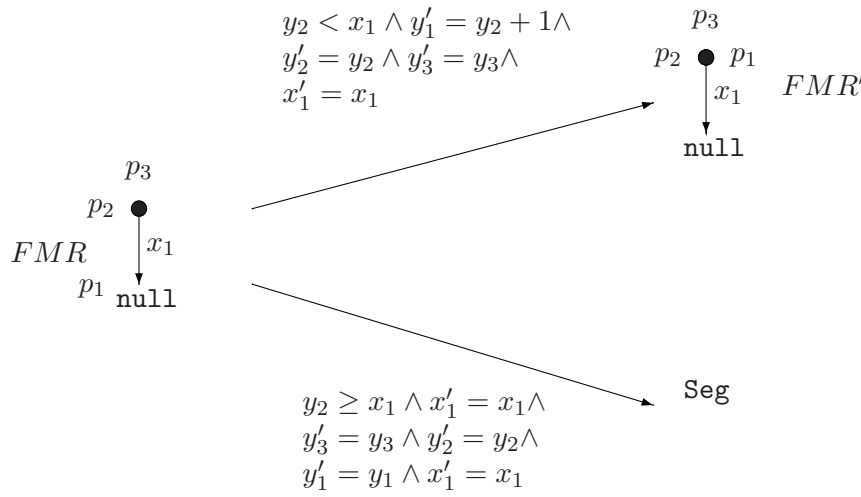


FIGURE 6.10 – La fonction $\mathbf{POST}_2$ appliquée à l'action $p_1 := p_2.\texttt{succ}$ et à FMR

De façon à montrer les propriétés des fonctions **Test** et $\mathbf{POST}_2$, nous introduisons deux relations $\sim_R$ et $\sim_R^{ac}$, toutes deux incluses dans $\mathcal{GM} \times (\mathcal{FMR} \times \mathbb{N}^{X_P})$. Ainsi pour tout $GM \in \mathcal{GM}$ et tout $(FMR, \mathbf{v}) \in \mathcal{FMR} \times \mathbb{N}^{X_P}$, nous avons :

– $GM \sim_R (FMR, \mathbf{v})$ si et seulement si

$$(FMR, \mathbf{v}_{FMR}, \mathbf{v}_P) \text{ est une forme mémoire restreinte valuée et } GM = FMR[\mathbf{v}_{FMR}, \mathbf{v}_P]$$

– $GM \sim_R^{ac} (FMR, \mathbf{v})$ si et seulement si

$$GM \sim_R (FMR, \mathbf{v}) \text{ et } FMR \text{ est acyclique}$$

Remarquons que si $GM \sim_R^{ac} (FMR, \mathbf{v})$ alors GM aussi est acyclique, ce point pouvant être vérifié grâce à la définition de $FMR[\mathbf{v}_{FMR}, \mathbf{v}_P]$. Nous étendons ensuite ces deux relations aux paires dans $\mathcal{GM}^{err} \times (\mathcal{FMR}^{err} \times \mathbb{N}^{X_P})$ de telle façon que pour toute valuation $\mathbf{v} \in \mathbb{N}^{X_P}$, $\texttt{Seg} \sim_R (\texttt{Seg}, \mathbf{v})$, $\texttt{Seg} \sim_R^{ac} (\texttt{Seg}, \mathbf{v})$, $\texttt{Leak} \sim_R (\texttt{Leak}, \mathbf{v})$,$\texttt{Leak} \sim_R^{ac} (\texttt{Leak}, \mathbf{v})$, $\texttt{LeakSeg} \sim_R (\texttt{LeakSeg}, \mathbf{v})$

et $\texttt{LeakSeg} \sim_R^{ac} (\texttt{LeakSeg}, \mathbf{v})$. Dans un premier temps, nous avons la propriété suivante concernant la fonction **Test** :

Lemme 6.14 *Soient $GM \in \mathcal{GM}$, $g \in \mathcal{G}$ une garde sur les pointeurs et $(FMR, \boldsymbol{v}) \in \mathcal{FMR} \times \mathbb{N}^{X_P}$ tels que $GM \sim_R (FMR, \boldsymbol{v})$. Si g n'utilise pas de test d'alias, nous avons alors $GM \models g$ si et seulement si $\boldsymbol{v} \models$ **Test**(g, FMR). De plus si $GM \sim_R^{ac} (FMR, \boldsymbol{v})$ alors $GM \models g$ si et seulement si $\boldsymbol{v} \models$ **Test**(g, FMR).*

Idée de la preuve : Nous considérons un graphe mémoire GM, et une paire $(FMR, \mathbf{v}) \in \mathcal{FMR} \times \mathbb{N}^{X_P}$ tel que $GM \sim_R^{ac} (FMR, \mathbf{v})$. Nous nous intéressons au cas où la garde g vaut $p_i == p_j$. Supposons que $GM \models g$. Nous posons $FMR = (N, P, X, succ, loc, c)$. Nous avons alors $GM = FMR[\mathbf{v}_{FMR}, \mathbf{v}_P]$. Alors nécessairement par définition de la concrétisation $FMR[\mathbf{v}_{FMR}, \mathbf{v}_P]$, nous ne pouvons pas avoir **Partage**$(FMR, p_i, p_j) = \emptyset$, par conséquent nous sommes dans le cas de la ligne 6 du tableau de la figure 6.3. Ainsi comme **Partage**$(FMR, p_i, p_j) \neq \emptyset$, l'inégalité $\mathbf{v}(y_i) \geq \sum_{n \in \textbf{List}(FMR, p_i) \setminus \textbf{Partage}(FMR, p_i, p_j)} \mathbf{v}(c(n))$ est vraie et il en va de même pour la condition similaire sur $\mathbf{v}(y_j)$ sinon cela signifierait que les deux pointeurs ne se trouvent pas sur la même liste chaînée. Finalement la dernière conjonction d'égalité est satisfaite car sinon les deux pointeurs pointeraient sur deux noeuds différents. Ce cas est bien illustré par l'exemple de la figure 6.2. La réciproque est de plus vraie pour les mêmes raisons. Nous ne traitons pas les autres cas comme par exemple pour $g = p_i == \texttt{null}$ qui se déduisent facilement en utilisant la définition de la fonction **Test** et de la relation d'équivalence $\sim_R$. □

Nous donnons également la propriété vérifiée par la fonction $\mathbf{POST}_2$ qui est similaire à celle donnée pour la fonction **POST** au chapitre précédent avec le lemme 5.3.

Lemme 6.15 *Soient $GM \in \mathcal{GM}$, $a \in \mathcal{A}$ une action de pointeurs telle que a n'est pas une mise à jour destructive et $(FMR, \boldsymbol{v}) \in \mathcal{FMR} \times \mathbb{N}^{X_P}$ tel que $GM \sim_R (FMR, \boldsymbol{v})$. Pour tout $GM' \in \mathcal{GM}^{err}$, nous avons $GM' = [\![a]\!]_P(GM)$ si et seulement si il existe $(FMR', \boldsymbol{v}') \in \mathcal{FMR}^{err} \times \mathbb{N}^{X_P}$ et $\phi \in$ **Presb**(X, X') tels que :*

- *$GM' \sim_R (FMR', \boldsymbol{v}')$ et*
- *$(\phi, FMR') \in \mathbf{POST}_2(a, FMR)$ et*
- *$(\boldsymbol{v}, \boldsymbol{v}') \models \phi$.*

Idée de la preuve : La preuve se fait par une étude de cas similaire à celle dont nous avons donné les principales lignes dans la preuve du lemme 5.3. Là encore le résultat énoncé est une conséquence directe de la construction de la fonction $\mathbf{POST}_2$ de de l'application des définitions de $[\![a]\!]_P(GM)$ et de $GM' \sim_R (FMR', \mathbf{v}')$.

Soient $GM \in \mathcal{GM}$, une action $a \in \mathcal{A}$ et $(FMR, \mathbf{v}) \in \mathcal{FMR} \times \mathbb{N}^{X_P}$ tel que $GM \sim_R (FM, \mathbf{v})$. Nous posons $FMR = (N, P, X, succ, loc, c)$. Par définition de $\sim_R$ nous avons $GM = FMR[\mathbf{v}_{FMR}, \mathbf{v}_R]$.

Si par exemple nous considérons l'action $p_i := p_j$ (avec $i \neq j$). Soit $GM' \in \mathcal{GM}^{err}$ tel que $GM' = [\![a]\!](GM)$. Soit $GM' \in \mathcal{GM}^{err}$.

- Supposons que $GM' \neq \texttt{Leak}$. Nous sommes alors dans le cas présenté à la première ligne du tableau de la figure 4.5.

– Si $loc(p_i) \in \{\texttt{null}, \perp\}$ alors p_i pointe sur `null` ou $\perp$ dans GM et la propriété est démontrée en utilisant la première ligne du tableau de la figure 6.4. On ne fait que déplacer le pointeur p_i vers le noeud pointé par p_j en passant de GM à GM' et si on considère la forme mémoire restreinte FMR' dans la ligne 1 du tableau de la figure 6.4, le pointeur p_i pointe sur le même noeud que p_j et de plus en prenant la valuation de compteurs $\mathbf{v}'$ telle que $(\mathbf{v}, \mathbf{v}') \models \phi$, on a bien que p_i pointe au même endroit que p_j dans $FMR'[\mathbf{v}'_{FMR'}, \mathbf{v}'_P]$ car la valeur du compteur y_i est égal à la valeur du compteur y_j.
– Si maintenant $loc(p_i) \notin \{\texttt{null}, \perp\}$, en accord avec la définition de $FMR[\mathbf{v}_{FMR}, \mathbf{v}_P]$, nous en déduisons que nécessairement p_i n'est pas seul dans FMR (sinon p_i serait seul aussi dans GM et l'action $p_i := p_j$ réaliserait une fuite mémoire, car si p_i est seul dans FMR, soit le noeud pointé par p_i dans GM est un noeud sans prédécesseur, soit il s'agit de l'unique noeud pointé par une variable dans une liste cyclique). Nous avons alors les deux cas suivants :
  1. si $loc(p_i)$ n'est pas sur une liste cyclique le déplacement du noeud p_i ne génère pas une fuite mémoire et cela est vrai si et seulement si il existe un autre pointeur p_k tel que $loc(p_k) = loc(p_i)$ et tel que $\mathbf{v}(y_k) = 0$ (c'est-à-dire que le noeud pointé par ce pointeur dans GM correspond au même noeud dans FMR). Ceci correspond au premier cas de la ligne 3 du tableau de la figure 6.4.
  2. Si $loc(p_i)$ est sur une liste cyclique, alors la ligne 4 du tableau de la figure 6.4 permet de traiter ce cas.

  Pour ces deux cas, nous constatons que dans FMR' le pointeur p_i pointe vers le même noeud que p_j et que de plus la formule ϕ assure que les valeurs des compteurs sont mis correctement à jour.
– Il est facile de voir que le cas où $GM' = \texttt{Leak}$ fonctionne également en utilisant les arguments déjà donnés.

Nous ne détaillons pas les autres cas qui se déduisent de façon identique en utilisant la définition de la fonction $\mathbf{POST}_2$ et de la concrétisation $FMR[\mathbf{v}_{FMR}, \mathbf{v}_P]$. □

Le lemme précédent est valable pour la relation $\sim_R$, mais remarquons que comme nous considérons des actions qui ne sont pas des mises à jour destructive, ces actions ne créent pas de cycle dans les graphes mémoire acycliques, par conséquent, nous en déduisons le résultat suivant :

Lemme 6.16 *Soient $GM \in \mathcal{GM}$, $a \in \mathcal{A}$ une action de pointeurs telle que a n'est pas une mise à jour destructive et $(FMR, \mathbf{v}) \in \mathcal{FMR} \times \mathbb{N}^{X_P}$ tel que $GM \sim_R^{ac} (FMR, \mathbf{v})$. Pour tout $GM' \in \mathcal{GM}^{err}$, nous avons $GM' = [\![a]\!]_P(GM)$ si et seulement si il existe $(FMR', \mathbf{v}') \in \mathcal{FMR}^{err} \times \mathbb{N}^{X_P}$ et $\phi \in \mathbf{Presb}(X, X')$ tels que :*

– *$GM' \sim_R^{ac} (FMR', \mathbf{v}')$, et,*
– *$(\phi, FMR') \in \mathbf{POST}_2(a, FMR)$, et,*
– *$(\mathbf{v}, \mathbf{v}') \models \phi$.*

Preuve : Comme nous l'avons dit il suffit de constater que lorsque GM est acyclique et que a n'est pas une mise à jour destructive alors $[\![a]\!]_P(GM)$ est encore acyclique et nous pouvons ensuite utiliser le résultat du lemme 6.15. □

Soit $S = (Q, P, E)$ un système à pointeurs sans mise à jour destructive. En utilisant les fonctions **Test** et $\mathbf{POST}_2$ que nous venons de définir, nous construisons le système à compteurs $S_R = (Q_R, X_P, E_R)$ avec :

– $Q_R = Q \times \mathcal{FMR}^{err}_{P,X}$,
– $E_R \subseteq (Q \times \mathcal{FMR}) \times \textbf{Presb}(X_P, X'_P) \times Q_R$ vérifie la condition suivante :

$$((q, FMR), \phi, (q', FMR')) \in E_R$$
si et seulement si
il existe $\phi_1 \in \textbf{Presb}(X_P)$ et $\phi_2 \in \textbf{Presb}(X_P, X'_P)$ et $(q, (g, a), q') \in E$ tels que :

1. $(g, FMR) \in \textbf{dom}(\textbf{Test})$ et,
2. $\phi = \phi_1 \wedge \phi_2$, et,
3. $\phi_1 \equiv \textbf{Test}(g, FMR)$, et,
4. $(\phi_2, FMR') \in \textbf{POST}_2(a, FMR)$.

Comme pour le système à compteurs S_C, étant donné que les formes mémoire restreintes sont des cas particuliers de formes mémoire, nous en déduisons que l'ensemble $\mathcal{FMR}^{err}_{P,X}$ est fini et par conséquent S_R a bien un nombre fini d'états de contrôle. Remarquons que là encore, il est possible de construire le système à pointeurs S_R en utilisant les définitions des fonctions **Test** et $\textbf{POST}_2$. Nous allons par la suite utiliser ce système à pointeurs S_R pour montrer des résultats de décidabilité sur les systèmes à pointeurs plats sans mise à jour destructive.

6.2.2.3 Propriétés de la traduction

Soient $S = (Q, P, E)$ un système à pointeurs sans mise à jour destructive et $S_R = (Q_R, X_P, E_R)$ le système à compteurs construit à partir de S en utilisant la méthode donnée dans la section précédente. Nous étudions ici les différentes propriétés du système à compteurs S_R.

Tout d'abord la définition des fonctions **Test** et $\textbf{POST}_2$ dont nous nous servons pour construire les formules étiquetant les transitions de S_R nous permettent d'établir la propriété suivante :

Lemme 6.17 *Le système à compteurs S_R est un système à compteurs linéaire à monoïde fini.*

Preuve : La preuve est similaire à la preuve du lemme 5.5 du chapitre précédent où nous avons montré que le système à compteurs S_C était un système à compteurs linéaire à monoïde fini.
Nous commençons par montrer que S_R est un système à compteurs linéaire. D'après la définition 1.15, il faut montrer que pour tout $((q, FMR), \phi, (q', FMR')) \in E_R$, la formule ϕ correspond à une fonction Presburger-linéaire. Or si $((q, FM), \phi, (q', FM')) \in E_R$, cela signifie qu'il existe une formule de Presburger $\phi_1 \in \textbf{Presb}(X_P)$ et une formule de Presburger ϕ_2 dans $\textbf{Presb}(X_P, X'_P)$ telles que $\phi = \phi_1 \wedge \phi_2$. Il nous faut donc vérifier que la formule ϕ_2 correspond à une fonction Presburger-linéaire. D'après la définition de E_R, il existe une action de pointeurs a (qui n'est pas une mise à jour destructive) telle que $(\phi_2, FMR') \in \textbf{POST}_2(a, FMR)$ et en regardant la définition de la fonction $\textbf{POST}_2$ fournie par les tableaux des figures 6.4 à 6.9, nous constatons que la formule ϕ_2 définit dans chaque cas une fonction Presburger-linéaire.
Nous montrons maintenant que le système à compteurs linéaire S_R est à monoïde fini. Nous utilisons un argument similaire à celui donné pour prouver que S_C est à monoïde fini. En effet, nous avions alors utilisé le fait que dans la première traduction, chaque matrice A appartenant à une fonction Presburger-linéaire présent dans S_C, était telle que chaque colonne de A ne comprenait que des 0 sauf un élément qui était égal à 1. Remarquons que pour cette nouvelle traduction, cet argument n'est plus valable, car nous avons parfois des actions de la forme $y'_i = y_j + 1 \wedge y'_j = y_j$ et la matrice

correspondante à deux éléments égaux à 1 dans la j-ème colonne. En revanche, pour toute fonction Presburger-linéaire $(\psi, A, \mathbf{b})$ tel qu'il existe $((q, FMR), (\psi, A, b), (q', FMR')) \in E_R$, nous remarquons que la matrice A est une matrice telle que chaque ligne de A ne contient que des 0 excepté un élément qui est égal à 1, or là encore le monoïde multiplicatif engendré par un tel ensemble de matrices est nécessairement fini. □

De plus par construction de S_R, nous avons également :

Lemme 6.18 *Si S est un système à pointeurs plat, alors S_R est un système à compteurs plat.*

Preuve : Nous rappelons que si $((q, FMR), \phi, (q', FMR')) \in E_R$ alors il existe une formule ϕ_2 dans **Presb**(X_P, X'_P) et $(q, (g, a), q') \in E$ tels que $(\phi_2, FMR') \in \mathbf{POST}_2(a, FMR)$. Or pour tout $FMR \in \mathcal{FMR}$ et pour tout $a \in \mathcal{A}_P$, si nous regardons la définition de la fonction $\mathbf{POST}_2$, nous nous apercevons qu'il existe au plus une paire $(\phi_2, FMR') \in \mathbf{POST}_2(a, FMR)$ telle que $FMR' \notin \{\mathtt{Seg}, \mathtt{Leak}, \mathtt{LeakSeg}\}$. Comme de plus dans S_R, il n'y a pas de transition partant d'états contenant une forme mémoire restreinte dans $\{\mathtt{Seg}, \mathtt{Leak}, \mathtt{LeakSeg}\}$ et comme S est plat, nous en déduisons que S_R est lui aussi plat. □

Nous verrons par la suite que le fait que le système à compteurs linéaire S_R est à monoïde fini nous servira pour réutiliser les résultats de décidabilité sur cette classe de système, en particulier le lemme 1.47 qui nous dit que la relation d'accessibilité des systèmes à compteurs linéaires plats et à monoïde fini est effectivement définissable dans Presburger, et le théorème 3.3 qui nous dit que le problème de model-checking symbolique généralisé est décidable pour les systèmes à compteurs linéaires plats et à monoïde fini. Nous voyons de plus apparaître l'importance d'avoir un système à compteurs plat pour pouvoir exploiter ces résultats.

Avant de montrer comment nous utilisons ces différents résultats, nous donnons le lien qui existe entre les systèmes de transitions $TS(S) = \langle Q \times \mathcal{GM}^{err}, E, \rightarrow \rangle$ et $TS(S_R) = \langle (Q \times \mathcal{FMR}^{err}) \times \mathbb{N}^{X_P}, E_R, \rightarrow_R \rangle$. Nous commençons par étendre les relations $\sim_R$ et $\sim_R^{ac}$ aux configurations de $TS(S)$ et de $TS(S_R)$. Ainsi nous considérons la relation $\sim_R \subseteq (Q \times \mathcal{GM}^{err}) \times \big((Q \times \mathcal{FMR}^{err}) \times \mathbb{N}^{X_P}\big)$ telle que $(q, GM) \sim_R ((q', FMR'), \mathbf{v})$ si et seulement si les conditions suivantes sont vérifiées :

- $q = q'$ et
- $GM \sim_R (FMR, \mathbf{v})$.

De la même façon, nous avons la relation $\sim_R^{ac} \subseteq (Q \times \mathcal{GM}^{err}) \times \big((Q \times \mathcal{FMR}^{err}) \times \mathbb{N}^{X_P}\big)$ telle que $(q, GM) \sim_R^{ac} ((q', FMR'), \mathbf{v})$ si et seulement si les conditions suivantes sont vérifiées :

- $q = q'$ et
- $GM \sim_R^{ac} (FMR, \mathbf{v})$.

Autrement dit pour tout $(q, GM) \in Q \times \mathcal{GM}^{err}$ et $((q', FMR), \mathbf{v}) \in \big((Q \times \mathcal{FMR}^{err}) \times \mathbb{N}^{X_P}\big)$, nous avons $(q, GM) \sim_R^{ac} ((q, FMR), \mathbf{v})$ si et seulement si $(q, GM) \sim_R ((q, FMR), \mathbf{v})$ et FMR est acyclique. En utilisant les propriétés vérifiés par la relation $\sim_R$ données par les lemme 6.14 et 6.15, nous obtenons le résultat suivant qui nous dit que cette relation est une bisimulation pour les systèmes à pointeurs sans test d'alias :

Lemme 6.19 *Nous considérons $(q_1, GM_1) \in Q \times \mathcal{GM}^{err}$ et $((q_1, FMR_1), \mathbf{v}_1) \in Q_R \times \mathbb{N}^{X_P}$ tels que $(q_1, GM_1) \sim_R ((q_1, FMR_1), \mathbf{v}_1)$. Si S est sans test d'alias, alors pour tout $(q_2, GM_2) \in Q \times \mathcal{GM}^{err}$, $(q_1, GM_1) \rightarrow (q_2, GM_2)$ si et seulement si il existe $((q_2, FMR_2), \mathbf{v}_2) \in Q_R \times \mathbb{N}^{X_P}$ tel que :*

– $((q_1, FMR_1), \mathbf{v}_1) \rightarrow_R ((q_2, FMR_2), \mathbf{v}_2)$ *et*
– $(q_2, GM_2) \sim_R ((q_2, FMR_2), \mathbf{v}_2)$.

Preuve : Supposons que $(q_1, GM_1) \xrightarrow{e} (q_2, GM_2)$ avec $e = (q_1, (g, a), q_2)$. Comme $GM_2 = [\![a]\!]_P(GM_1)$, en utilisant le résultat du lemme 6.15, nous en déduisons qu'il existe $(FMR_2, \mathbf{v}_2) \in \mathcal{FMR}^{err} \times \mathbb{N}^{X_P}$ et $\phi \in \mathbf{Presb}(X_P, X'_P)$ tels que $GM_2 \sim_R (FMR_2, \mathbf{v}_2)$ et aussi $(\phi, FMR_2) \in \mathbf{POST}_2(a, FM_1)$ et $(\mathbf{v}_1, \mathbf{v}_2) \models \phi$. Par définition de E_R comme g n'est pas un test d'alias, nous avons de plus que la transition $e' = ((q_1, FMR_1), \mathbf{Test}(g, FMR_1) \wedge \phi, (q_2, FMR_2))$ appartient à E_R. Finalement, encore une fois comme g n'utilise pas de test d'alias et que $GM_1 \models g$ et $GM_1 \sim_R (FMR_1, \mathbf{v}_1)$, nous avons d'après le lemme 6.14, $\mathbf{v}_1 \models \mathbf{Test}(g, FMR_1)$. Par conséquent, nous en déduisons que $(\mathbf{v}_1, \mathbf{v}_2) \models \mathbf{Test}(g, FMR) \wedge \phi$ et donc nous avons bien $((q_1, FMR_1), \mathbf{v}_1) \rightarrow_R ((q_2, FMR_2), \mathbf{v}_2)$. Comme de plus $GM_2 \sim_R (FMR_2, \mathbf{v}_2)$, nous avons également $(q_2, GM_2) \sim_R ((q_2, FMR_2), \mathbf{v}_2)$. La réciproque se prouve de la même façon. □

Nous avons le lemme équivalent en considérant la relation $\sim_R^{ac}$, mais cette fois nous pouvons prendre en compte des systèmes à pointeurs utilisant des tests d'alias.

Lemme 6.20 *Nous considérons $(q_1, GM_1) \in Q \times \mathcal{GM}^{err}$ et $((q_1, FMR_1), \mathbf{v}_1) \in Q_R \times \mathbb{N}^{X_P}$ tels que $(q_1, GM_1) \sim_R^{ac} ((q_1, FMR_1), \mathbf{v}_1)$. Alors pour tout $(q_2, GM_2) \in Q \times \mathcal{GM}^{err}$, $(q_1, GM_1) \rightarrow (q_2, GM_2)$ si et seulement si il existe $((q_2, FMR_2), \mathbf{v}_2) \in Q_R \times \mathbb{N}^{X_P}$ tel que :*

– $((q_1, FMR_1), \mathbf{v}_1) \rightarrow_R ((q_2, FMR_2), \mathbf{v}_2)$ *et*
– $(q_2, GM_2) \sim_R^{ac} ((q_2, FMR_2), \mathbf{v}_2)$.

Preuve : Supposons que $(q_1, GM_1) \xrightarrow{e} (q_2, GM_2)$ avec $e = (q_1, (g, a), q_2)$. Comme $GM_2 = [\![a]\!]_P(GM_1)$, en utilisant le résultat du lemme 6.16, nous en déduisons qu'il existe $(FMR_2, \mathbf{v}_2) \in \mathcal{FM}^{err} \times \mathbb{N}^{X_P}$ et $\phi \in \mathbf{Presb}(X_P, X'_P)$ tels que $GM_2 \sim_R^{ac} (FMR_2, \mathbf{v}_2)$ et aussi $(\phi, FMR_2) \in \mathbf{POST}_2(a, FM_1)$ et $(\mathbf{v}_1, \mathbf{v}_2) \models \phi$. Par définition de E_R comme FMR_1 est acyclique (par définition de $\sim_R^{ac}$), nous avons de plus que la transition $e' = ((q_1, FMR_1), \mathbf{Test}(g, FMR_1) \wedge \phi, (q_2, FMR_2))$ appartient à E_R. Finalement, comme $GM_1 \models g$ et $GM_1 \sim_R^{ac} (FMR_1, \mathbf{v}_1)$, nous avons d'après le lemme 6.14, $\mathbf{v}_1 \models \mathbf{Test}(g, FMR_1)$. Par conséquent, nous en déduisons que $(\mathbf{v}_1, \mathbf{v}_2) \models \mathbf{Test}(g, FMR) \wedge \phi$ et donc nous avons bien $((q_1, FMR_1), \mathbf{v}_1) \rightarrow_R ((q_2, FMR_2), \mathbf{v}_2)$. Comme de plus $GM_2 \sim_R^{ac} (FMR_2, \mathbf{v}_2)$, nous avons également $(q_2, GM_2) \sim_R^{ac} ((q_2, FMR_2), \mathbf{v}_2)$. La réciproque se prouve de la même façon. □

Ce lemme nous permet alors d'établir une propriété sur les ensembles d'accessibilité de S et de S_R, à savoir :

Lemme 6.21 *Soient $c_0 \in Q \times \mathcal{GM}^{err}$ une configuration initiale de $TS(S)$ et $c'_0 \in Q_R \times \mathbb{N}^{X_P}$ tels que $c_0 \sim_R c'_0$. Nous avons :*
– *Si S est sans test d'alias, alors :*

$$\mathbf{Reach}(S, c_0) = \{c \in Q \times \mathcal{GM}^{err} \mid \exists c' \in \mathbf{Reach}(S_R, c'_0) \text{ tel que } c \sim_R c'\}$$

– *Si $c_0 \sim_R^{ac} c'_0$, alors :*

$$\mathbf{Reach}(S, c_0) = \{c \in Q \times \mathcal{GM}^{err} \mid \exists c' \in \mathbf{Reach}(S_R, c'_0) \text{ tel que } c \sim_R^{ac} c'\}$$

Preuve : Supposons que $c_0 \sim_R c'_0$. Nous nous attachons à montrer le premier point. Nous supposons que S est sans test d'alias. Soit $c \in \mathbf{Reach}(S, c_0)$. Montrons qu'il existe $c' \in \mathbf{Reach}(S_R, c'_0)$ tel que $c \sim_R c'$. Comme $c \in \mathbf{Reach}(S, c_0)$, il existe une exécution dans $TS(S)$ de la forme $c_0 \rightarrow c_1 \ldots \rightarrow c_f$ avec $f \in \mathbb{N}$ et $c_f = c$. Nous raisonnons par induction sur la taille de cette exécution, c'est-à-dire sur f, en utilisant le lemme 6.19. Si $f = 0$, alors nous avons bien $c_0 \sim_R c'_0$ (par hypothèse) et $c'_0 \in \mathbf{Reach}(S_R, c'_0)$. Supposons que $f > 0$ et que pour tout $j \in [0..f-1]$, il existe $c'_j \in \mathbf{Reach}(S_C, c'_0)$ tel que $c_j \sim c'_j$. En particulier, nous avons que $c_{f-1} \sim_R c'_{f-1}$ et comme $c_{f-1} \rightarrow c_f$, d'après le lemme 6.19, il existe une configuration c' de $TS(S_C)$ telle que $c'_{f-1} \rightarrow_R c'$ et $c_f \sim c'$. Comme $c'_{f-1} \in \mathbf{Reach}(S_C, c'_0)$, nous avons bien que $c' \in \mathbf{Reach}(S_C, c'_0)$.
La réciproque se prouve par une induction similaire. Le deuxième point se prouve aussi de la même façon en utilisant cette fois-ci le résultat du lemme 6.20 sur la relation $\sim_R^{ac}$. □

Nous verrons par la suite comment nous pouvons utiliser cette caractérisation sur l'ensemble d'accessibilité du système à pointeurs S muni d'une configuration initiale c_0, mais avant nous devons prouver quelques propriétés concernant les formes mémoire symboliques et les formes mémoire restreintes valuées.

6.2.2.4 Encoder des formes mémoire symboliques avec des formes mémoire restreintes

Nous allons montrer dans cette section, comment encoder des formes mémoire symboliques avec des formes mémoire restreintes. Pour rappel, une forme mémoire symbolique est une paire (FM, ϕ) telle que FM est une forme mémoire et ϕ une formule de Presburger dans $\mathbf{Presb}(X_{FM})$ vérifiant $\phi \Rightarrow \bigwedge_{x \in X} x > 0$. Une forme mémoire symbolique permet de définir un ensemble de graphes mémoire $[\![(FM, \phi)]\!] = \{FM(\mathbf{v}) \in \mathcal{GM} \mid \mathbf{v} \models \phi\}$. Comme nous l'avons vu, une forme mémoire restreinte valuée $(FMR, \mathbf{v}, \mathbf{u})$ permet aussi de définir un graphe mémoire que nous notons $FMR[\mathbf{v}, \mathbf{u}]$. Nous allons maintenant faire un lien entre les graphes mémoire encodés par une forme mémoire symbolique et les graphes mémoire que l'on peut encoder en utilisant des formes mémoire restreintes.

Soient (FM, ϕ) une forme mémoire symbolique avec $FM \in \mathcal{FM}$ et $\phi \in \mathbf{Presb}(X_{FM})$, et $FMR \in \mathcal{FMR}$ une forme mémoire restreinte. Nous considérons alors l'ensemble des valuations possibles $\mathbf{v} : X_P \mapsto \mathbb{N}$ telles que $(FMR, \mathbf{v}_{FMR}, \mathbf{v}_P)$ est une forme mémoire retreinte valuée et telles que le graphe mémoire correspondant appartient à $[\![(FM, \phi)]\!]$. Nous appelons $T_{FMR}(FM, \phi)$ cet ensemble. Formellement, nous avons :

$$T_{FMR}(FM, \phi) =$$
$$\{\mathbf{v} \in \mathbb{N}^{X_P} \mid (FMR, \mathbf{v}_{FMR}, \mathbf{v}_P) \text{ est une forme mémoire restreinte valuée}$$
$$\text{et } FMR[\mathbf{v}_{FMR}, \mathbf{v}_P] \in [\![(FM, \phi)]\!]\}$$

Nous allons montrer qu'il existe une formule du fragment existentiel de la logique du premier ordre sur les entiers avec addition et divisibilité telle qu'une valuation vérifiera cette formule si et seulement si elle appartient à l'ensemble $T_{FMR}(FM, \phi)$. Tout d'abord, nous définissons le fragment existentiel de la logique du premier ordre sur les entiers avec addition et divisibilité. Nous notons $(\mathbb{N}, +, |, 0, 1)^{\exists}$ ce fragment logique. L'ensemble des formules de $(\mathbb{N}, +, |, 0, 1)^{\exists}$ peut être décrit par la grammaire suivante, où t représente un terme, ϕ une formule sans quantificateur de $(\mathbb{N}, +, |, 0, 1)^{\exists}$ et Ψ une formule de $(\mathbb{N}, +, |, 0, 1)^{\exists}$:

$$\begin{array}{rcl} t & ::= & 0 \mid 1 \mid x \mid t + t \\ \phi & ::= & t = t \mid t | t \mid \neg\phi \mid \phi \vee \phi \\ \Psi & ::= & \phi \mid \exists y.\Psi \end{array}$$

où x et y appartiennent à un ensemble de variables et la formule $t'|t$ signifie que le terme t' est un diviseur entier de t. Nous ne détaillons pas la relation de satisfiabilité de cette logique qui est similaire à celle définie pour l'arithmétique de Presburger excepté pour les formules de la forme $t'|t$. Ainsi nous dirons que pour une fonction $f : \mathbf{var}(\Psi) \rightarrow \mathbb{N}$, nous avons :

– $f \models t'|t$ si et seulement si $\mathbf{App}_f(t') \neq 0$ et $\mathbf{App}_f(t')$ est un diviseur entier de $\mathbf{App}_f(t)$.

Comme pour les formules de l'arithmétique de Presburger, nous interprétons les formules de la logique $(\mathbb{N}, +, |, 0, 1)^{\exists}$ sur l'ensemble $\mathbb{N}$ des entiers naturels. De plus, étant donné un ensemble X de variables, nous notons $\mathcal{L}_{|}^{\exists}(X)$ l'ensemble des formules de $(\mathbb{N}, +, |, 0, 1)^{\exists}$ qui ont leurs variables libres dans X. Une propriété intéressante de ce fragment logique réside dans le fait que, tout comme pour l'arithmétique de Presburger, le problème de satisfiabilité d'une formule de $(\mathbb{N}, +, |, 0, 1)^{\exists}$ est décidable. La décidabilité de ce problème est prouvé dans [Lip78].

Avant de montrer comment définir l'ensemble $T_{FMR}(FM, phi)$ grâce à une formule de $(\mathbb{N}, +, |, 0, 1)^{\exists}$, nous introduisons quelques notations. Soient FM une forme mémoire symbolique et FMR une forme mémoire restreinte. Nous dirons que FMR est compatible avec FM, noté $FMR \succ FM$, si et seulement si $T_{FMR}(FM, \bigwedge_{x \in X_{FM}} x > 0) \neq \emptyset$.

Nous considérons maintenant une forme mémoire $FM = (N, P, X, succ, loc, c)$ et une forme mémoire restreinte $FMR = (N', P, X, succ', loc', c')$. Nous notons N_R (respectivement N'_R) l'ensemble des noeuds de FM (respectivement de FMR) n'ayant pas de prédécesseur. C'est-à-dire que $N_R = \{n \in N \mid succ^{-1}(\{n\}) = \emptyset\}$ et $N'_R = \{n \in N' \mid succ'^{-1}(\{n\}) = \emptyset\}$. De la même façon, nous notons N_2 (respectivement N'_2) l'ensemble des noeuds de FM (respectivement de FMR) ayant au moins deux prédécesseurs. Nous avons donc $N_2 = \{n \in N \mid |succ^{-1}(\{n\})| \geq 2\}$ et $N'_2 = \{n \in N' \mid |succ'^{-1}(\{n\})| \geq 2\}$. Pour finir, nous notons N_C (respectivement N'_C) l'ensemble des noeuds de FM (respectivement de FMR) qui appartiennent à une liste cyclique et qui ne sont pas accessibles par un noeud sans prédécesseur. Par conséquent $N_C = \{n \in N \mid n \in \mathbf{List}(FM, succ(n))$ et $\nexists m \in N_R$ tel que $n \in \mathbf{List}(FM, m)\}$ et de la même façon $N'_C = \{n \in N' \mid n \in \mathbf{List}(FMR, succ(n))$ et $\nexists m \in N'_R$ tel que $n \in \mathbf{List}(FMR, m)\}$. Nous disons alors qu'une fonction $g : N' \rightarrow N$ est une fonction de compatibilité entre FMR et FM si et seulement si g est une fonction totale injective telle que :

– $g(N'_R) = N_R$, $g(N'_2) = N_2$ et $g(N'_C) \subseteq N_C$, et
– pour tout noeud $n, m \in N'$, $m \in \mathbf{List}(FMR, n)$ si et seulement si $g(m) \in \mathbf{List}(FM, g(n))$, et,
– pour tout noeud $n \in N'$, $\texttt{null} \in \mathbf{List}(FMR, n)$ si et seulement si $\texttt{null} \in \mathbf{List}(FM, g(n))$, et,
– pour tout noeud $n \in N'$, $\perp \in \mathbf{List}(FMR, n)$ si et seulement si $\perp \in \mathbf{List}(FM, g(n))$, et,
– pour toute variable $p \in P$, $loc(p) \in \mathbf{List}(FM, g(loc'(p)))$.

Intuitivement g est une sorte d'isomorphisme entre le graphe de FMR et celui de FM modulo les noeuds qui sont enlevés en passant de FMR à FM lorsque les variables sont déplacés de façon à satisfaire les conditions données par la définition des formes mémoire restreintes pour lesquelles chaque variable est soit sur un noeud sans prédécesseur soit sur un noeud qui est l'unique noeud d'une liste cyclique, non accessible par d'autres noeuds. Nous avons alors le lemme suivant :

Lemme 6.22 *$FMR \succ FM$ si et seulement si il existe une fonction de compatibilité entre FMR et FM.*

Preuve : Nous rappelons que $FM = (N, P, X, succ, loc, c)$ et $FMR' = (N', P, X, succ', loc', c')$. Nous reprenons les notations introduites pour définir les fonctions de compatibilité en ce qui concerne les ensembles N_R, N'_R,N_2, N'_2, N_C et N'_C.
Supposons que $FMR \succ FM$, nous avons donc qu'il existe une valuation $\mathbf{v} : X_{FM} \to \mathbb{N}^*$ et une valuation $\mathbf{v}' : X_P \to \mathbb{N}$ tels que $(FMR, \mathbf{v}'_{FMR}, \mathbf{v}'_P)$ est une forme mémoire restreinte et telle que $FM(\mathbf{v}) = FMR[\mathbf{v}'_{FMR}, \mathbf{v}'_P]$. Nous allons maintenant construire une fonction de compatibilité entre FMR et FM en utilisant les définitions des concrétisations $FM(\mathbf{v})$ et $FMR[\mathbf{v}'_{FMR}, \mathbf{v}'_P]$.

- Tout d'abord remarquons que pour tout pointeur $p \in P$, si $loc(p) \in N_R$ alors nous avons dans FMR, $loc'(p) \in N'_R$ ceci car au moment de la construction de $FM(\mathbf{v})$ les variables ne sont pas déplacées mais des noeuds sont insérés (par conséquent comme dans $FM(\mathbf{v})$ le pointeur p pointe aussi vers un noeud sans prédécesseur). Ainsi pour tout noeud $n \in N_R$, nous pouvons associer un unique noeud n' dans N'_R tel que $loc^{-1}(\{n\}) \subseteq loc'^{-1}(\{n'\})$ et de cette façon nous définissons $g(n') = n$. Remarquons que nous avons alors bien $g(N'_R) = N_R$.
- Nous considérons maintenant un noeud n dans N_2. Comme $FM(\mathbf{v}) = FMR[\mathbf{v}'_{FMR}, \mathbf{v}'_P]$, nous avons nécessairement $|N_2| = |N'_2|$ et de plus pour chaque noeud n dans N_2, par définition des formes mémoire, il existe une variable $p \in P$ telle que $loc(p) \in N_R$ et $n \in \mathbf{List}(FM, loc(p))$. Comme au moment des concrétisations $FM(\mathbf{v})$ et $FMR[\mathbf{v}'_{FMR}, \mathbf{v}'_P]$ nous n'insérons pas de nouveaux noeuds avec deux prédécesseurs, nous en déduisons que pour chaque noeud $n \in N_2$, nous pouvons associer un noeud n' dans N'_2 de telle sorte que si m' est un noeud dans N'_R tel que $n' \in \mathbf{List}(FMR, m')$ alors $n \in \mathbf{List}(FM, g(m'))$. Remarquons que nous avons alors bien $g(N'_2) = N_2$.
- Nous considérons finalement les noeuds n dans N_C. Nous pouvons partitionner N_C en un ensemble $N_C^1 \uplus \ldots \uplus N_C^r$ de telle sorte que pour tout $i \in [1..r]$, pour tous noeuds $n, m \in N_C^i$, $n \in \mathbf{List}(FM, m)$ (et par conséquent nous avons aussi $m \in \mathbf{List}(FM, n)$) et de plus, pour tout $j \in [1..r]$ tel que $j \neq i$, pour tout noeud $m \in N_C^j$ et $n \in N_C^i$ nous avons $m \notin \mathbf{List}(FM, n)$. Ainsi la partition $N_C^1 \uplus \ldots \uplus N_C^r$ regroupe ensemble les noeuds appartenant à une même liste cyclique dans FM. Remarquons que par définition de $FM(\mathbf{v})$ et de $FMR[\mathbf{v}'_{FMR}, \mathbf{v}'_P]$, nous avons $r = |N'_C|$ (car il y a autant de listes cycliques non accessibles par un noeud sans prédécesseur dans FM que dans FMR et que dans FMR ces listes cycliques ne comportent qu'un unique noeud). Ainsi nous pouvons associer à chaque noeud $n' \in N_C$ un entier $i \in [1..r]$ différent de telle sorte que $loc'^{-1}(\{n'\}) = loc^{-1}(N_C^i)$, et finalement la fonction g associe à n' un des noeuds de N_C^i.

Comme, par définition des formes mémoire restreintes, $N' = N_R \uplus N'_2 \uplus N'_C$, nous avons bien défini une fonction totale et injective de N' vers N. De plus, on peut vérifier que par construction cette fonction g est bien une fonction de compatibilité entre FMR et FM.
Nous supposons maintenant qu'il existe une fonction de compatibilité g entre FMR et FM. Soit $\mathbf{v} : X_{FM} \to \mathbb{N}^*$ une valuation sur les compteurs de FM, nous allons montrer que nous pouvons construire une valuation $\mathbf{v}' : X_P \to \mathbb{N}$ telle que $(FMR, \mathbf{v}'_{FMR}, \mathbf{v}'_P)$ soit une forme mémoire restreinte valuée vérifiant $FMR[\mathbf{v}'_{FMR}, \mathbf{v}'_P] = FM(\mathbf{v})$. Ainsi pour tout noeud $m \in N'$, nous définissons

$$\mathbf{v}'(c'(m)) = \mathbf{v}(c(g(m)) + \sum_{n \in \mathbf{Entre}(FM, g(m), g(succ'(m))) \cap N} c(n)$$

Et pour chaque variable de pointeur $p_i \in P$ tel que $m = loc'(p_i)$, si $m \in \{\texttt{null}, \bot\}$ alors $\mathbf{v}'(y_i) = 0$, sinon nous posons :

$$\mathbf{v}'(y_i) = \mathbf{v}(c(g(m))) + \sum_{n \in \mathbf{Entre}(FM, g(m), loc(p_i)) \cap N} c(n)$$

Et finalement, pour tout $x \in X_P \setminus (X_{FMR} \cup Y_P)$, nous posons $\mathbf{v}'(x) = 0$. En utilisant les définitions, on peut facilement montrer que $(FMR, \mathbf{v}'_{FMR}, \mathbf{v}'_P)$ est une forme mémoire valuée telle que $FM(\mathbf{v}) = FMR[\mathbf{v}'_{FMR}, \mathbf{v}'_P]$. □

FIGURE 6.11 – Une forme mémoire restreinte compatible avec une forme mémoire $(FMR \succ FM)$

Exemple 6.23 *Sur la figure 6.11 nous avons représenté la forme mémoire retreinte FMR compatible avec la forme mémoire FM. Si nous notons $FMR = (\{n'_1, n'_2\}, P, X, succ', loc', c')$ de telle sorte que $loc'(p_1) = n'_1$ et $loc'(p_3) = n'_2$ et si nous $FM = (\{n_1, n_2, n_3, n_4\}, P, X, succ, loc, c)$ de telle façon que $loc(p_1) = n_1$, $loc(p_2) = n_2$, $loc(p_3) = n_3$ et $loc(p_4) = n_4$ alors la fonction g définie de la façon suivante : $g(n'_1) = n_1$ et $g(n'_2) = n_3$ est une fonction de compatibilité entre FMR et FM.*

Comme il est possible de vérifier si une fonction totale et injective de N' dans N est une fonction de compatibilité entre FMR et FM et comme le nombre de telles fonctions est fini, nous en déduisons que :

Lemme 6.24 *Décider si une forme mémoire restreinte est compatible avec une forme mémoire est un problème décidable.*

Nous avons maintenant les outils nécessaires pour prouver le lemme suivant, qui nous montre comment encoder une forme mémoire symbolique grâce à une forme mémoire restreinte :

Lemme 6.25 *Soit (FM, ϕ) une forme mémoire symbolique et FMR une forme mémoire restreinte. Alors il existe une formule effectivement calculable $\Psi_{FMR}(FM, \phi)$ de $\mathcal{L}^{\exists}_{|}(X_P)$ telle que pour tout $\mathbf{v} : X_P \to \mathbb{N}$, nous avons $\mathbf{v} \in T_{FMR}(FM, \phi)$ si et seulement si $\mathbf{v} \models \Psi_{FMR}(FM, \phi)$. Et si FM est acyclique alors $\Psi_{FMR}(FM, \phi)$ est une formule de l'arithmétique de Presburger.*

Preuve : Nous supposons que $FM = (N, P, X, succ, loc, c)$ et $FMR = (N', P, X, succ', loc', c')$. Nous rappelons que comme (FM, ϕ) est une forme mémoire symbolique, $\phi \in \mathbf{Presb}(X_{FM})$. Nous posons $X_{FM} = \{x_1, \ldots, x_k\}$. Nous lui associons l'ensemble $\hat{X}_{FM} = \{\hat{x}_1, \ldots, \hat{x}_k\}$ et nous construisons la fonction $\hat{c}$ telle que pour tout $n \in N$, $\hat{c}(n) = \hat{x}_i$ si et seulement si $c(n) = x_i$. De plus, nous notons $\hat{\phi}$ la formule obtenue à partir de ϕ en renommant chaque x_i par $\hat{x}_i$. Nous construisons la formule logique $\Psi_{FMR}(FM, \phi)$ sur les variables X_P en suivant les étapes suivantes :

– Si $FMR \not\succ FM$ alors $\Psi_{FMR}(FM, \phi) = false$
– Si $FMR \succ FM$ alors soit g une fonction de compatibilité entre FMR et FM (il en existe nécessairement une d'après le lemme 6.22) :

1. Pour chaque noeud $m \in N'$, nous construisons la formule ψ_m qui permet d'assurer que les longueurs des segments de listes dans les graphes mémoire correspondent :

$$\psi_m = \big(c'(m) = \hat{c}(g(m)) + \sum_{n \in \textbf{Entre}(FM, g(m), g(succ'(m))) \cap N} \hat{c}(n)\big)$$

2. Soit $N_{Cl} \subseteq N$ (resp. $N'_{Cl} \subseteq N'$) l'ensemble des noeuds appartenant à une liste cyclique dans FM (respectivement dans FMR). Pour chaque pointeur $p_i \in P$, nous définissons une formule ψ_i pour garantir la position des pointeurs dans les graphes mémoire :

   (a) Si $loc(p_i) \in \{\texttt{null}, \perp\}$ alors :
   $$\psi_i = (y_i = 0)$$

   (b) Si $loc(p_i) \notin N_{Cl}$ et $m = loc'(p_i)$ alors :
   $$\psi_i = \big(y_i = \hat{c}(g(m)) + \sum_{n \in \textbf{Entre}(FM, g(m), loc(p_i)) \cap N} \hat{c}(n)\big)$$

   (c) Si $loc(p_i) \in N_{Cl}$ et $loc'(p_i) \in N'_{Cl}$ (dans ce cas la lise cyclique à laquelle appartient $loc'(p_i)$ dans FMR n'est pas accessible par un noeud sans prédecesseur) , nous notons alors $L = \sum_{n \in \textbf{List}(FM, loc(p_i))} \hat{c}(n)$, intuitivement L encode la taille de la liste cyclique et nous avons alors :
   $$\psi_i = \big(\bigwedge_{\{p_j | loc(p_j) = loc(p_i)\}} (y_i) mod(L) = (y_j) mod(L) \wedge \bigwedge_{\{p_j | loc(p_j) \in \textbf{List}(FM, loc(p_i)) \setminus \{loc(p_i)\}\}} (y_j) mod(L) = (y_i + \hat{c}(loc(p_i)) + \sum_{n \in \textbf{Entre}(FM, loc(p_i), loc(p_j))} \hat{c}(n)) mod(L)\big)$$

   (d) Finalement, si $loc(p_i) \in N_{Cl}$ et $loc'(p_i) \notin N'_{Cl}$ (dans ce cas la lise cyclique à laquelle appartient $loc'(p_i)$ dans FMR est accessible par un noeud sans prédecesseur, en particulier le noeud pointé par p_i), nous notons alors $L = \sum_{n \in \textbf{List}(FM, loc(p_i))} \hat{c}(n)$, intuitivement L encode la taille de la liste cyclique, et $S = \sum_{n \in \textbf{List}(FM, g(loc'(p_i)) \setminus N_{Cl}} \hat{c}(n)$, c'est-à-dire la taille du segment partant de $g(loc'(p_i))$ et finissant sur la liste cyclique, nous avons alors :
   $$\psi_i = \big(y_i \geq S \wedge \hat{c}(g(loc'(p_i)) + \sum_{n \in \textbf{Entre}(FM, g(loc'(p_i)), loc(p_i))} \hat{c}(n) = S + (y_i - S) mod(L)\big)$$

3. Nous avons finalement :

$$\Psi_{FMR}(FM, \phi) = \exists \hat{x}_1 \ldots \exists \hat{x}_k . \hat{\phi} \wedge \bigwedge_{m \in N'} \psi_m \wedge \bigwedge_{p_i \in P} \psi_i$$

Tout d'abord remarquons que $\Psi_{FMR}(FM, \phi)$ est bien une formule de $\mathcal{L}^{\exists}_{|}(X_P)$, ceci car même si il peut y avoir des quantificateurs universels dans la formule de Presburger $\hat{\phi}$, ceux-ci peuvent être éliminés car l'élimination des quantificateurs universels est possible pour une formule de Presburger

(ceci est une propriété classique de l'arithmétique de Presburger que l'on peut redémontrer en utilisant le fait que les ensembles Presburger-définissables sont exactement les ensembles semi-linéaires [GS66]). De plus, dans les formules que nous construisons, nous avons des sous-formules de la forme $(t)mod(L) = (t')mod(L)$ qui peuvent se réécrire de la façon suivante : $\exists z.(t = z + t' \vee t' = t + z) \wedge L|z$. Finalement, nous constatons que si FM ne contient pas de liste cyclique, la formule construite est une formule de l'arithmétique de Presburger.
Finalement en utilisant les définitions de $FMR[\mathbf{v}_{FMR}, \mathbf{v}_P]$ et de $[\![(FM, \phi)]\!]$ et des fonctions de compatibilité, nous avons bien que pour tout $\mathbf{v} \in \mathbb{N}^{X_P}$, $\mathbf{v} \models \Psi_{FMR}(FM, \phi)$ si et seulement si $(FMR, \mathbf{v}_{FMR}, \mathbf{v}_P)$ est une forme mémoire restreinte valuée telle que $FMR[\mathbf{v}_{FMR}, \mathbf{v}_P] \in [\![(FM, \phi)]\!]$. □

$$\Psi_{FMR}(FM, \phi) = \exists \hat{x}_1.\exists \hat{x}_2.\exists \hat{x}_3.\exists \hat{x}_4.\hat{\phi} \wedge$$
$$x_1 = \hat{x}_1 + \hat{x}_2 \wedge x_2 = \hat{x}_3 + \hat{x}_4 \wedge$$
$$y_1 = 0 \wedge y_2 = \hat{x}_1 \wedge$$
$$(y_4)mod(\hat{x}_3 + \hat{x}_4) = (y_3 + \hat{x}_3)mod(\hat{x}_3 + \hat{x}_4) \wedge$$
$$(y_3)mod(\hat{x}_3 + \hat{x}_4) = (y_4 + \hat{x}_4)mod(\hat{x}_3 + \hat{x}_4)$$

FIGURE 6.12 – Exemple du calcul de $\Psi_{FMR}(FM, \phi)$

Nous allons voir par la suite que le résultat de ce dernier lemme est très important pour établir les résultats de décidabilité.

6.2.3 Systèmes à pointeurs plats sans mise à jour destructive

Grâce aux différents outils que nous avons mis en place dans les sections précédentes nous allons maintenant pouvoir établir différents résultats de décidabilité. Nous montrerons ensuite que les hypothèses que nous posons pour obtenir la décidabilité peuvent difficilement être relâchées. Les résultats que nous donnons ici concernent les systèmes à pointeurs plats sans mise à jour destructive. Notons que dans [BI07] les auteurs ont aussi étudié la décidabilité de certains problèmes sur des programmes plats. Le principal problème traité dans [BI07] consiste à savoir si, étant donnés l'équivalent d'une forme mémoire et un programme muni d'assertions à vérifier à certaines lignes du programme, dans toutes les exécutions partant de graphes mémoires pouvant être encodé grâce à la forme mémoire donnée, les différentes assertions sont toujours respectées. Notons que les assertions insérées dans le programme sont des combinaisons booléennes de propositions de la forme $p == \texttt{null}$ ou $p == p'$ dans lesquelles p et p' sont des variables de pointeurs. Ils montrent ainsi que dans le cas de programmes plats sans mise à jour destructive et avec une forme mémoire initiale contenant au plus une liste cyclique, le problème considéré est décidable, mais que lorsque la forme mémoire initiale contient plus d'une seule liste cyclique le problème devient indécidable. Ainsi, le problème étudié dans

[BI07] est un cas particulier de problème d'accessibilité symbolique généralisé, par conséquent nous en déduisons que dans le cas de systèmes à pointeurs plats sans mise à jour destructive le problème d'accessibilité généralisé est indécidable. Notons de plus que la traduction vers les systèmes à compteurs présentée dans ce chapitre s'inspire de la méthode utilisée dans [BI07]. Remarquons également que les propriétés que nous pouvons exprimer sont plus fortes que dans [BI07] car nous considérons des formes mémoire symboliques qui nous autorisent à manipuler les longueurs des différentes listes avec des formules de Presburger.

6.2.3.1 Décidabilité

Dans un premier temps, nous proposons une nouvelle classe de systèmes à pointeurs plats sans mise à jour destructive pour laquelle nous avons le résultat de décidabilité suivant :

Théorème 6.26 *Le problème d'accessibilité symbolique généralisé est décidable pour les systèmes à pointeurs plats sans mise à jour destructive et sans test d'alias.*

Preuve : Soient $S = \langle Q, P, E\rangle$ un système à pointeurs plat sans mise à jour destructive et sans test d'alias et (q_0, EMS_0) et (q, EMS) deux configurations symboliques dans $Q \times \mathcal{EMS}$ telles que $EMS_0 = \{(FM_1, \phi_1), \ldots, (FM_k, \phi_k)\}$ et $EMS = \{(FM'_1, \phi'_1), \ldots, (FM'_l.\phi'_l)\}$. Nous considérons de plus le système à compteurs $S_R = \langle Q_R, X_P, E_R\rangle$ dont nous avons présenté la construction précédemment. Nous notons $TS(S) = \langle Q \times \mathcal{GM}^{err}, E, \rightarrow\rangle$ et $TS(S_R) = \langle Q_R \times \mathbb{N}^{X_P}, E_R, \rightarrow_R\rangle$ les systèmes de transitions associés à S et S_R. Pour rappel nous avons $Q_R = Q \times \mathcal{FMR}$.
D'après le lemme 6.17, S_R est un système à compteurs linéaire à monoïde fini, de plus comme S est plat, d'après le lemme 6.18, S_R est aussi plat. Nous pouvons alors réutiliser le résultat du théorème [FL02] qui nous dit que la relation d'accessibilité d'un système à compteurs linéaire plat et à monoïde fini est effectivement définissable dans Presburger. Nous notons X_{P0} l'ensemble $X_p \uplus \{x_0\}$ dont la variable x_0 nous servira pour parler des états de contrôle de S_R qui, comme nous l'avons déjà vu pour d'autres systèmes à compteurs, peuvent être considérés comme des entiers de $[1..|Q_R|]$. Il existe alors une formule de Presburger $\Phi_\rightarrow \in$ **Presb**(X_{P0}, X'_{P0}) telle que $\rightarrow_R = [\![\Phi_\rightarrow]\!]_{X_{P0}, X'_{P0}}$.
Soient $i \in [1..k]$ et $j \in [1..l]$ et $FMR, FMR' \in \mathcal{FMR}$, nous construisons alors la formule $\Phi_{i,j}(FMR, FMR')$ suivante :

$$x_0 = (q_0, FMR) \wedge \Psi_{FMR}(FM_i, \phi_i) \wedge \Phi_\rightarrow \wedge x'_0 = (q, FMR') \wedge \Psi'_{FMR'}(FM'_j, \phi'_j)$$

où la formule $\Psi'_{FMR'}(MS'_j, \phi_j)$ est la formule $\Psi_{FMR'}(MS'_j, \phi_j)$ dans laquelle les variables libres dans X_P ont été renommés avec leur version primée appartenant à X'_P. D'après le lemme 6.25, les formules $T_{FMR}(MS_i, \phi_i)$ et $T'_{FMR'}(MS'_j, \phi'_j)$ appartiennent à logique $(\mathbb{N}, +, |, 0, 1)^\exists$ et en utilisant l'élimination des quantificateurs pour les formules de l'arithmétique de Presburger, nous en déduisons que la formule $\Phi_{i,j}(FMR, FMR')$ peut être récrite en une formule de $(\mathbb{N}, +, |, 0, 1)^\exists$.

Nous montrons maintenant qu'il existe $GM' \in [\![(FM'_j, \phi'_j)]\!]$ tel que nous avons (q, GM') dans **Reach**$(S, (q_0, (FM_i, \phi_i))$ si et seulement si il existe FMR et FMR' dans $\mathcal{FMR}$ tels que la formule $\Phi_{i,j}(FMR, FMR')$ est satisfiable.

Supposons qu'il existe $GM' \in [\![(FM'_j, \phi'_j)]\!]$ tel que $(q, GM') \in$ **Reach**$(S, (q_0, (FM_i, \phi_i))$. Il existe donc $GM \in [\![(FM_i, \phi_i)]\!]$ tel que $(q, GM') \in$ **Reach**$(S, (q_0, GM))$. Alors, en utilisant le lemme 6.10, il existe une forme mémoire restreinte FMR et une valuation $\mathbf{v} : X_P \rightarrow \mathbb{N}$ telles que $GM =$

$FMR[\mathbf{v}_{FMR}, \mathbf{v}_P]$. D'après le lemme 6.25, nous avons alors $\mathbf{v} \models \Psi_{FMR}(FM_i, \phi_i)$. De plus, par définition de la relation $\sim_R$, nous avons $(q_0, GM) \sim_R ((q_0, FMR), \mathbf{v})$. Or, d'après le lemme 6.19, comme S est un système sans test d'alias, la relation $\sim_R$ est une bisimulation entre $TS(S)$ et $TS(S_R)$ et en utilisant le résultat du lemme 6.21, comme $(q, GM') \in \mathbf{Reach}(S, (q_0, GM))$, nous en déduisons qu'il existe une forme mémoire restreinte FMR' et une valuation $\mathbf{v}' : X_P \to \mathbb{N}$ tels que $(q, GM') \sim_R ((q, FMR'), \mathbf{v}')$ et $((q, FMR'), \mathbf{v}') \in \mathbf{Reach}(S_R, ((q_0, FMR), \mathbf{v})$. Comme nous avons $(q, GM') \sim_R ((q, FMR'), \mathbf{v}')$, nous en déduisons que $FMR[\mathbf{v}'_{FMR'}, \mathbf{v}'_P] \in [\![(FM'_j, \phi'_j)]\!]$ et donc $\mathbf{v}' \models \Psi_{FMR'}(FM'_j, \phi'_j)$. Ainsi si nous récapitulons, nous avons :

- $\mathbf{v} \models \Psi_{FMR}(FM_i, \phi_i)$, et,
- $\mathbf{v}' \models \Psi_{FMR'}(FM'_j, \phi'_j)$, et,
- $((q_0, FMR), \mathbf{v}) \to_R^* ((q, FMR'), \mathbf{v}')$.

Comme $\to_R = [\![\Phi_\to]\!]_{X_{P0}, X'_{P0}}$, nous en déduisons que la formule $\Phi_{i,j}(FMR, FMR')$ est satisfiable. Nous pouvons prouver de la même façon la réciproque, à savoir que si il existe FMR et FMR' dans $\mathcal{FMR}$ tels que la formule $\Phi_{i,j}(FMR, FMR')$ est satisfiable alors il existe $GM' \in [\![(FM'_j, \phi'_j)]\!]$ tel que $(q, GM') \in \mathbf{Reach}(S, (q_0, (FM_i, \phi_i))$.

Nous avons finalement que le problème d'accessibilité symbolique généralisé est vérifié avec les instances S, (q_0, EMS_0) et (q, EMS) si et seulement si il existe $i \in [1..k]$ et $j \in [1..l]$ tels qu'il existe $GM \in [\![(FM'_j, \phi'_j)]\!]$ vérifiant $(q, GM) \in \mathbf{Reach}(S, (q_0, (FM_i, \phi_i))$ c'est-à-dire si et seulement si il existe $i \in [1..k]$ et $j \in [1..l]$ et FMR et FMR' dans $\mathcal{FMR}$ tels que la formule $\Psi_{i,j}(FMR, FMR')$ est satisfiable. Comme de plus, pour tout $i \in [1..k]$ et $j \in [1..l]$ et FMR et FMR' dans $\mathcal{FMR}$, la formule $\Phi_{i,j}(FMR, FMR')$ appartient à la logique $(\mathbb{N}, +, |, 0, 1)^\exists$ pour laquelle le problème de satisfiabilité est décidable et comme le nombre de formes mémoire restreintes est fini, nous en déduisons la décidabilité du problème d'accessibilité symbolique généralisé pour les systèmes à pointeurs plats sans mise à jour destructive et sans test d'alias. □

Nous allons maintenant nous intéresser au problème de model-checking pour les systèmes à pointeurs plats sans mise à jour destructive muni d'une configuration symbolique initiale acyclique. Nous dirons qu'un état mémoire symbolique $EMS = \{(FM_1, \phi_1), \ldots, (FM_k, \phi_k)\}$ est acyclique si et seulement si pour tout $i \in [1..k]$, la forme mémoire FM_i est acyclique. Rappelons que lorsque nous considérons une forme mémoire symbolique acyclique (FM, ϕ), pour toute forme mémoire restreinte FMR, la formule $\Psi_{FMR}(FM, \phi)$ est une formule de l'arithmétique de Presburger. Ceci va nous permettre de traduire une formule de CTL^*_{mem} sur le système à pointeurs sans mise à jour destructive S en une formule de FOCTL*(Pr) sur S_R.

Soient $S = \langle Q, P, E\rangle$ un système à pointeurs plat sans mise à jour destructive et (q_0, EMS_0) une configuration symbolique acyclique dans $Q \times \mathcal{EMS}$ telle que $EMS_0 = \{(FM_1, \phi_1), \ldots, (FM_k, \phi_k)\}$. Nous considérons le système à compteurs $S_R = \langle Q_R, X_P, E_R\rangle$ dont nous avons présenté la construction précédemment. Nous notons $TS(S) = \langle Q \times \mathcal{GM}^{err}, E, \to\rangle$ et $TS(S_R) = \langle Q_R \times \mathbb{N}^{X_P}, E_R, \to_R\rangle$ les systèmes de transitions associés à S et S_R. Pour rappel, nous avons $Q_R = Q \times \mathcal{FMR}$. Comme pour la preuve du théorème 6.4, dans lequel nous avons montré que le problème de model-checking est décidable pour les systèmes à pointeurs dont le système à compteurs obtenue avec la première traduction est plat, nous construisons maintenant à partir d'une formule Φ de $\text{CTL}^*_{mem}[Q, P, X]$ une formule de FOCTLX*(Pr)$[X_{P0}]$ $T(\Phi)$. La construction de $T(\Phi)$ se fait par induction de la façon suivante :

- $T(q) = \bigvee_{FMR \in \mathcal{FMR}} x_0 = (q, FMR)$,
- si $\Phi = EMS$ avec $EMS = \{(FM_1, \phi_1), \ldots, (FM_t, \phi_k)\}$ alors $T(\Phi) = \bigvee_{i \in [1..t]} T(FM_i, \phi_i)$, avec :
  - si FM n'est pas acyclique, $T(FM, \phi) = false$,
  - si FM est acyclique,

$$T(FM, \phi) = \bigvee_{q \in Q} \bigvee_{FMR \in \mathcal{FMR}} x_0 = (q, FMR) \wedge \Psi_{FMR}(FM, \phi)$$

- $T(\neg\Phi) = \neg T(\Phi)$,
- $T(\Phi \wedge \Phi') = T(\Phi) \wedge T(\Phi')$,
- $T(\mathtt{X}\Phi) = \mathtt{X}T(\Phi)$,
- $T(\Phi \mathtt{U} \Phi') = T(\Phi) \mathtt{U} T(\Phi')$,
- $T(\mathtt{A}\Phi) = \mathtt{A}T(\Phi)$.

Nous avons alors le lemme suivant :

Lemme 6.27 *Soit Φ une formule de* CTL$^*_{mem}[Q, P, X]$. *Pour toutes exécutions π de $TS(S)$ et π_R de $TS(S_R)$ telles que $|\pi| = |\pi_C|$ et telles que pour tout $i \in [0..|\pi| - 1]$, $\pi(i) \sim^{ac}_R \pi_C(i)$, et pour tout entier $k \in [1..|\pi| - 1]$, nous avons $\pi, k \models \Phi$ si et seulement si $\pi_C, k \models T(\Phi)$.*

Preuve : La preuve se fait par induction structurelle sur la formule Φ. Tout d'abord montrons que si $\Phi = q$ ou si $\Phi = EMS$ le lemme est vrai.
Supposons que $\Phi = q$. Soient π une exécution de $TS(S)$ et π_R une exécution de $TS(S_R)$ vérifiant les conditions énoncées. Soit $k \in [0..|\pi| - 1]$. Nous avons $\pi(k) = (q', GM)$ et $\pi_R(k) = ((q'', FM), \mathbf{v})$. Comme $\pi(k) \sim^{ac}_R \pi_R(k)$, nous avons nécessairement $q' = q''$ et par conséquent $\pi, k \models q$ si et seulement si $\pi_R, k \models \bigvee_{FMR \in \mathcal{FMR}} x_0 = (q, FMR)$ et ceci si et seulement si $q' = q$.
Supposons que $\Phi = EMS$ avec $EMS = \{(FM_1, \phi_1), \ldots, (FM_t, \phi_t)\}$. Soient π une exécution de $TS(S)$ et π_R une exécution de $TS(S_R)$ vérifiant les conditions énoncées. Soit $k \in [0..|\pi| - 1]$. Nous notons $\pi(k) = (q, GM)$ et $\pi_R(k) = ((q, FMR), \mathbf{v})$. Supposons que $\pi, k \models EMS$. Alors il existe $i \in [1..t]$ tel que $GM \in [\![(FM_i, \phi_i)]\!]$. Comme de plus $GM \sim^{ac}_R (FMR, \mathbf{v})$ (ceci car $\pi(k) \sim^{ac}_R \pi_R(k)$), nous avons $GM = FMR[\mathbf{v}_{FM}, \mathbf{v}_P]$. Par conséquent, d'après le lemme 6.25, nous avons $\mathbf{v} \models \Psi_{FMR}(FM_i, \phi_i)$. Et comme nécessairement FM_i est acyclique (car GM est acyclique), nous en déduisons que $\pi_R, k \models T(EMS)$. Le fait que si $\pi_R, k \models T(EMS)$ alors $\pi, k \models EMS$ se prouve de manière identique.
Nous supposons maintenant le lemme vrai pour toutes les sous-formules de Φ et montrons que le lemme reste vrai pour Φ. Soient π une exécution de $TS(S)$ et π_R une exécution de $TS(S_R)$ vérifiant les conditions énoncées. Soit $k \in [0..|\pi| - 1]$.

- Si $\Phi = \neg\Phi'$, comme par hypothèse d'induction nous avons $\pi, k \models \Phi'$ si et seulement si $\pi_R, k \models T(\Phi')$ et comme $T(\neg\Phi') = \neg T(\Phi')$, nous avons bien $\pi, k \models \Phi$ si et seulement si $\pi_R, k \models T(\Phi)$.
- Si $\Phi = \Phi' \vee \Phi''$, comme par hypothèse d'induction nous avons $\pi, k \models \Phi'$ si et seulement si $\pi_R, k \models T(\Phi')$ et $\pi, k \models \Phi''$ si et seulement si $\pi_R, k \models T(\Phi'')$ et comme $T(\Phi' \vee \Phi') = T(\Phi') \vee T(\Phi'')$, nous avons bien $\pi, k \models \Phi$ si et seulement si $\pi_R, k \models T(\Phi)$.
- Si $\Phi = \mathtt{X}\Phi'$. Si $k \geq |\pi| - 1$, nous avons $\pi, k \not\models \mathtt{X}\Phi$ et $\pi_R, k \not\models \mathtt{X}T(\Phi)$. Supposons que $k < |\pi| - 1$, alors par hypothèse d'induction nous avons $\pi, k + 1 \models \Phi'$ si et seulement si $\pi_R, k + 1 \models T(\Phi')$ et par conséquent nous avons bien que $\pi, k \models \mathtt{X}\Phi'$ si et seulement si $\pi_R, k \models \mathtt{X}T(\Phi')$.

– Si $\Phi = \Phi' \mathtt{U} \Phi''$. Si il existe $j \in \mathbb{N}$ tel que $i \leq j \leq |\pi|$ et tel que $\pi, j \models \Phi''$ et tel que pour tout $l \in \mathbb{N}$ vérifiant $i \leq l < j$, $\pi, l \models \Phi'$ alors par hypothèse d'induction $\pi_R, j \models T(\Phi'')$ et pour tout $l \in \mathbb{N}$ vérifiant $i \leq l < j$, $\pi_R, l \models T(\Phi')$. Par conséquent $\pi_R, k \models T(\Phi') \mathtt{U} T(\Phi'')$. Si il existe $j \in \mathbb{N}$ tel que $i \leq j \leq |\pi|$ et tel que $\pi_R, j \models T(\Phi'')$ et tel que pour tout $l \in \mathbb{N}$ vérifiant $i \leq l < j$, $\pi_R, l \models T(\Phi')$ alors par hypothèse d'induction $\pi, j \models \Phi''$ et pour tout $l \in \mathbb{N}$ vérifiant $i \leq l < j$, $\pi, l \models \Phi'$. Par conséquent $\pi, k \models \Phi' \mathtt{U} \Phi''$.
– Si $\Phi = \mathtt{A}\Phi'$. Supposons que pour toute exécution π' de $TS(S)$ telle que $\pi_{\leq k} = \pi'_{\leq k}$, $\pi', k \models \Phi'$. Soit π'_R une exécution de $TS(S_R)$ telle que $\pi_{R_{\leq k}} = \pi'_{R_{\leq k}}$. Alors en utilisant le lemme 6.20, comme $\pi(k) \sim_R^{ac} \pi'_R(k)$, nous en déduisons qu'il existe une exécution de π' de $TS(S)$ telle que $\pi_{\leq k} = \pi'_{\leq k}$ et telle que $|\pi'| = |\pi'_R|$ et telle que pour tout $i \in [0..|\pi'| - 1]$, $\pi'(i) \sim_R^{ac} \pi'_R(i)$. En utilisant l'hypothèse d'induction, comme $\pi', k \models \Phi'$, nous avons $\pi'_R, k \models T(\Phi')$. Nous en déduisons que $\pi_R, k \models \mathtt{A}T(\Phi')$. De la même façon, nous pouvons montrer que si $\pi_R, k \models \mathtt{A}T(\Phi')$ alors $\pi, k \models \mathtt{A}\Phi'$.

□

Nous pouvons alors donner le théorème suivant qui étend au model-checking les résultats de [BI07] dans le cas d'une configuration initiale acyclique, à savoir :

Théorème 6.28 *Le problème de model-checking est décidable pour les systèmes à pointeurs plats sans mise à jour destructive et munis d'une configuration symbolique initiale acyclique.*

Preuve : Soient $S = \langle Q, P, E \rangle$ un système à pointeurs plat sans mise à jour destructive, (q_0, EMS_0) avec $EMS_0 = \{(FM_1, \phi_1), \ldots, (FM_t, \phi_t)\}$ une configuration symbolique initiale acyclique de $TS(PS)$ et Φ une formule de CTL$^*_{mem}[Q, P, X]$. Nous considérons le système à compteurs $S_R = \langle Q_R, X_P, E_R \rangle$. Soit $i \in [1..t]$. Nous allons montrer que pour toutes les exécutions π de $TS(S)$ vérifiant $\pi(0) \in \{q_0\} \times [\![(FM_i, \phi_i)]\!]$, nous avons $\pi, 0 \models \Phi$ si et seulement si pour tout $FMR \in \mathcal{FMR}$ et pour toutes les exécutions π_R de $TS(S_R)$ vérifiant $\pi_R(0) \in \{q_0, FMR\} \times [\![\Psi_{FMR}(FM_i, \phi_i)]\!]$ nous avons $\pi_R, 0 \models T(\Phi)$. Remarquons qu'une fois que nous aurons montré cela, comme le système à compteurs linéaire S_R est plat (d'après le lemme 6.18 et le fait que S est plat) et à monoïde fini (d'après le lemme 6.17) et comme de plus il existe un nombre fini de formes mémoire restreintes, en utilisant le théorème 3.3 nous pourrons déduire le résultat énoncé en utilisant que le problème du model-checking symbolique généralisé est décidable pour les systèmes à compteurs linéaires plats et à monoïde fini.
Supposons que pour toutes les exécutions π de $TS(S)$ vérifiant $\pi(0) \in \{q_0\} \times [\![(FM_i, \phi_i)]\!]$ nous avons $\pi, 0 \models \Phi$. Soit $FMR \in \mathcal{FMR}$ et soit π_R une exécution de $TS(S_R)$ vérifiant $\pi_R(0) \in \{q_0, FMR\} \times [\![\Psi_{FMR}(FM_i, \phi_i)]\!]$. Supposons que $\pi_R(0) = ((q_0, FMR), \mathbf{v})$ alors nous avons $\mathbf{v} \models \Psi_{FMR}(FM_i, \phi_i)$ et par conséquent $FMR[\mathbf{v}_{FM_i}, \mathbf{v}_P] \in [\![(FM_i, \phi_i)]\!]$. Nous posons maintenant $c_0 = (q_0, FMR[\mathbf{v}_{FM_i}, \mathbf{v}_P])$, par définition de $\sim_R^{ac}$, nous avons $c_0 \sim_R^{ac} \pi_R(0)$ (car FM_i est acyclique). En utilisant le lemme 6.20, nous montrons facilement par induction qu'il existe alors une exécution π de $TS(S)$ tel que $|\pi| = |\pi_R|$ et $\pi(0) = c_0$ et pour tout $j \in [0..|\pi| - 1]$, $\pi(j) \sim_R^{ac} \pi_R(j)$. Comme $\pi, 0 \models \Phi$, en utilisant le lemme 6.27, nous déduisons que $\pi_R, 0 \models T(\Phi)$. De la même façon, nous pouvons montrer que si pour tout $FMR \in \mathcal{FMR}$, pour toutes les exécutions π_R de $TS(S_R)$ vérifiant $\pi_R(0) \in \{q_0, FMR\} \times [\![\Psi_{FMR}(FM_i, \phi_i)]\!]$ nous avons $\pi_R, 0 \models T(\Phi)$, alors pour toutes les exécutions π de $TS(S)$ vérifiant $\pi(0) \in \{q_0\} \times [\![(FM_i, \phi_i)]\!]$, nous avons $\pi, 0 \models \Phi$. □

Nous avons ainsi établi différentes classes de systèmes à pointeurs plats pour lesquelles certains des problèmes que nous avons donnés précédemment sont décidables.

6.2.3.2 Indécidabilité

Dans cette section, nous allons voir que les hypothèses permettant d'obtenir les résultats de décidabilité dans les théorèmes 6.26 et 6.28 présentés précédemment, peuvent difficilement être relâchées. Pour montrer les résultats d'indécidabilité qui suivent, nous réduisons un problème indécidable de satisfiabilité de formules arithmétiques vers différents problèmes.

Nous exposons d'abord le problème arithmétique que nous considérons (ce problème est une variante d'un problème équivalent introduit dans [BI07]). Soit $Z = \{z_1, ..., z_k\}$ un ensemble de variables et nous considérons une formule arithmétique $\mathcal{E} = \bigwedge_{i\in[1..t]} \mathcal{E}_i$ qui est une conjonction de formules de la forme $z_i = z_j + z_k$ ou $z_i = a$ (où a est un entier) ou $z_i = \textbf{ppcm}(z_j, z_k)$ (**ppcm** désignant ici le plus petit multiple commun). Le problème consiste à savoir si il existe une valuation de variables $\mathbf{v} : Z \to \mathbb{N}$ telle que cette valuation satisfait la formule $\mathcal{E}$. Il s'avère que le 10ème problème de Hilbert se réduit à ce problème. Le 10ème problème de Hilbert consiste à savoir si étant donné un polynôme $P(x_1, \ldots, x_n)$ à coefficients dans $\mathbb{Z}$ sur un ensemble de variables $\{x_1, \ldots, x_n\}$, il existe une solution entière $\{k_!, \ldots, k_n\} \in \mathbb{N}^n$ telle que $P(k_1, \ldots, k_n) = 0$. Le 10ème problème de Hilbert a été montré indécidable dans [Mat70]. Les idées clefs, qui sont données dans [BI07], pour réduire le 10ème problème de Hilbert à notre problème sont les suivantes. Pour tout $x, y, z \in \mathbb{N}$, nous avons :

– $y = x^2$ si et seulement si $y + x = \textbf{ppcm}(x, x + 1)$, et,
– $z = xy$ si et seulement si $4z = (x + y)^2 - (x - y)^2$.

Il est ainsi possible en ajoutant des variables de réduire le 10ème problème de Hilbert à la satisfiabilité d'une formule $\mathcal{E}$ ayant la forme donnée ci-dessus. Notons que, sans perte de généralités, nous pouvons supposer que nous cherchons une valuation à valeurs dans $\mathbb{N}^*$ et que ce problème est également indécidable

Nous allons montrer comment encoder les propositions de la forme $z_i = z_j + z_k$ ou $z_i = a$ ou $z_i = \textbf{ppcm}(z_j, z_k)$ dans un système à pointeurs plat sans mise à jour destructive.

Tout d'abord nous considérons le système à pointeurs S et la forme mémoire FM représentée à la figure 6.13. Nous constatons que si (FM, ϕ) est une forme mémoire symbolique telle que FM est la forme mémoire représentée à la figure 6.13, alors il existe $GM \in \mathcal{GM}$ tel que $(q_0, GM) \in \textbf{Reach}(S, (q_1, (FM, \phi)))$ si et seulement si $\phi \Rightarrow x_3 = x_1 + x_2$. De plus le système à pointeurs S est plat sans mise à jour destructive et sans test d'alias.

Considérons le système à pointeurs S et la forme mémoire FM représentée à la figure 6.14. Nous constatons que si (FM, ϕ) est une forme mémoire symbolique, alors il existe $GM \in \mathcal{GM}$ tel que $(q_0, GM) \in \textbf{Reach}(S, (q_1, (FM, \phi)))$ si et seulement si $\phi \Rightarrow x_1 = a$ avec $a \in \mathbb{N}^*$. De plus, là encore le système à pointeurs S est plat sans mise à jour destructive et sans test d'alias.

Considérons le système à pointeurs S et la forme mémoire FM représentée à la figure 6.15. Nous constatons que si (FM, ϕ) est une forme mémoire symbolique, nous avons qu'il existe $GM \in \mathcal{GM}$ tel que $(q_0, GM) \in \textbf{Reach}(S, (q_1, (FM, \phi)))$ si et seulement si $\phi \Rightarrow x_3 = \textbf{ppcm}(x_1, x_2)$. Ceci car si nous regardons le système à pointeurs S, nous nous apercevons que si une exécution partant de l'état q_1 avec un graphe mémoire dans $[\![(FM, \phi)]\!]$ arrive dans l'état q_0, alors la garde $p == p_0 \wedge s == s_0 \wedge r == \texttt{null}$ assure que la valuation associé à x_3 est un multiple des valuations associées à x_1 et x_2, de plus la garde entre l'état q_4 et q_1 assure qu'il s'agit du plus petit multiple. Là encore le système

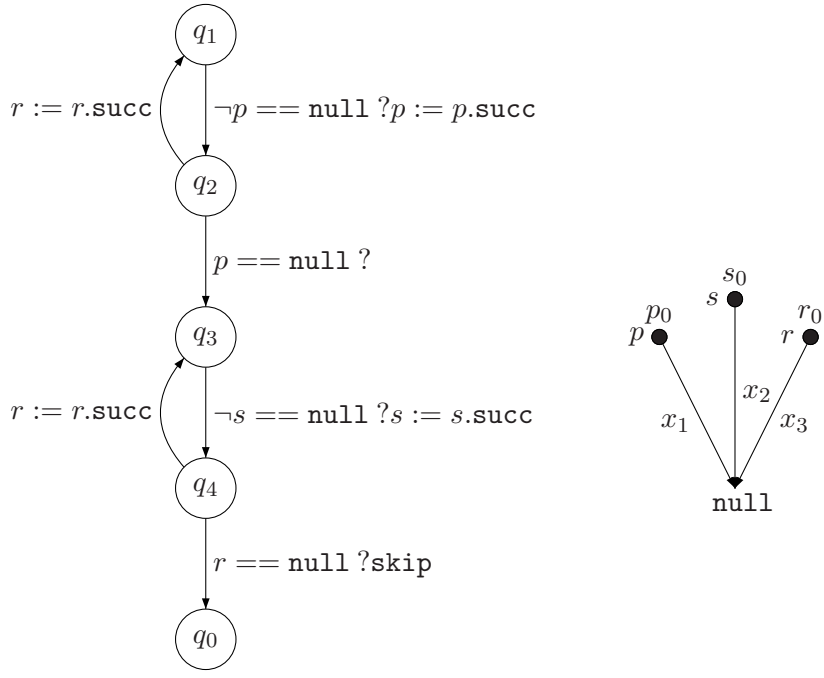


FIGURE 6.13 – Un système à pointeurs pour encoder $x_3 = x_2 + x_1$

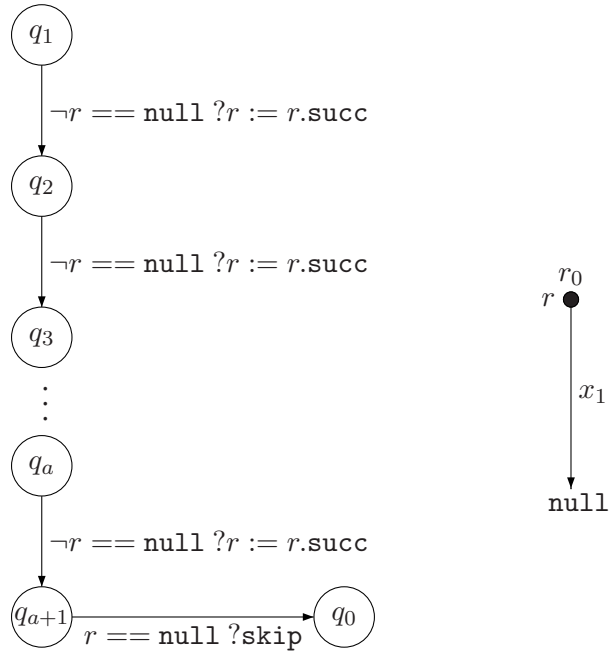


FIGURE 6.14 – Un système à pointeurs pour encoder $x_1 = a$

à pointeurs S est plat sans mise à jour destructive mais il comporte des tests d'alias.

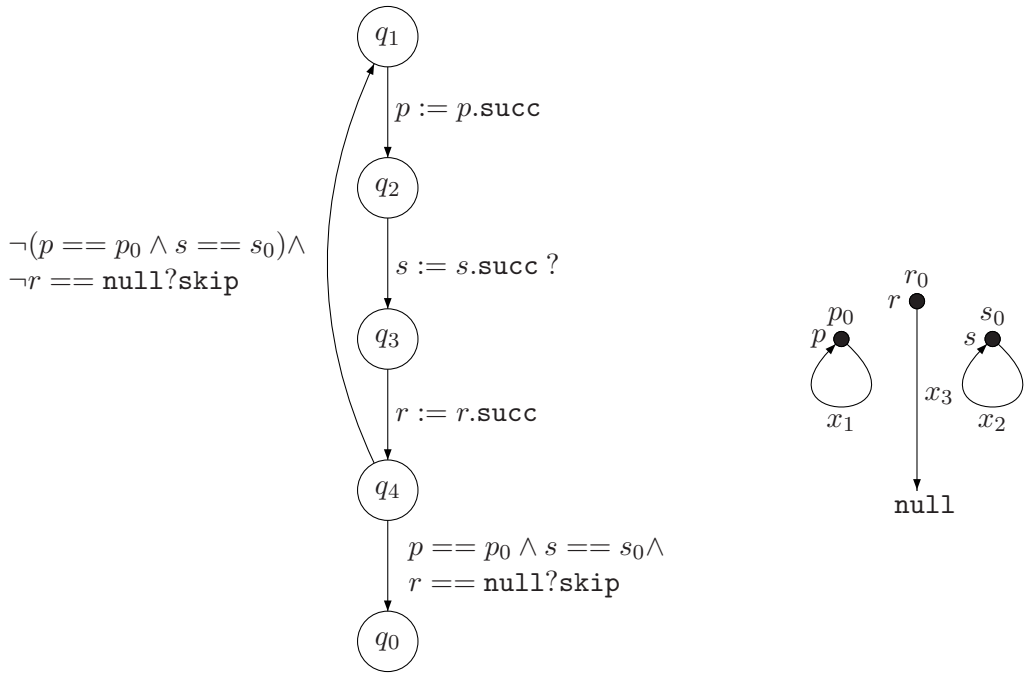


FIGURE 6.15 – Un système à pointeurs pour encoder $x_3 = \mathbf{ppcm}(x_1, x_2)$

Si maintenant nous considérons une formule $\mathcal{E} = \bigwedge_{i\in[1..t]} \mathcal{E}_i$ où pour tout $i \in [1..t]$, $\mathcal{E}_i$ est une formule de la forme $z_i = z_j + z_k$ ou $z_i = a$ (avec $a \in \mathbb{N}^*$) ou $z_i = \mathbf{ppcm}(z_j, z_k)$ alors nous pouvons construire un système a pointeurs $S_\mathcal{E}$ qui revient en fait à connecter pour tout $i \in [1..(t-1)]$, l'état de contrôle q_0^i (l'état correspondant à q_0 sur les différentes figures) du système à pointeurs permettant d'encoder la proposition $\mathcal{E}_i$ à l'état de contrôle q_1^{i+1} (l'état correspondant à q_1 sur la formule) du système à pointeurs encodant la proposition $\mathcal{E}_{i+1}$. Le système à pointeurs obtenu $S_\mathcal{E}$ est alors un système à pointeurs plat sans mise à jour destructive. De la même façon, nous construisons une forme mémoire $FM_\mathcal{E}$ de façon à regrouper au sein d'une même forme mémoire (en faisant une union disjointe) les différentes formes mémoire nécessaires pour encoder chaque $\mathcal{E}_i$, la formule $\phi \in \mathbf{Presb}(X_{FM_\mathcal{E}})$ s'assure juste que les compteurs des différentes listes dans $FM_\mathcal{E}$ encodant les mêmes variables font bien la même longueur. Remarquons de plus que la forme mémoire $FM_\mathcal{E}$ construite n'est pas acyclique (car chaque forme mémoire utilisée pour encoder les propositions de la forme $z_i = \mathbf{ppcm}(z_j, z_k)$ contient aux moins deux listes cycliques). D'après les propriétés des différents systèmes à pointeurs des figures 6.13 à 6.15 que nous avons données précédemment, nous en déduisons qu'il existe un graphe mémoire GM tel que $(q_0^t, GM) \in \mathbf{Reach}(S_\mathcal{E}, (q_1^0, (FM_\mathcal{E}, \phi)))$ si et seulement si la formule $\mathcal{E}$ est satisfiable. Remarquons de plus, que le système à pointeurs $S_\mathcal{E}$ muni de la configuration symbolique initiale $(q_1^0, (FM_\mathcal{E}, \phi))$ a un comportement sans erreur, par conséquent nous pourrions ajouter une variable de pointeurs p pointant toujours sur le noeud `null` et réaliser l'action $p := p.\texttt{succ}$ une fois arrivé dans l'état q_0^t. Ceci aurait pour conséquence d'effectuer une erreur de segmentation. Nous pourrions de plus réaliser un stratagème similaire pour les fuites mémoire. L'indécidabilité du problème de satisfiabilité des formules de la forme $\mathcal{E}$, nous permet alors de donner le théorème suivant :

Théorème 6.29 *Les problèmes d'accessibilité généralisés d'une erreur de segmentation et d'une fuite mémoire sont indécidables pour les systèmes à pointeurs plats sans mise à jour destructive.*

En utilisant la proposition 4.23 qui nous dit que ces problèmes peuvent se ramener à un nombre fini d'instances du problème d'accessibilité symbolique généralisé, nous obtenons :

Corollaire 6.30 *Le problème d'accessibilité symbolique généralisé est indécidable pour les systèmes à pointeurs plats sans mise à jour destructive.*

Ainsi avec le théorème 6.28, nous avons montré que le problème de model-checking est décidable pour les systèmes à pointeurs plats sans mise à jour destructive muni d'une configuration symbolique initiale acyclique, et nous voyons ici que si nous supprimons l'hypothèse sur les configurations acycliques, le problème devient indécidable. De la même façon avec le théorème 6.26, nous avons montré que le problème d'accessibilité symbolique généralisé est décidable pour les systèmes à pointeurs plats sans mise à jour destructive et sans test d'alias, et nous voyons que si nous retirons l'hypothèse de l'absence de test d'alias, là encore le problème devient indécidable.

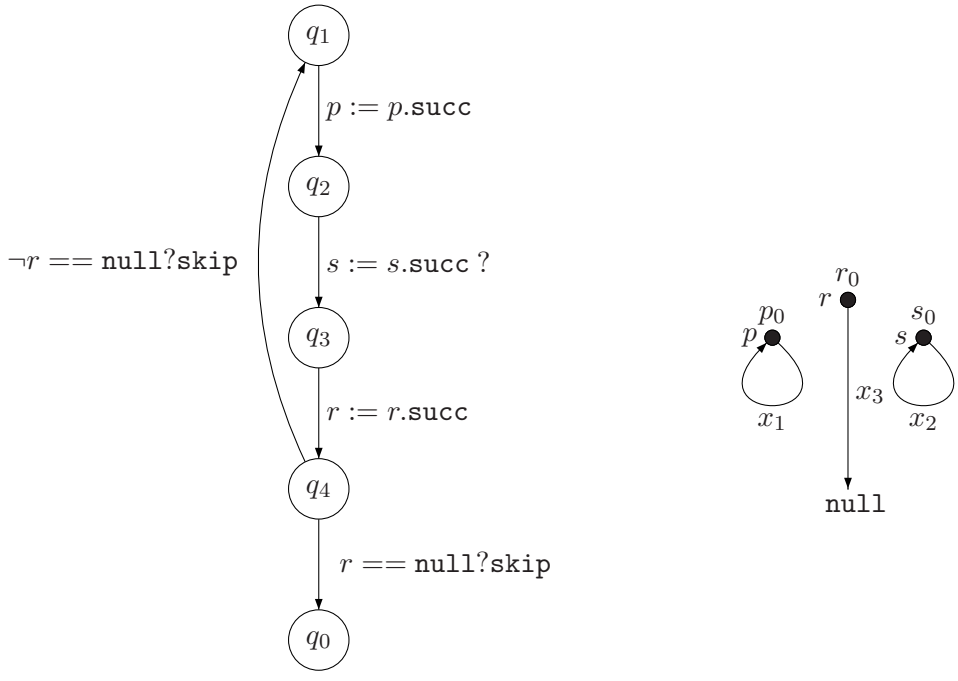


FIGURE 6.16 – Un autre système à pointeurs pour encoder $x_3 = \mathbf{ppcm}(x_1, x_2)$

Pour finir une preuve similaire à celle présentée ci-dessus va nous permettre de montrer que le problème de model-checking est indécidable pour les systèmes à pointeurs plats sans mise à jour destructive et sans test d'alias, en d'autres termes que le résultat du théorème 6.26 ne s'étend pas au model-checking. Pour ce faire, nous modifions la façon d'encoder les formules de la forme $z_i = \mathbf{ppcm}(z_j, z_k)$, car nous avons constaté que le système à pointeurs proposé dans la figure 6.15 était effectivement plat et sans mise à jour destructive, mais en revanche il utilise des tests d'alias. Nous utilisons maintenant le système à pointeurs de la figure 6.16 qui est en fait le même que celui de la figure 6.15 dans lequel les tests d'alias ont été tout simplement enlevés, quant à la forme mémoire FM considérée elle reste la même.

Cependant avec ce nouveau système à compteurs nous ne pouvons plus garantir pour une forme mémoire symbolique (FM, ϕ) qu'il existe un graphe mémoire GM tel que (q_0, GM) appartient à

Reach$(S, (q_1, (FM, \phi))$ si et seulement si $\phi \Rightarrow x_3 = \textbf{ppcm}(x_1, x_2)$ car c'étaient précisément les tests d'alias qui permettaient de garantir ce point. Cependant nous pouvons construire une formule de CTL$^*_{mem}$qui permet de simuler l'effet des gardes. En effet, il suffit que nous nous assurions que parmi toutes les exécutions commençant avec la configuration symbolique initiale il existe une exécution telle que lorsque elle est dans l'état q_4, nous avons $\neg(p == p_0 \wedge s == s_0)$ jusqu'à ce qu'elle arrive dans l'état q_4 et que la garde $(p == p_0 \wedge s == s_0 \wedge r == \texttt{null})$ est satisfaite. Cette propriété s'exprime facilement avec une formule de CTL$^*_{mem}$utilisant le quantificateur existentiel sur les chemins et l'opérateur temporel U qui permet de dire "jusqu'à". En effet, si nous considérons la formule $\Psi = \texttt{E}((q_4 \Rightarrow EMS_1)\texttt{U}(q_4 \wedge (EMS_2))$ où EMS_1 est la disjonction de toutes les formes mémoire symboliques vérifiant $\neg(p == p_0 \wedge s == s_0)$ et EMS_2 est la disjonction de toutes les formes mémoire symboliques satisfaisant $(p == p_0 \wedge s == s_0 \wedge r == \texttt{null})$, nous en déduisons que le problème de model-checking avec les instances S, $(q_1, (FM, \phi))$ et ψ retourne vrai si et seulement si $\phi \Rightarrow x_3 = \textbf{ppcm}(x_1, x_2)$.

En utilisant une méthode similaire à celle décrite auparavant, il est ainsi possible de réduire le problème de satisfiabilité pour une formule $\mathcal{E}$ au problème de model-checking pour les systèmes à pointeurs plats sans mise à jour destructive et sans test d'alias, d'où nous déduisons le résultat déjà énoncé plus haut à savoir :

Théorème 6.31 *Le problème du model-checking est indécidable pour les systèmes à pointeurs plats sans mise à jour destructive et sans test d'alias.*

Ce dernier théorème conclut notre étude des systèmes à pointeurs plats sans mise à jour destructive.

6.2.4 Tableau récapitulatif

Le tableau de la figure 6.17 récapitule les différents résultats de décidabilité et d'indécidabilité que nous avons établis dans ce chapitre concernant les problèmes d'accessibilité symbolique généralisé et de model-checking pour les systèmes à pointeurs plats.

Conclusion

Dans ce chapitre, nous avons proposé un cadre pour vérifier des propriétés temporelles sur les systèmes à pointeurs en étendant la logique CTL* de façon à pouvoir utiliser des états mémoire symboliques comme proposition atomique afin de parler de l'évolution du tas mémoire au cours du temps. Dans [BI07], les auteurs ont montré que certains problèmes de sûreté étaient indécidables pour des systèmes à pointeurs plats sans mise à jour destructive et devenaient décidables lorsque l'on supposaient que les systèmes à pointeurs considérés étaient munis d'une configuration symbolique initiale contenant au plus une liste cyclique. Nous avons étendu ces résultats au model-checking dans le cas de systèmes à pointeurs plats munis d'une configuration initiale acyclique. Nous avons de plus trouvé une nouvelle classe de systèmes à pointeurs plats sans mise à jour destructive pour laquelle le problème d'accessibilité symbolique généralisé est décidable mais pas le problème de model-checking, cette classe étant obtenue en considérant l'absence de tests d'alias dans les gardes utilisées. Nous avons finalement montré que les hypothèses que nous faisions pour obtenir la décidabilité pouvait difficilement être relâchées. Notons enfin qu'en faisant cette étude, nous espérions trouver une classe de systèmes à pointeurs plats permettant de regrouper un certain nombre de programmes sans boucles

Systèmes à pointeurs plats	Configurations symboliques initiales	Problème de l'accessibilité symbolique généralisé	Problème du model-checking
sans déplacement au successeur	Pas de restriction	Décidable	Décidable
sans mise à jour destructive	Acycliques	Décidable	Décidable
sans mise à jour destructive	Pas de restriction	Indécidable	Indécidable
sans mise à jour destructive et sans test d'alias	Pas de restriction	Décidable	Indécidable

FIGURE 6.17 – Tableau récapitulatif

imbriquées et pour laquelle les problèmes d'accessibilité seraient décidables. Cependant si nous observons les résultats obtenus, force est de constater qu'une telle classe de systèmes à pointeurs semble difficile à trouver car même avec des restrictions très fortes comme l'absence de mise à jour destructive, nous n'avons pas la décidabilité dans le cas général.

Troisième partie

L’outil TOPICS

Chapitre 7

Translation of Programs Into Counter Systems

Ce chapitre est dédié à la présentation de l'outil TOPICS que nous avons développé pendant cette thèse et dont nous nous servons pour résoudre les problèmes d'accessibilité symbolique d'une fuite mémoire et d'une erreur de segmentation sur des programmes écrits en C.

7.1 Présentation

TOPICS est l'acronyme pour "Translation Of Programs Into Counter Systems". Cet outil implante la traduction des systèmes à pointeurs vers des systèmes à compteurs que nous avons présentée au chapitre 5. TOPICS prend en entrée un programme écrit dans un fragment syntaxique du langage de programmation C ainsi qu'une description de la configuration initiale de la mémoire et à partir de cela, il produit un système à compteurs. Les programmes donnés en entrée manipulent des listes simplement chaînées, des entiers et des tableaux de taille fixe d'entiers et de listes. Les systèmes à compteurs obtenus sont fournis dans trois formats différents :

1. au format dot, qui permet ensuite grâce à l'outil GRAPHVIZ de visualiser une représentation graphique du système à compteurs,
2. au format de l'outil FAST ,
3. au format de l'outil ASPIC .

GRAPHVIZ est un outil qui, à partir d'une description structurelle d'un graphe, en génère une représentation graphique que l'on peut ensuite exporter vers différents formats. Cet outil est librement disponible à l'adresse suivante : http://www.graphviz.org/.
Comme nous l'avons vu au chapitre 1, FAST est un outil qui vérifie automatiquement des systèmes à compteurs en calculant de façon exacte leur ensemble d'accessibilité à partir d'un ensemble d'états initiaux.
ASPIC est un outil permettant aussi de vérifier automatiquement des systèmes à compteurs. Cet outil utilise la technique de l'interprétation abstraite [CC77] et calcule une surapproximation de l'ensemble d'accessibilité d'un système à compteurs. La méthode implantée manipule des abstractions décrites par des relations linéaires. Cette méthode est présentée dans [GH06]. L'avantage de cet outil est que son calcul termine toujours, contrairement à FAST , en revanche il se peut qu'il ne puisse pas résoudre un problème d'accessibilité à cause des surapproximations réalisées et il renvoie alors la réponse "I

don't know". Cet outil est disponible à l'adresse suivante : `http://laure.gonnord.org/pro/aspic/aspic.html`

`TOPICS` est développé en JAVA et le code source est constitué d'environ 16000 lignes. Il utilise les outils `JFLEX` [JFl] et `CUP` [Cup] pour parcourir les fichiers donnés en entrée. `TOPICS` est donné sous forme d'un fichier `jar` et par conséquent il peut être utilisé sur n'importe quelle plate-forme munie d'un interpréteur Java sans qu'il y ait besoin d'une installation particulière.

Des informations supplémentaires sur `TOPICS` sont disponibles à l'adresse suivante :
`http://www.lsv.ens-cachan.fr/~sangnier/TOPICS/`

7.2 Syntaxe d'entrée

Comme nous l'avons dit `TOPICS` prend deux fichiers en entrée :

1. un programme écrit dans un fragment syntaxique du C, et,
2. un fichier de description des configurations symboliques initiales.

Nous donnons maintenant la syntaxe que doivent vérifier ces deux fichiers.

7.2.1 Syntaxe des programmes

Dans le fichier correspondant au programme à analyser, l'utilisateur peut déclarer des types de structures de données, des variables globales et donner la définition de fonctions. La syntaxe des programmes pris en entrée par `TOPICS` est donnée par les figures 7.1 à 7.3.

Cette syntaxe a été définie dans le cadre du projet RNTL AVERILES [Ave] sur l'analyse et la vérification de logiciels embarqués avec structures de mémoire dynamique. Elle sert de base commune pour les différents outils développés dans le cadre de ce projet. Mis à part l'utilisation du symbole **any**, cette syntaxe correspond à une restriction du langage C. Nous avons introduit ce symbole de façon à pouvoir réaliser des tests non-déterministes, ce qui peut être utile pour donner à `TOPICS` un programme qui soit l'abstraction d'un autre programme. En effet, lorsque ce symbole est utilisé dans une garde, cela signifie que nous ne savons pas si la garde est vraie ou fausse, et nous supposons alors qu'elle peut être soit l'un soit l'autre.

Avec cette syntaxe, nous remarquons qu'un utilisateur peut déclarer des structures de données beaucoup plus complexes que des listes simplement chaînées ou aussi définir des fonctions récursives, toutefois `TOPICS` vérifiera si le programme donné en entrée vérifie certaines conditions que nous décrivons maintenant :

– Le programme ne peut comporter qu'au plus trois déclarations de type :
    1. Un type pour définir des listes simplement chaînées pour lesquelles chaque cellule ne contient qu'un seul champ permettant de stocker l'adresse de l'élément successeur,
    2. Un type pour déclarer des tableaux d'entiers,
    3. Un type pour déclarer des tableaux de listes simplement chaînées.
– Les fonctions définies ne doivent pas être récursives.

program : := { *declaration* }*

declaration : := *type-declaration*
| *var-declaration*
| *function-declaration*
| *function-definition*

type-declaration : := **typedef** *type-name* ***** *identifier* **;**
| **typedef struct** *identifier* **{** { struct-field }* **}** ***** *identifier* **;**

struct-field : := *type-name identifier* **;**
| **struct** *identifier* ***** *identifier* **;**

type-name : := **int**
| *identifier*

var-declaration : := *type-name identifier* { **,** *identifier* }* **;**

function-declaration : := *return-type identifier* **(** [*fpar* { **,** *fpar* }*] **)** **;**

return-type : := *type-name*
| **void**

fpar : := *type-name identifier*

function-definition : := *return-type identifier* **(** [*fpar* { **,** *fpar* }*] **)** *block*

FIGURE 7.1 – Déclarations

block : := **{** { *statement* } **}**

statement : := *var-declaration*
| /* empty */ **;**
| *lvalue-expression* **=** *rvalue-expression* **;**
| *identifier* **=** **malloc** **(** *malloc-expression* **)** **;**
| **free** **(** *identifier* **)** **;**
| [*lvalue-expression* **=**] *identifier* **(** [*term* { **,** *term* }*] **)** **;**
| **break** **;**
| **continue** **;**
| **goto** *label* **;**
| **return** [*term*] **;**
| **if** **(** *boolean-expression* **)** *statement*
| **if** **(** *boolean-expression* **)** *statement* **else** *statement*
| **while** **(** *boolean-expression* **)** *statement*
| *label* **:** *statement*
| *block*

FIGURE 7.2 – Instructions

malloc-expression	: :=	*size-expression-point*
	|	*integer* ***** *size-expression-tab*
	|	*size-expression-tab* ***** *integer*
size-expression-point	: :=	**sizeof** **(** **struct** *identifier* **)**
size-expression-tab	: :=	**sizeof** **(** *type-name* **)**
index-expression	: :=	*integer*
	|	*identifier*
lvalue-expression	: :=	*identifier*
	|	*identifier* **[** *index-expression* **]**
	|	*identifier* -> *identifier*
term	: :=	*lvalue-expression*
	|	*integer*
	|	**NULL**
rvalue-expression	: :=	*term*
	|	*term* **+** *term*
	|	*term* **-** *term*
	|	**any**
boolean-expression	: :=	*term* == *term*
	|	*term* **!=** *term*
	|	*term* <= *term*
	|	*term* >= *term*
	|	*term* < *term*
	|	*term* > *term*
	|	**any**
	|	**!** *boolean-expression*
	|	*boolean-expression* **&&** *boolean-expression*
	|	*boolean-expression* || *boolean-expression*
	|	**(** *boolean-expression* **)**

FIGURE 7.3 – Expressions

Lorsque ces conditions ne sont pas respectées, TOPICS le détecte et retourne alors un message d'erreur à l'utilisateur. TOPICS affichera un message d'erreur également lorsque le programme ne respectera pas la syntaxe d'entrée ou bien des règles classiques de programmation, comme par exemple lors de l'utilisation de variables non déclarées. Notons que TOPICS ne réalise pas de vérification de typage, et l'on suppose que le programme donné en entrée est correct à ce niveau là. Ceci dit un utilisateur peut vérifier ce point en utilisant un compilateur pour le C.

```
typedef struct List {
  struct List *next;
}* Liste;

void main(){
  Liste y,z;
  y=NULL;
  while(any){
    z=malloc(sizeof(struct List));
    z->next=y;
    y=z;
  }
  y=reverse(z);
}

Liste reverse(Liste x){
  Liste u,v;
  u=NULL;
  while(!(x==NULL)){
    v=x;
    x=x->next;
    v->next=u;
    u=v;
  }
  return u;
}
```

FIGURE 7.4 – Un programme C dans la syntaxe de TOPICS

Exemple 7.1 *La figure 7.4 donne un exemple de programmes écrit dans la syntaxe de* TOPICS *. Le symbole* ***any*** *est ici utilisé dans la boucle pour créer des listes de taille quelconque, ainsi la boucle pourra être prise un nombre indéterminé de fois. Après avoir créé une liste de taille quelconque, la fonction* `main` *appelle la fonction* `reverse` *sur cette liste.*

Étant donné qu'il peut y avoir plusieurs fonctions dans un programme donné en entrée de TOPICS , l'utilisateur doit spécifier quelle est la fonction qu'il souhaite analyser. Ainsi pour le programme présenté à la figure 7.4, l'utilisateur pourra choisir d'analyser la fonction `reverse` ou bien la fonction `main`.

7.2.2 Syntaxe pour les configurations initiales

La fonction donnée en entrée de TOPICS peut avoir des arguments, c'est pourquoi l'utilisateur doit décrire la configuration initiale du programme. La syntaxe du langage permettant de décrire les configurations initiales est donnée par la figure 7.5. Avec ce langage, l'utilisateur a la possibilité de donner les valeurs initiales des différents pointeurs, tableaux et entiers du programme. Il peut également utiliser des compteurs dont la valeur est implicitement strictement positive. Pour décrire le graphe initial, l'utilisateur déclare des noeuds ou des noeuds abstraits qui sont étiquetés par des compteurs. Ces compteurs ont alors le même rôle que dans les formes mémoire (à savoir qu'un noeud abstrait étiqueté par k, correspond à une succession de k noeuds). Il peut ensuite décrire des contraintes sur ces compteurs en disant que les valeurs de deux compteurs sont égales ou que la valeur d'un compteur est strictement plus grande qu'un entier. En ce qui concerne les tableaux, il peut dire quelle est la taille d'un tableau et définir ces éléments. Dans le cas des tableaux d'entiers, l'utilisateur peut aussi utiliser des compteurs, si il souhaite spécifier que la valeur d'un élément est n'importe quelle valeur strictement positive. De la même façon, l'utilisateur peut associer un compteur à une variable entière. Grâce à ces compteurs, l'utilisateur peut vérifier des programmes de façon paramétrée.

Exemple 7.2 *Par exemple, pour décrire que la variable de pointeurs* x *pointe vers une liste de taille quelconque paire terminant sur l'adresse* NULL*, nous utiliserons la description suivante :*

```
counter k;
abstract node n[k];
abstract node n2[k];
pointer x->n;
succ n=n2;
succ n2=NULL;
```

Et pour décrire une configuration initiale dans laquelle le tableau t *est de taille* 3*, dont chacun des éléments est un entier quelconque strictement supérieur à sa position dans le tableau, nous pouvons utiliser le fichier de configuration initiale suivant :*

```
counter k0;
counter k1;
counter k2;
int tab size t=3;
value t[0]=k0;
value t[1]=k1;
value t[2]=k2;
constraint k0>0;
constraint k1>1;
constraint k2>2;
```

Si le fichier de configuration initiale utilise des variables d'entiers, de tableaux ou de pointeurs qui ne sont pas des variables du programme donné en entrée, TOPICS affichera un message d'erreur. Cela arrivera également si le graphe pour décrire le tas mémoire contient des noeuds non accessibles par une variable, ou si des indices utilisés dans un tableau sont plus grands que la taille du tableau.

configuration : := $\left\{ \textit{description} \right\}^{*}$
description : := *counter-declaration*
| *node-delaration*
| *abstract-node-declaration*
| *succ-declaration*
| *pointer-declaration*
| *tab-int-declaration*
| *tab-list-declaration*
| *tab-declaration*
| *value-declaration*
| *constraint-declaration*
counter-declaration : := **counter** *identifier* **;**
node-delaration : := **node** *identifier* **;**
abstract-node-declaration : := **abstract** **node** *identifier* **[** *identifier* **]** **;**
succ-declaration : := **succ** *identifier* **=** *identifier* **;**
| **succ** *identifier* **=** **NULL** **;**
pointer-declaration : := **pointer** *identifier* **->** *identifier* **;**
| **pointer** *identifier* **->** **NULL** **;**
tab-int-declaration : := **int** **tab** **size** *identifier* **=** *integer* **;**
tab-list-declaration : := **list** **tab** **size** *identifier* **=** *integer* **;**
tab-declaration : := **tab** *identifier* **=** *identifier* **;**
value-declaration : := **value** *identifier* **[** *integer* **]** **=** *identifier* **;**
| **value** *identifier* **[** *integer* **]** **=** *integer* **;**
| **value** *identifier* **[** *integer* **]** **=** **NULL** **;**
| **value** *identifier* **=** *identifier* **;**
| **value** *identifier* **=** *integer* **;**
constraint-declaration : := **constraint** *identifier* **≥** *integer* **;**
| **constraint** *identifier* **=** *identifier* **;**

FIGURE 7.5 – Syntaxe de TOPICS pour la configuration initiale

7.3 Fonctionnement

À partir d'un programme écrit dans le fragment syntaxique du C défini auparavant, du nom d'une fonction du programme à analyser et d'un fichier de configuration initiale, `TOPICS` réalise les opérations suivantes :

1. Il construit le graphe de flots de contrôle de la fonction donnée en entrée en insérant les graphes de flots de contrôle des différentes fonctions appelées. Comme il n'y a pas d'appel de fonctions récursives, cette opération est possible. On obtient alors un système à pointeurs étendu avec des actions sur les entiers et des actions sur les tableaux. `TOPICS` produit de plus une représentation de ce système à pointeurs étendu au format `dot` de façon à permettre à l'utilisateur de le visualiser.
2. À partir du fichier de configuration initiale, `TOPICS` construit une forme mémoire et une formule de Presburger représentant l'ensemble des configurations initiales du système à pointeurs étendu.
3. En utilisant l'algorithme de traduction présenté au chapitre 5, `TOPICS` construit un système à compteurs, qui nous le rappelons est bisimilaire au système à pointeurs donné. Il produit alors un fichier au format `dot` qui permet de visualiser le système à compteurs obtenu ainsi que les formes mémoire associées à chaque état de contrôle. Il produit aussi des fichiers au format de `FAST` et d'`ASPIC`. Le système à compteurs obtenu peut avoir quatre états de contrôle spéciaux :
   (a) l'état `SegF` qui est accessible si le programme initialisé réalise une erreur de segmentation,
   (b) l'état `MemL` qui est accessible si le programme initialisé réalise une fuite mémoire,
   (c) l'état `OOBound` qui est accessible si le programme initialisé réalise un débordement d'indices sur un tableau,
   (d) l'état `Undef` qui est accessible si le programme initialisé teste des variables dont la valeur n'a pas été définie auparavant.

   Les fichiers au format de `FAST` et d'`ASPIC` contiennent alors une propriété à vérifier qui consiste à savoir si ces états sont accessibles à partir de l'ensemble de configurations initiales donné en entrée.

Comme nous l'avons signalé au chapitre 5, l'analyse du programme fourni en entrée de `TOPICS` se fait en deux phases, car après la traduction, on peut déjà obtenir des informations. En effet, si un des états spéciaux décrits précédemment n'est pas présent dans le système à compteurs produit alors on sait que l'erreur à laquelle il correspond ne sera pas réalisée. Par exemple, si l'état `SegF` n'existe pas dans le système à compteurs, alors le programme ne réalise pas d'erreur de segmentation. En revanche, si un état correspondant à une erreur est présent, il est nécessaire de lancer l'analyse du système à compteurs pour vérifier si cet état est accessible ou non. Finalement remarquons que la traduction présentée au chapitre 5 ne mentionne pas comment traiter les opérations sur des entiers ou sur des tableaux. En ce qui concerne les opérations sur les variables entières, la traduction implantée ne fait que les insérer directement dans le système à compteurs produit. Pour l'instant, les opérations sur les tableaux sont quant à elles traduites en considérant chaque élément du tableau comme une variable, ce qui est possible car les tableaux ont une taille fixe connue. Notons que cette méthode permettant de traiter les tableaux n'est pas efficace car elle peut générer un grand nombre de variables.

Exemple 7.3 *La figure 7.6 montre le système à pointeurs étendu obtenu avec `TOPICS` à partir de la fonction* `main` *du programme donné à la figure 7.4. Cette représentation graphique a été générée grâce à l'outil `GRAPHVIZ`.*

7.4 Résultats expérimentaux

Le tableau de la figure 7.8 donne les résultats expérimentaux que nous avons eu pour des programmes classiques manipulant des listes simplement chaînées. Ces différentes études de cas sont disponibles à l'adresse :
`http://www.lsv.ens-cachan.fr/~sangnier/TOPICS/`

Pour réaliser l'analyse des systèmes à compteurs avec `FAST` , nous avons utilisé le plugin `LASH` . Il s'avère que dans la plupart des cas, le calcul avec `ASPIC` est plus rapide qu'avec `FAST` , ceci car `ASPIC` calcule une surapproximation de l'ensemble d'accessibilité, alors que `FAST` calcule cet ensemble de façon exacte. Remarquons que l'utilisation de ces deux outils est assez complémentaire, car dans certains cas `FAST` prend trop de temps (ou ne termine pas) pour réaliser le calcul, comme c'est le cas par exemple avec le programme `merge`, mais dans d'autres cas `ASPIC` n'est pas capable de dire si un des états d'erreur est accessible ou non, comme avec le programme `doubleFree`, pour lequel il ne peut pas dire si l'état `SegF` est accessible. Finalement remarquons que l'une des principales difficultés posées par la traduction est qu'elle peut produire des systèmes à compteurs avec un grand nombre d'états de contrôle. Ceci car le nombre de formes mémoire (que l'on trouve dans les états de contrôle) est borné, mais par une fonction exponentielle dans le nombre de variables (cf. proposition 4.12). Toutefois, bon nombre de ces états de contrôle ne sont pas accessibles dans le système à compteurs, une solution pour améliorer le fonctionnement de `TOPICS` pourrait donc être d'essayer de diminuer ce nombre d'états de contrôle avant d'analyser les systèmes à compteurs avec `FAST` ou `ASPIC` .

Parmi les études de cas réalisées, le cas du programme `doubleFree` est particulièrement intéressant car la vérification de ce programme utilise pleinement le fait que l'on puisse exprimer des propriétés sur la taille des listes. Ce programme est donné à la figure 7.7. Nous remarquons que ce programme réalise une erreur de segmentation si on lui donne en entrée une liste acyclique de taille quelconque pointée par `x`. Ceci est dû au fait qu'à chaque itération, on libère le noeud pointé par `x` et son successeur en ne testant qu'une seule fois si `x` pointe ou non vers le noeud `NULL`. Toutefois, si l'on considère des listes de taille paire en entrée, ce programme ne réalise pas d'erreur. Nous avons pu vérifier cette propriété avec `TOPICS` .

Conclusion

Dans ce chapitre, nous avons présenté l'outil `TOPICS` implantant la traduction des systèmes à pointeurs vers les systèmes à compteurs présentée au chapitre 5. Cet outil nous a permis de constater que les résultats théoriques présentés permettaient effectivement de vérifier des programmes manipulant des listes simplement chaînées. Toutefois l'implantation peut encore être améliorée de façon à pouvoir traiter des programmes de plus grande taille.

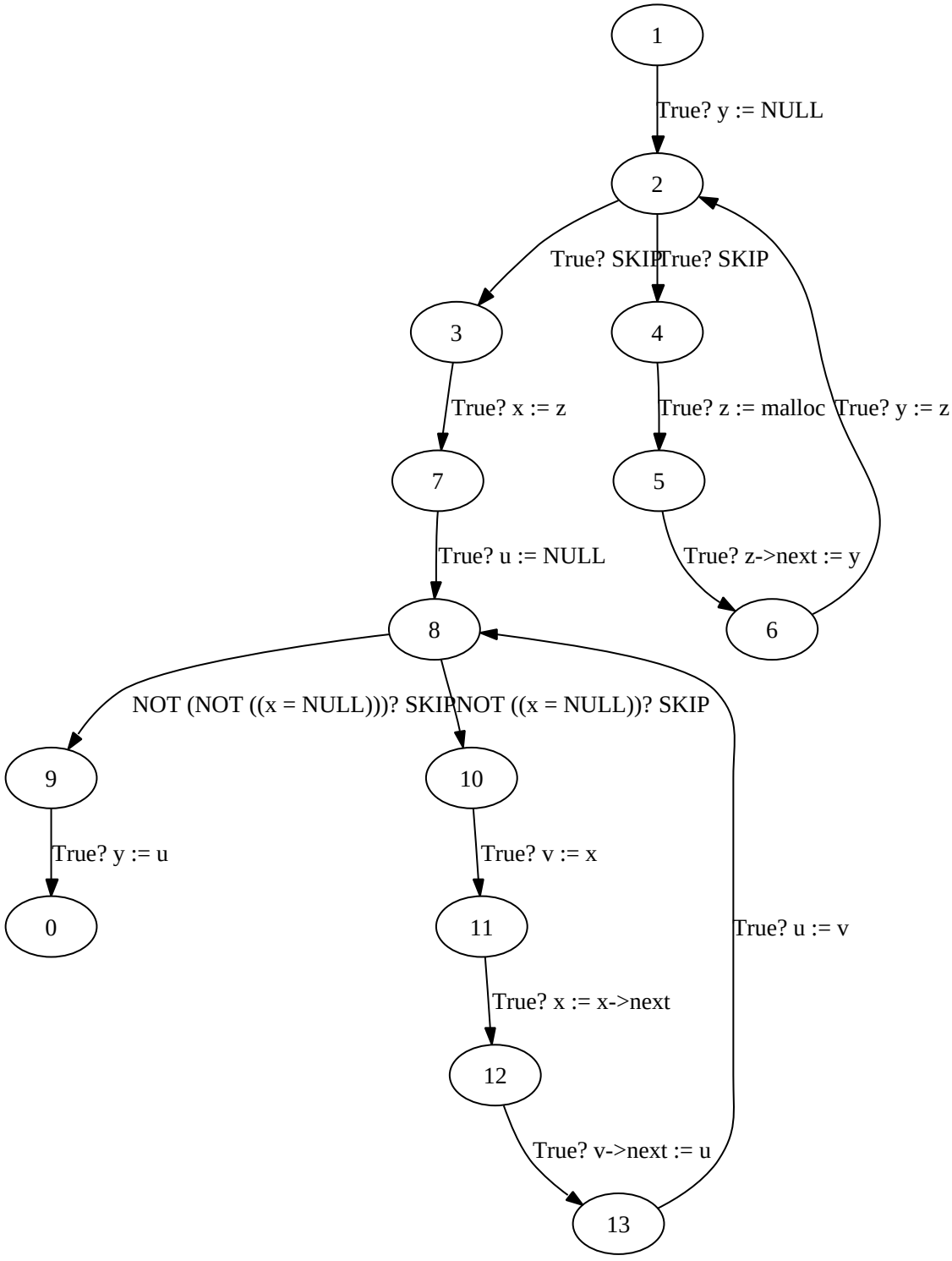


FIGURE 7.6 – Système à pointeurs étendu pour la fonction `main` de la figure 7.4

```
typedef struct List {
  struct List *next;
}* Liste;

void doubleFree(Liste x){
  Liste y;
  while(x!=NULL){
    y=x;
    x=x->next;
    free(y);
    y=x;
    x=x->next;
    free(y);
  }
}
```

FIGURE 7.7 – Le programme `doubleFree`

Etude de cas	**Nb. états du système à pointeurs étendu**	**Nb. états du système à compteurs**	**Résultats après traduction**	**Résultats ASPIC**	**Résultats FAST**
`create`	7	14	No SegF No Undef No OOBound	No MemL	No MemL
`insert`	19	44	No SegF No Undef No OOBound	No MemL	
`reverse`	8	19	No SegF No Undef No OOBound	No MemL	No MemL
`deleteAll`	5	8	No SegF No Undef No OOBound	No MemL	No MemL
`merge`	20	532	No OOBound No Undef	No SegF No MemL	
`mainReverse`	14	45	No SegF No OOBound	No MemL	
`doubleFree`	8	15	No OOBound No Undef	No MemL	No MemL SegF
`doubleFree` listes paires	7	15	No OOBound No Undef	No MemL	No MemL No SegF

FIGURE 7.8 – Résultats obtenus avec `TOPICS`

Conclusion générale

Bilan

Dans cette thèse, nous avons étudié le problème de la vérification pour deux classes de systèmes infinis, les systèmes à compteurs et les systèmes à pointeurs. Pour chacune de ces classes de systèmes, nous avons dans un premier temps proposé des méthodes pour résoudre des problèmes d'accessibilité symboliques. Nous avons ensuite défini des logiques temporelles pour décrire les exécutions de ces systèmes et permettant en particulier de parler de l'évolution des configurations. Nous avons alors étudié la décidabilité de problèmes de model-checking pour ces logiques temporelles.

En ce qui concerne les systèmes à compteurs, nous avons défini une nouvelle classe de systèmes ayant un ensemble de configurations accessibles effectivement définissables dans l'arithmétique de Presburger. Cette propriété est particulièrement intéressante car elle permet d'obtenir des algorithmes pour résoudre le problèmes d'accessibilité symbolique. Cette classe étend la classe des machines à compteurs *reversal*-bornées introduites par Ibarra dans [Iba78]. Nous avons également étudié le problème de savoir si l'on pouvait décider si une machine à compteurs appartenait ou non à cette classe et nous avons vu que ce problème est en général indécidable sauf si la machine à compteurs considérée est un SAVE (ou un réseau de Petri), c'est-à-dire une machine dans laquelle on ne peut pas tester si la valeur d'un compteur est égale à une constante.
Nous avons ensuite étudié le model-checking d'automates à un compteur avec des formules de logiques pour les mots de données. Nous avons considéré deux logiques, la logique temporelle linéaire avec registres et la logique du premier ordre sur les mots de données. De récents travaux étudiaient principalement des problèmes de satisfiabilité pour ces logiques, et à notre connaissance c'est la première fois que ces logiques sont utilisées pour spécifier les exécutions d'un modèle opérationnel. Nous nous sommes restreint au cadre des automates à un compteur car ces logiques permettent d'exprimer des propriétés indécidables sur les machines à deux compteurs, en particulier l'accessibilité d'un état de contrôle. Bien que les automates à un compteur soient un modèle très simple, nous avons vu que dans le cas général le problème de model-checking de ces logiques est indécidable, mais qu'il devient décidable pour des automates à un compteur déterministes.

Les systèmes à pointeurs que nous avons introduits dans la deuxième partie de cette thèse servent à modéliser des programmes manipulant des listes simplement chaînées. Nous avons défini pour ces systèmes une représentation symbolique permettant de décrire des propriétés paramétrées sur des programmes. Par exemple, nous pouvons exprimer le fait qu'un programme ne réalise pas d'erreur de segmentation quelle que soit la taille de la liste simplement chaînée donnée en entrée. Nous avons présenté une méthode pour analyser ces systèmes en les traduisant vers un système à compteurs bisimilaire. Grâce à cela, nous pouvons réutiliser les algorithmes et les outils permettant d'analyser les systèmes à compteurs pour vérifier le bon fonctionnement de programmes avec listes.

Nous avons ensuite étudié la décidabilité de problèmes d'accessibilité et de model-checking de formules de logiques temporelles pour ces systèmes. Il s'avère que même dans le cas très simple de systèmes à pointeurs plats sans mise à jour destructive, le problème d'accessibilité d'une erreur de segmentation est indécidable, mais devient décidable si nous supposons que les configurations initiales sont acycliques. Nous avons ainsi pu constaté qu'il était ardu de trouver une sous-classe intéressante de systèmes à pointeurs pour laquelle les problèmes d'accessibilité sont décidables.

Pour finir, nous avons décrit le fonctionnement de l'outil TOPICS qui implante l'algorithme de traduction des systèmes à pointeurs vers des systèmes à compteurs. Cet outil nous a permis de constater sur des exemples de taille raisonnable que cette méthode permet effectivement de vérifier des programmes manipulant dynamiquement la mémoire. Toutefois pour pouvoir traiter des exemples de plus grande taille, il faudrait développer des méthodes permettant de réduire la taille du système à compteurs produit.

Perspectives

Comme cela apparaît dans les conclusions des différents chapitres, cette thèse ouvre la voie à de nombreuses directions de recherche.

Nous avons vu au deuxième chapitre que l'algorithme implanté dans l'outil FAST permet de calculer l'ensemble d'accessibilité d'une machine à compteurs *reversal*-bornée, mais il pourrait être intéressant de développer un outil spécifique permettant l'analyse de ces machines. Cet outil pourrait comme FAST analyser également des machines à compteurs qui ne sont pas nécessairement *reversal*-bornées en calculant les ensembles d'accessibilité de sous-systèmes *reversal*-bornés. De plus, un tel outil pourrait aussi être utilisé pour calculer la formule de Presburger correspondant à l'image de Parikh de langages réguliers. En effet, si dans un automate fini on ajoute des compteurs comptant le nombre de lettres lues, on obtient une machine à compteurs *reversal*-bornée.

Une autre direction possible concerne l'étude de problèmes de model-checking de logiques sur les mots de données en considérant d'autres modèles que les automates à un compteur. En particulier, le problème du model-checking de la logique LTL avec registres sur des machines à compteurs *reversal*-bornées reste ouvert.

En ce qui concerne l'analyse de systèmes à pointeurs, une direction possible de recherche consiste à développer une méthode similaire de traduction vers des systèmes à compteurs pour des programmes utilisant des structures de données plus complexes que des listes simplement chaînées comme par exemple des listes doublement chaînées. Cette méthode pourrait ensuite être également utilisée par TOPICS.

Finalement, une perspective plus générale de recherche dans le cadre du model-checking de systèmes infinis pourrait être d'étendre la classe des systèmes à compteurs en leur ajoutant des formalismes pris dans d'autres modèles. Cela permettrait de voir si les classes décidables de systèmes à compteurs peuvent être étendues tout en gardant la décidabilité. Deux classes de systèmes semblent particulièrement intéressantes, la classe des systèmes à compteurs temporisés [BFS08] qui correspondent à des systèmes à compteurs munis d'horloges, et la classe des systèmes à compteurs probabilistes, dans lesquels des probabilités sont utilisées pour modéliser la chance qu'une transition a d'être franchise.

Bibliographie

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

[AH94] Rajeev Alur and Thomas A. Henzinger. A really temporal logic. *Journal of the ACM*, 41(1) :181–204, 1994.

[AJ96] Parosh Aziz Abdulla and Bengt Jonsson. Verifying programs with unreliable channels. *Information and Computation*, 127(2) :91–101, 1996.

[AK77] Toshiro Araki and Tadao Kasami. Decidable problems on the strong connectivity of Petri net reachability sets. *Theoretical Computer Science*, 4(1) :99–119, 1977.

[Ave] Projet RNTL AVERILES. http ://www.lsv.ens-cachan.fr/rntl-averiles/.

[Bar05] Sébastien Bardin. *Vers un model checking avec accélération plate de systèmes hétérogènes*. Thèse de doctorat, Laboratoire Spécification et Vérification, ENS Cachan, France, 2005.

[BBH+06] Ahmed Bouajjani, Marius Bozga, Peter Habermehl, Radu Iosif, Pierre Moro, and Tomáš Vojnar. Programs with lists are counter automata. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 517–531. Springer, 2006.

[BCC+07] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07)*, volume 4590 of *Lecture Notes in Computer Science*. Springer, 2007.

[BCO05a] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot : Modular automatic assertion checking with separation logic. In *Proceedings of the 4th International Symposium on Formal Methods for Components and Objects (FMCO'05)*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2005.

[BCO05b] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems (APLAS'05)*, volume 3780 of *Lecture Notes in Computer Science*, pages 52–68. Springer, 2005.

[BDM+06] Mikolaj Bojańczyk, Claire David, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. Two-variable logic on data trees and XML reasoning. In *Proceedings of the 25th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'06)*, pages 10–19. ACM, 2006.

[BEM97] Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata : Application to model-checking. In *Proceedings of the 8th International*

Conference on Concurrency Theory (CONCUR'97), volume 1243 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 1997.

[BFLP03] Sébastien Bardin, Alain Finkel, Jérôme Leroux, and Laure Petrucci. Fast : Fast acceleration of symbolic transition systems. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV'03)*, volume 2725 of *Lecture Notes in Computer Science*, pages 118–121. Springer, 2003.

[BFLP08] Sébastien Bardin, Alain Finkel, Jérôme Leroux, and Laure Petrucci. FAST : Acceleration from theory to practice. *International Journal on Software Tools for Technology Transfer*, 2008. To appear.

[BFLS05] Sébastien Bardin, Alain Finkel, Jérôme Leroux, and Philippe Schnoebelen. Flat acceleration in symbolic model checking. In *Proceedings of the 3rd International Symposium on Automated Technology for Verification and Analysis (ATVA'05)*, volume 3707 of *Lecture Notes in Computer Science*, pages 474–488. Springer, 2005.

[BFLS06] Sébastien Bardin, Alain Finkel, Étienne Lozes, and Arnaud Sangnier. From pointer systems to counter systems using shape analysis. In *5th International Workshop on Automated Verification of Infinite-State Systems (AVIS'06)*, Vienna, Austria, 2006.

[BFN04] Sébastien Bardin, Alain Finkel, and David Nowak. Toward symbolic verification of programs handling pointers. In *3rd International Workshop on Automated Verification of Infinite-State Systems (AVIS'04)*, 2004.

[BFS08] Florent Bouchy, Alain Finkel, and Arnaud Sangnier. Reachability in timed counter systems. In Peter Habermehl and Tomáš Vojnar, editors, *Proceedings of the 10th International Workshop on Verification of Infinite State Systems (INFINITY'08)*, Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers, 2008. To appear.

[BHMV05] Ahmed Bouajjani, Peter Habermehl, Pierre Moro, and Tomáš Vojnar. Verifying programs with dynamic 1-selector-linked structures in regular model checking. In *Proceedings of the 11th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'05)*, volume 3440 of *Lecture Notes in Computer Science*, pages 13–29. Springer, 2005.

[BHRV06a] Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz, and Tomáš Vojnar. Abstract regular tree model checking. In *Proceedings of the 7th International Workshop on Verification of Infinite State Systems (INFINITY'05)*, volume 149 of *Electronic Notes in Theoretical Computer Science*, pages 37–48. Elsevier Science Publishers, 2006.

[BHRV06b] Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz, and Tomáš Vojnar. Abstract regular tree model checking of complex dynamic data structures. In *Proceedings of the 13th International Symposium Static Analysis (SAS'06)*, volume 4134 of *Lecture Notes in Computer Science*, pages 52–70. Springer, 2006.

[BHV04] Ahmed Bouajjani, Peter Habermehl, and Tomáš Vojnar. Abstract regular model checking. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV'04)*, volume 3114 of *Lecture Notes in Computer Science*, pages 372–386. Springer, 2004.

[BI06] Marius Bozga and Radu Iosif. Quantitative verification of programs with lists. In *Proceedings of the NATO Advanced Research Workshop on Verification of Infinite-State Systems with Applications to Security (VISSAS'05)*, volume 1 of *NATO Security through Science Series D : Information and Communication Security*. IOS Press, 2006.

[BI07] Marius Bozga and Radu Iosif. On flat programs with lists. In *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'07)*, volume 4349 of *Lecture Notes in Computer Science*, pages 122–136. Springer, 2007.

[BIL04] Marius Bozga, Radu Iosif, and Yassine Lakhnech. On logics of aliasing. In *Proceedings of the 11th International Symposium on Static Analysis (SAS'04)*, volume 3148 of *Lecture Notes in Computer Science*, pages 344–360. Springer, 2004.

[BJNT00] Ahmed Bouajjani, Bengt Jonsson, Marcus Nilsson, and Tayssir Touili. Regular model checking. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *Lecture Notes in Computer Science*, pages 403–418. Springer, 2000.

[BLARS07] Igor Bogudlov, Tal Lev-Ami, Thomas W. Reps, and Mooly Sagiv. Revamping tvla : Making parametric shape analysis competitive. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 221–225. Springer, 2007.

[BLP06] Sébastien Bardin, Jérôme Leroux, and Gérald Point. FAST extended release. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 63–66, 2006.

[BMMR01] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of c programs. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI'01)*, pages 203–213. ACM, 2001.

[BMS+06] Mikolaj Bojańczyk, Anca Muscholl, Thomas Schwentick, Luc Segoufin, and Claire David. Two-variable logic on words with data. In *Proceedings of the 21st Annual IEEE Symposium on Logic in Computer Science (LICS'06)*, pages 7–16. IEEE Computer Society Press, 2006.

[BR01] Thomas Ball and Sriram K. Rajamani. The slam toolkit. In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 260–264. Springer, 2001.

[Cau92] Didier Caucal. On the regular structure of prefix rewriting. *Theoretical Computer Science*, 106(1) :61–86, 1992.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles Of Programming Languages (POPL'77)*, pages 238–252. ACM, 1977.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop, 1981*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1982.

[CFP96] Gérard Cécé, Alain Finkel, and S. Purushothaman Iyer. Unreliable channels are easier to verify than perfect channels. *Information and Computation*, 124(1) :20–31, 1996.

[CJ98] Hubert Comon and Yan Jurski. Multiple counters automata, safety analysis and Presburger arithmetic. In *Proceedings of the 10th International Conference on Computer Aided Verification (CAV'98)*, volume 1427 of *Lecture Notes in Computer Science*, pages 268–279. Springer, 1998.

[CLM76] E. Cardoza, Richard J. Lipton, and Albert R. Meyer. Exponential space complete problems for petri nets and commutative semigroups : Preliminary report. In *Proceedings of the 8th Annual ACM Symposium on Theory of Computing (STOC'76)*, pages 50–54. ACM, 1976.

[Cup] CUP - LALR Parser Generator in Java. http ://www2.cs.tum.edu/projects/cup/.

[DEG06] Jyotirmoy V. Deshmukh, E. Allen Emerson, and Prateek Gupta. Automatic verification of parameterized data structures. In *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, volume 3920 of *Lecture Notes in Computer Science*, pages 27–41. Springer, 2006.

[DFGD06] Stéphane Demri, Alain Finkel, Valentin Goranko, and Govert van Drimmelen. Towards a model-checker for counter systems. In *Proceedings of the 4th International Symposium on Automated Technology for Verification and Analysis (ATVA'06)*, volume 4218 of *Lecture Notes in Computer Science*, pages 493–507. Springer, 2006.

[DIP01] Zhe Dang, Oscar H. Ibarra, and Pierluigi San Pietro. Liveness verification of reversal-bounded multicounter machines with a free counter. In *Proceedings of the 21st Conference on Fundations of Software Technology and Theoretical Computer Science (FSTTCS'01)*, volume 2245 of *Lecture Notes in Computer Science*, pages 132–143. Springer, 2001.

[DL06] Stéphane Demri and Ranko Lazic. LTL with the freeze quantifier and register automata. In *Proceedings of the 21st Annual IEEE Symposium on Logic in Computer Science (LICS'06)*, pages 17–26. IEEE Computer Society Press, 2006.

[DL08] Stéphane Demri and Ranko Lazić. LTL with the freeze quantifier and register automata. *ACM Transactions on Computational Logic*, 2008. To appear.

[DLN05] Stéphane Demri, Ranko Lazić, and David Nowak. On the freeze quantifier in constraint LTL : Decidability and complexity. In *Proceedings of the 12th International Symposium on Temporal Representation and Reasoning (TIME'05)*, pages 113–121. IEEE Computer Society Press, 2005.

[DLN07] Stéphane Demri, Ranko Lazić, and David Nowak. On the freeze quantifier in constraint LTL : Decidability and complexity. *Information and Computation*, 205(1) :2–24, 2007.

[DLS08] Stéphane Demri, Ranko Lazić, and Arnaud Sangnier. Model checking freeze LTL over one-counter automata. In *Proceedings of the 11th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS'08)*, volume 4962 of *Lecture Notes in Computer Science*, pages 490–504. Springer, 2008.

[DOY06] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, volume 3920 of *Lecture Notes in Computer Science*, pages 287–302. Springer, 2006.

[EH83] E. Allen Emerson and Joseph Y. Halpern. "sometimes" and "not never" revisited : On branching versus linear time. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles Of Programming Languages (POPL'83)*, pages 127–140. ACM, 1983.

[EMS00] Jacob Elgaard, Anders Møller, and Michael I. Schwartzbach. Compile-time debugging of C programs working on trees. In *Proceedings of the 9th European Symposium on*

Programming (ESOP'00), volume 1782 of *Lecture Notes in Computer Science*, pages 119–134. Springer, 2000.

[EN94] Javier Esparza and Mogens Nielsen. Decidability issues for Petri nets - a survey. *Elektronische Informationsverarbeitung und Kybernetik*, 30(3) :143–160, 1994.

[Esp97] Javier Esparza. Petri nets, commutative context-free grammars, and basic parallel processes. *Fundamenta Informaticae*, 31(1) :13–25, 1997.

[Fin94] Alain Finkel. Decidability of the termination problem for completely specified protocols. *Distributed Computing*, 7(3) :129–135, 1994.

[FL02] Alain Finkel and Jérôme Leroux. How to compose presburger-accelerations : Applications to broadcast protocols. In *Proceedings of the 22nd Conference on Fundations of Software Technology and Theoretical Computer Science (FSTTCS'02)*, volume 2556 of *Lecture Notes in Computer Science*, pages 145–156. Springer, 2002.

[FLS07] Alain Finkel, Étienne Lozes, and Arnaud Sangnier. Towards model checking pointer systems. In Benedikt Löwe, editor, *Proceedings of the International Conference on Infinity in Logic & Computation (ILC'07)*, 2007.

[FS08] Alain Finkel and Arnaud Sangnier. Reversal-bounded counter machines revisited. In *Proceedings of the 33rd International Symposium on Mathematical Fundations of Computer Science (MFCS'08)*, volume 5162 of *Lecture Notes in Computer Science*, pages 323–334. Springer, 2008.

[Gab81] Dov M. Gabbay. Expressive functional completeness in tense logic. In *Aspects of Philosophical Logic*, pages 91–117. Reidel, 1981.

[Gas07] Régis Gascon. *Spécification et vérification de propriétés quantitatives sur des automates à contraintes*. Thèse de doctorat, Laboratoire Spécification et Vérification, ENS Cachan, France, 2007.

[GH06] Laure Gonnord and Nicolas Halbwachs. Combining widening and acceleration in linear relation analysis. In *13th Iternational Conference Symposium Static Analysis*, volume 4134 of *Lecture Notes in Computer Science*, pages 144–160. Springer, 2006.

[GS66] Seymour Ginsburg and Edwin H. Spanier. Semigroups, presburger formulas, and languages. *Pacific Journal of Mathematics*, 16(2) :285–296, 1966.

[Hig52] Graham Higman. Ordering by divisibility in abstract algebras. *Proceedings of the London Mathematical Society* (3), 2 :326–336, 1952.

[HIV06] Peter Habermehl, Radu Iosif, and Tomáš Vojnar. Automata-based verification of programs with tree updates. In *Proceedings of the 12th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'06)*, volume 3920 of *Lecture Notes in Computer Science*, pages 350–364. Springer, 2006.

[HP79] John E. Hopcroft and Jean-Jacques Pansiot. On the reachability problem for 5-dimensional vector addition systems. *Theoretical Computer Science*, 8 :135–159, 1979.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[Iba78] Oscar H. Ibarra. Reversal-bounded multicounter machines and their decision problems. *Journal of the ACM*, 25(1) :116–133, 1978.

[ISD+02] Oscar H. Ibarra, Jianwen Su, Zhe Dang, Tevfik Bultan, and Richard A. Kemmerer. Counter machines and verification problems. *Theoretical Computer Science*, 289(1) :165–189, 2002.

[Jan87] Matthias Jantzen. Complexity of place/transition nets. In *Petri Nets : Central Models and Their Properties, Advances in Petri Nets 1986, Part I, Proceedings of an Advanced Course*, volume 254 of *Lecture Notes in Computer Science*, pages 413–434. Springer, 1987.

[JFl] JFlex - The Fast Scanner Generator for Java. http ://jflex.de/.

[JJKS97] Jakob L. Jensen, Michael E. Jørgensen, Nils Klarlund, and Michael I. Schwartzbach. Automatic verification of pointer programs using monadic second-order logic. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI'97)*, pages 226–236. ACM, 1997.

[JL07] Marcin Jurdziński and Ranko Lazić. Alternation-free modal mu-calculus for data trees. In *Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science (LICS'07)*, pages 131–140. IEEE Computer Society Press, 2007.

[Kle87] Stepehn Cole Kleene. *Introduction to Metamathematics*. North-Holland, second edition, 1987.

[KM69] Richard M. Karp and Raymond E. Miller. Parallel program schemata. *Journal of Computer System Sciences*, 3(2) :147–195, 1969.

[KMM+97] Yonit Kesten, Oded Maler, Monica Marcus, Amir Pnueli, and Elad Shahar. Symbolic model checking with rich ssertional languages. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 424–435. Springer, 1997.

[KMM+01] Yonit Kesten, Oded Maler, Monica Marcus, Amir Pnueli, and Elad Shahar. Symbolic model checking with rich assertional languages. *Theoretical Computer Science*, 256(1-2) :93–112, 2001.

[Kos82] S. Rao Kosaraju. Decidability of reachability in vector addition systems (preliminary version). In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing (STOC'82)*, pages 267–281. ACM, 1982.

[Koy90] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4) :255–299, 1990.

[LAMS04] Tal Lev-Ami, Roman Manevich, and Shmuel Sagiv. Tvla : A system for generating abstract interpreters. In *Proceedings of the IFIP 18th World Computer Congress, Topical Sessions*, pages 367–376. Kluwer, 2004.

[Laz06] Ranko Lazic. Safely freezing LTL. In *Proceedings of the 26th Conference on Fundations of Software Technology and Theoretical Computer Science (FSTTCS'06)*, volume 4337 of *Lecture Notes in Computer Science*, pages 381–392. Springer, 2006.

[Ler03] Jérôme Leroux. *Algorithmique de la vérification des systèmes à compteurs. Approximation et accélération. Implémentation de l'outil FAST*. Thèse de doctorat, Laboratoire Spécification et Vérification, ENS Cachan, France, 2003.

[Lip78] Leonard Lipshitz. The diophantine problem for addition and divisibility. *Transactions of the American Mathematical Society*, 235 :271–283, 1978.

[LMS02] François Laroussinie, Nicolas Markey, and Philippe Schnoebelen. Temporal logic with forgettable past. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS'02)*, pages 383–392. IEEE Computer Society Press, 2002.

[LR78] Lawrence H. Landweber and Edward L. Robertson. Properties of conflict-free and persistent Petri nets. *Journal of the ACM*, 25(3) :352–364, 1978.

[LS05] Jérôme Leroux and Grégoire Sutre. Flat counter automata almost everywhere ! In *Proceedings of the 3rd International Symposium on Automated Technology for Verification and Analysis (ATVA'05)*, volume 3707 of *Lecture Notes in Computer Science*, pages 489–503. Springer, 2005.

[Mat70] Yuri Matiyasevich. Enumerable sets are diophantine. *Journal of Sovietic Mathematics*, 11 :354–357, 1970.

[May81] Ernst W. Mayr. Persistence of vector replacement systems is decidable. *Acta Informatica*, 15 :309–318, 1981.

[May84] Ernst W. Mayr. An algorithm for the general Petri net reachability problem. *SIAM Journal on Computing*, 13(3) :441–460, 1984.

[MBCC07] Stephen Magill, Josh Berdine, Edmund M. Clarke, and Byron Cook. Arithmetic strengthening for shape analysis. In *Proceedings of 14th International Symposium on Static Analysis (SAS'07)*, volume 4634 of *Lecture Notes in Computer Science*, pages 419–436. Springer, 2007.

[Min67] Marvin L. Minsky. *Computation : finite and infinite machines*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1967.

[MS77] Arnaldo Mandel and Imre Simon. On finite semigroups of matrices. *Theoretical Computer Science*, 5(2) :101–111, 1977.

[MS85] David E. Muller and Paul E. Schupp. The theory of ends, pushdown automata, and second-order logic. *Theoretical Computer Science*, 37 :51–75, 1985.

[MS01] Anders Møller and Michael I. Schwartzbach. The pointer assertion logic engine. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI'01)*, pages 221–231, 2001.

[MS03] Nicolas Markey and Philippe Schnoebelen. Model checking a path (preliminary report). In *Proceedings of the 14th International Conference on Concurrency Theory (CONCUR'03)*, volume 2761 of *Lecture Notes in Computer Science*, pages 251–265. Springer, 2003.

[OW06] Joël Ouaknine and James Worrell. On metric temporal logic and faulty turing machines. In *Proceedings of the 9th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS'06)*, pages 217–230, 2006.

[OW07] Joël Ouaknine and James Worrell. On the decidability and complexity of metric temporal logic over finite words. *Logical Methods in Computer Science*, 3(1), 2007.

[Par66] Rohit Parikh. On context-free languages. *Journal of the ACM*, 13(4) :570–581, 1966.

[Pet62] Carl Adam Petri. *Kommunikation mit Automaten.* Bonn : Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations Of Computer Science (FOCS'77)*, pages 46–57. IEEE Computer Society Press, 1977.

[Pol] Polyspace embedded software verification. http ://www.mathworks.com/products/polyspace/.

[Pre29] Mojzesz Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. *Comptes Rendus du premier congrès de mathématiciens des Pays Slaves, Warszawa*, pages 92–101, 1929.

[PRW08] Andreas Podelski, Andrey Rybalchenko, and Thomas Wies. Heap assumptions on demand. In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV'08)*, volume 5123 of *Lecture Notes in Computer Science*. Springer, 2008.

[PW05] Andreas Podelski and Thomas Wies. Boolean heaps. In *Proceedings of the 12th International Symposium on Static Analysis (SAS'05)*, volume 3672 of *Lecture Notes in Computer Science*, pages 268–283. Springer, 2005.

[Rac78] Charles Rackoff. The covering and boundedness problems for vector addition systems. *Theoretical Computer Science*, 6 :223–231, 1978.

[Reu89] Christophe Reutenauer. *Aspects mathématiques des réseaux de Petri*. Masson, Paris, France, 1989.

[Rey68] John C. Reynolds. Automatic computation of data set definitions. In *IFIP Congress (1)*, pages 456–461, 1968.

[Rey02] John C. Reynolds. Separation logic : A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS'02)*, pages 55–74. IEEE Computer Society, 2002.

[Sch02] Philippe Schnoebelen. Verifying lossy channel systems has nonprimitive recursive complexity. *Information Processing Letters*, 83(5) :251–261, 2002.

[Spi] SPIN. http ://spinroot.com/.

[SRW98] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages Systems (TOPLAS)*, 20(1) :1–50, 1998.

[SRW02] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages Systems (TOPLAS)*, 24(3) :217–298, 2002.

[Upp] UPPAAL. http ://www.uppaal.com/.

[VVN81] Rüdiger Valk and Guy Vidal-Naquet. Petri nets and regular languages. *Journal of Computer and System Sciences*, 23(3) :299–325, 1981.

[WB98] Pierre Wolper and Bernard Boigelot. Verifying systems with infinite but regular state spaces. In *Proceedings of the 10th International Conference on Computer Aided Verification (CAV'98)*, volume 1427 of *Lecture Notes in Computer Science*, pages 88–97. Springer, 1998.