Rémi Brochenin

# Techniques d'automates pour raisonner sur la mémoire

Rapport de stage M2 - encadré par
Stéphane Demri et Étienne Lozes

26 juin 2006

Laboratoire
Spécification
et
Vérification

CENTRE NATIONAL
DE LA RECHERCHE
SCIENTIFIQUE

Ecole Normale Supérieure de Cachan
61, avenue du Président Wilson
94235 Cachan Cedex France

# Techniques d'automates pour raisonner sur la mémoire

Rémi Brochenin,
encadré par Stéphane Demri et Étienne Lozes
dans un stage au LSV, ENS de Cachan

pour le Master 2 d'Informatique Fondamentale
de l'ENS Lyon et l'Université Claude Bernard

26 juin 2006

## Table des matières

**Première partie**

# La logique $\mathrm{LTL}^{\mathrm{mem}}$

## 1 Contexte

### 1.1 Problématique

Nous nous plaçons dans le contexte de la vérification de programmes. Les enjeux du domaine sont notamment la vérification de systèmes logiciels embarqués. Dans les cas industriels, les langages utilisés sont souvent de bas niveau, n'incluant que peu de mécanismes de détection automatique d'erreur ; par exemple les systèmes de types de C et Java sont très peu expressifs, et ne permettent pas de s'assurer correctement de l'utilisation de certaines structures de données récursives. Il apparaît donc utile de vérifier les programmes par un procédé indépendant, d'autant plus lorsque une erreur peut avoir des conséquences économiques sérieuses. Plus précisément, nous souhaitons étudier la vérification de programmes à pointeurs. En effet, la programmation avec gestion dynamique de la mémoire amène plusieurs types d'erreurs difficiles à détecter lors de la programmation. Une raison à cette difficulté est la présence d'aliasing, c'est-à-dire qu'un même objet peut être désigné par plusieurs noms. Plusieurs méthodes existent pour analyser de tels programmes.

Pour les programmes à pointeurs, il existe plusieurs types de propriétés que l'on peut vouloir vérifier. Tout d'abord, il est possible de s'intéresser à des problèmes concernant la sûreté, comme les fuites mémoire et violations mémoire, dont les effets peuvent être brutaux et provoquer par exemple l'arrêt du programme. Aussi des propriétés plus fortes, sur la forme de la mémoire et des structures de données qu'elle contient : la "shape analysis" vérifie par exemple que la mémoire contient une liste, ou bien un arbre. Ensuite, on peut vouloir exprimer des propriétés quantitatives, sur la conservation de la longueur d'une liste par exemple. Ou encore des propriétés structurelles précises, établissant par exemple qu'une liste est triée ou qu'un arbre Rouge-Noir est bien équilibré. Enfin, notons que la gestion dynamique de la mémoire peut être source de problèmes de sécurité, en ouvrant la voie à des attaques par débordement de tampon.

Pour exprimer ces propriétés, selon le programme vérifié, il peut être plus pertinent de choisir une représentation particulière de la mémoire, c'est-à-dire une abstraction de celle-ci. On peut s'intéresser à l'arithmétique des pointeurs, en considérant que la mémoire est constituée de tableaux, sans récursion. Il est aussi possible de contraindre la forme de la mémoire en se restreignant au cas des listes simples, en n'autorisant qu'un successeur. Mais il peut être nécessaire de s'intéresser à des structures de données complexes.

Enfin, il existe différents degrés d'automatisation des procédures, ce que nous allons maintenant voir.

## 1.2 Méthodes

Il existe donc plusieurs manières de considérer le problème de la vérification de programmes avec pointeurs. Nous retiendrons les méthodes suivantes. Nous les avons classées en deux catégories, semi-automatiques et automatiques, et à l'intérieur de ces catégories, les procédés sont classés informellement, en essayant d'obtenir un ordre croissant d'expressivité de leur formalisme.

D'une part, certains des procédés de vérification nécessitent une annotation idoine du programme vérifié, par l'utilisateur : ces procédés sont semi-automatiques, et en particulier, ne peuvent pas traiter les boucles sans la donnée d'un invariant. Ils sont basés sur des logiques d'états, c'est-à-dire des langages de description d'états mémoire. Ils fonctionnent, selon les cas, par calcul de précondition et postcondition. Parmi ceux-ci, citons la logique des Alias introduite dans [BIL03]. Il existe aussi la logique de séparation, introduite par Reynolds, autorisant un raisonnement local et dont les annotations sont de type logique de Hoare, sur laquelle nous nous attarderons dans la prochaine section.

Dans cette catégorie doit aussi être classé un outil basé sur la logique monadique du second ordre et nommé PALE ("Pointer Assertion Logic Engine"), présenté dans [JJKS97] et [MS01]. Les propriétés vérifiées sont plus fortes, mais la mémoire ne peut pas prendre n'importe quelle structure : il est considéré comme acquis que la mémoire est constituée d'arbres ou de listes, et garde cette structure. Dans cet outil, il est possible d'exprimer la forme de la mémoire à l'aide de parcours pour explorer la mémoire ; les ensembles de tels parcours sont décrits par des langages réguliers. La méthode consiste alors à transformer en automate régulier toute condition exprimée par une formule de la logique utilisée ; pour vérifier une implication, ce procédé calcule alors des inclusions de langages d'automates réguliers.

Enfin, citons LRP ("Logic of Reachable Patterns"), introduit dans [YRS+05], formalisme dont l'expressivité est importante, notamment par la possibilité de traiter de récursion.

D'autre part, outre ces procédés par annotation de programmes, il existe des procédés automatiques, pour lesquels la réponse peut être négative pour un programme correct, ou ne pas terminer. Un raffinement des conditions de vérification peut alors être effectué par l'utilisateur, après observation du résultat proposé par le procédé. Selon les cas, ce raffinement peut être formel ou une simple augmentation de paramètres bornant l'exécution du procédé. Beaucoup de ces méthodes sont basées sur des automates, à l'exception notable de TVLA. Par exemple, les approches par automates d'arbres présentées dans [DEG06] et [HIV06] sont à classer ici.

Ainsi, un outil, basé sur la logique tri-valuée, est appelé TVLA ("Three-Valued Logic Analyser"), et a été introduit dans [LAMS04] et [SRW98]. La logique tri-valuée a trois valeurs de vérité : `vrai`, `faux`, et `indetermine` (indéterminé). C'est un procédé automatique, mais dont le résultat peut être `indetermine` ; c'est dans ce cas qu'il faut raffiner l'analyse, mais ce raffinement n'assure toujours pas l'obtention d'un résultat déterminé. En pratique, la logique tri-valuée fait une abstraction du modèle ; ainsi, plutôt que de connaître précisément la longueur d'une liste, en sachant alors précisément si la liste a un $n^{\text{ème}}$ élément, on dira que l'existence d'un $n^{\text{ème}}$ élément dans la liste est indéterminée. La mémoire que le programme vérifié prend pour entrée est ainsi abstraite, et ensuite chaque pas de calcul du programme modifie l'abstraction. Notons aussi un autre formalisme, l'"abstract regular model-checking", présenté dans [BHV04].

Un autre procédé, entièrement automatique mais non toujours applicable, permet d'exprimer des propriétés quantitatives fortes, par exemple sur la taille de la mémoire allouée. Il est présenté dans [BFN06] et [BFLS04]. Il fonctionne par systèmes à compteurs : la mémoire n'est retenue que par sa forme, et les informations sur la longueur des branches de l'arbre qu'est cette mémoire sont stockées dans des compteurs. L'automate à compteurs obtenu est alors soumis à l'outil FAST, qui utilise l'accélération plate pour le model-checking ([BFL04]).

Enfin, en marge de ces travaux, remarquons un travail utilisant la logique temporelle, ce qui, nous le verrons, nous intéresse. Dans [DKR04] est introduite une logique permettant de traiter de la forme de la mémoire, et assez précisément de l'évolution de la forme de la mémoire, c'est-à-dire de sa gestion dynamique. Lorsque le procédé ne valide pas un programme, il est possible de raffiner l'analyse, par une simple augmentation de paramètre. La méthode utilisée est une abstraction des structures de la mémoire dans des bornes données (les paramètres que l'on peut augmenter), combinée à des automates de Büchi. Par rapport au travail qui suit, ce procédé se distingue par de possibles erreurs (suivies d'un raffinement), et de l'absence de connecteurs spatiaux, dont nous allons maintenant parler.

## 1.3 Logique séparante

Voici donc quelque mots sur la logique séparante. Elle a été introduite par Reynolds dans [Rey02], et a mené à l'outil Smallfoot, présenté dans [BCO05]. Nous allons utiliser la logique séparante dans l'approche qui suit. Mais contrairement à ce que nous ferons, l'outil Smallfoot, et la logique séparante en général, fonctionnent par triplets de Hoare, c'est-à-dire par l'annotation systématique de programme par des formules de cette logique. Il s'agit plus de prouver un programme, par des règles de déduction, que de le vérifier. Le type de question posée est : étant donnés une précondition et une instruction de mon programme, peut-on assurer une postcondition, donnée elle aussi ?

Notons que la logique séparante ne permet pas de traiter naturellement les structures de données récursives. En effet, il faut introduire un nouveau prédicat dans la logique, par exemple pour pouvoir traiter le cas des listes.

L'ingrédient le plus marquant de cette logique est la conjonction séparante, qui permet de couper la mémoire. Pour l'introduire, prenons un exemple. Supposons qu'après la désallocation de `x`, on veuille exprimer cette désallocation.

Alors le triplet de Hoare suivant est faux :

$$\{\mathtt{x} \mapsto 3 \wedge \mathtt{y} \mapsto 3\}\mathtt{free(x)}\{\mathtt{y} \mapsto 3\}$$

En effet il est possible que $\mathtt{x} = \mathtt{y}$ : c'est le problème de l'aliasing. La logique séparante contient l'opérateur $*$, la conjonction séparante, pour lequel le triplet suivant est correct.

$$\{\mathtt{x} \mapsto 3 * \mathtt{y} \mapsto 3\}\mathtt{free(x)}\{\mathtt{y} \mapsto 3\}$$

La formule $\mathtt{x} \mapsto 3 * \mathtt{y} \mapsto 3$ signifie que dans une partie de la mémoire, $\mathtt{x} \mapsto 3$ est vraie, et dans une autre partie, séparée, $\mathtt{y} \mapsto 3$ est vraie.

Introduisons aussi deux règles de déduction des triplets de Hoare dans ce formalisme. Elles ne sont pas formulées parfaitement, mais donnent une idée de ce qu'est $*$.

$$\frac{\{C_1\}P\{C_2\}}{\{C_1 * C\}P\{C_2 * C\}} \quad \text{Frame rule}$$

La première, ci-dessus, stipule que lorsque un triplet de Hoare $\{C_1\}P\{C_2\}$ est correct, il est possible d'ajouter toute condition $C$ (dont les variables ne sont pas modifiées par $P$, ici est l'imprécision de ma formulation de cette règle) sur une partie séparée de la mémoire.

$$\frac{\{C_1\}P\{C_2\} \qquad \{C_1'\}P'\{C_2'\}}{\{C_1 * C_1'\}P \parallel P'\{C_2 * C_2'\}} \quad \text{Composition parallèle}$$

La seconde, ci-dessus, soumise à des conditions de non-interférence notamment entre $P$ et $P'$, précise que si deux programmes sont spécifiés, alors leur exécution dans deux parties disjointes de la mémoire suit la même spécification.

L'opérateur $*$ permet ainsi de séparer des zones de mémoire de taille quelconque. Mais la logique séparante permet aussi de spécifier des propriétés précises, par exemple que la mémoire est vide, ou ne contient qu'une cellule allouée. L'intérêt essentiel de cette logique est la possibilité d'un raisonnement local.

## 1.4 Objectif du stage

Nous allons utiliser la logique séparante dans un cadre différent de celui présenté ci-dessus. En effet, notre approche ne traite pas de triplets de Hoare. Nous ne prouverons pas un programme, mais nous exprimerons des propriétés sur son exécution, vérifiées par model-checking ; il n'y aura donc pas de règles de déduction comme ci-dessus. Par contre, nous allons utiliser la logique de séparation pour décrire des états mémoire, et à cela s'ajouteront des contraintes temporelles, de façon à pouvoir décrire des séquences d'états mémoire. Ainsi, nous obtiendrons une logique temporelle dont le but est de spécifier l'évolution de la mémoire pendant l'exécution d'un programme. La logique temporelle peut aussi être utilisée pour explorer la structure de la mémoire, possiblement récursive, ce qui fera probablement l'objet d'un futur travail.

Nous considèrerons la logique LTL, introduite dans [Pnu77], qui est un formalisme générique pour spécifier des dépendances temporelles. Afin de pouvoir comparer la mémoire à différents instants, notre logique prend en compte des formules $\mathtt{x} = \mathsf{X}\mathtt{y}$ permettant de comparer les valeurs des variables dans des états différents. Cette idée a été introduite dans [CC00] dans le cadre de LTL sur domaine concret.

Dans ce rapport se trouvent donc la description d'une logique, $\mathrm{LTL}^{\mathrm{mem}}$, dont nous décrivons et discutons la définition et l'expressivité dans ce qui suit. Notre approche se fait par model-checking, pour lequel nous tirons certaines informations d'un programme afin de le modéliser sous forme d'un automate dont les transitions sont étiquetées par des instructions élémentaires sur la mémoire. Ensuite, partant du constat que le model-checking se traite de façon analogue, nous prouverons que le problème de la satisfaisabilité pour la logique $\mathrm{LTL}^{\mathrm{mem}}$ est PSPACE-complet (théorème 1).

# 2 La logique

## 2.1 Notre modèle des états mémoire

Avant tout, nous allons ici définir le modèle de la mémoire que nous utiliserons dans le document. Il s'agit d'une abstraction de ce qui se passe physiquement.

### 2.1.1 Définition

Soient `Var` un ensemble infini dénombrable de variables, et `Val` un ensemble infini dénombrable de valeurs, habituellement $\mathbb{N}$. Soit `Loc` un sous-ensemble infini de `Val` : c'est l'ensemble des adresses mémoire.

Un état mémoire est un couple $(s, h)$ de $\mathcal{S} \times \mathcal{H}$

- $s$ est un store $\texttt{Var} \cup \{\texttt{null}\} \to \texttt{Val} \cup \{nil\}$, c'est-à-dire une fonction des variables dans les valeurs, sachant que $s(\texttt{null}) = nil$ ;
- $h$ est un tas $\texttt{Loc} \rightharpoonup_{fin} (\texttt{Val} \cup \{nil\})^+$, c'est-à-dire une fonction partielle d'un domaine fini, et inclus dans l'ensemble des adresses, dans les mots de valeurs.

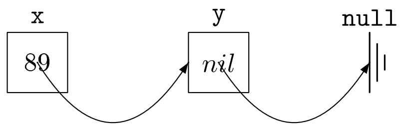
Le store est donc une valuation des variables.

Le domaine de $h$ est l'ensemble des adresses allouées. Une adresse allouée contient une cellule mémoire. L'ensemble de ces cellules est le tas $h$. On dit que $\texttt{x} \in \texttt{Var}$ pointe vers une cellule si $h(s(\texttt{x})$ est cette cellule.

### 2.1.2 Exemples

Prenons deux exemples d'états mémoire.

FIG. 1 – état mémoire

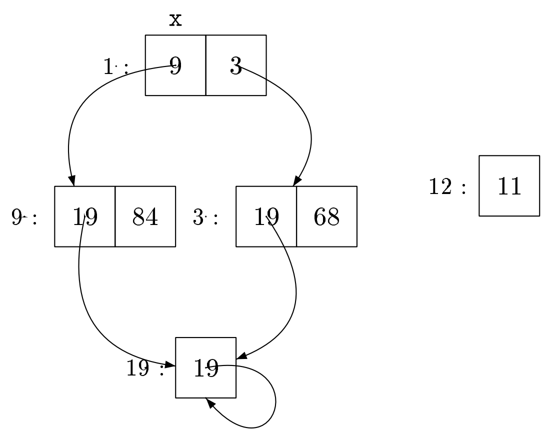



L'état mémoire décrit en figure 1, avec `x` pointant vers `y`, et `y` pointant vers `null`, est le suivant. Nous avons arbitrairement décidé que la valeur de `x` est 17, et celle de `y` est 89, c'est-à-dire que nous choisissons la valeur des variables dans la valuation des variables $s$, le store : $s(\texttt{x}) = 17$ et $s(\texttt{y}) = 89$ ; et par définition, $s(\texttt{null}) = nil$. Alors, $h(17) = 89$ et $h(89) = nil$. Prenons maintenant un autre exemple.

La figure 2 représente aussi un état mémoire. On y remarque qu'une cellule mémoire :

- ne pointe pas forcément vers une adresse caractérisée par une variable,
- peut comporter plusieurs champs,

FIG. 2 – état mémoire




– peut pointer vers la même cellule qu'une autre,
– ne correspond pas toujours à une variable,
– et même n'est pas nécessairement accessible.

Et donc, ici, $s(\mathtt{x}) = 1$ (et $s(\mathtt{null}) = nil$, non représenté) ; et $h(1) = (9, 3)$, $h(9) = (19, 84)$, $h(3) = (19, 68)$, $h(19) = (19)$ et finalement $h(12) = (11)$.

### 2.1.3 Évolution temporelle

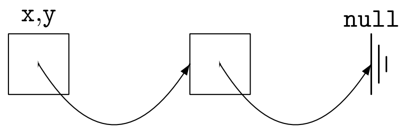
Notre logique traite de suites d'états mémoire. Ainsi, les modèles, dont l'ensemble est Mod, sont définis par :

**Définition 1** *L'ensemble des modèles est l'ensemble des suites d'états mémoire.*

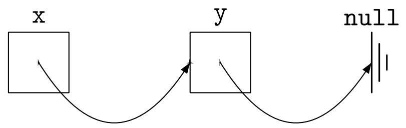
$$\mathtt{Mod} = \mathbb{N} \rightarrow \mathcal{S} \times \mathcal{H}$$

Dans la suite, on notera $\rho$ un tel modèle, et l'état $\rho(i)$ sera noté $(s_i, h_i)$.

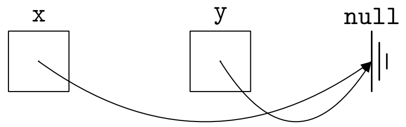
Par exemple, définissons un modèle $\rho_r$ décrivant l'évolution temporelle de la mémoire durant l'exécution d'un programme qui prend pour entrée la liste à deux éléments du premier dessin, et la retourne. Chaque dessin d'état mémoire est suivi par le commentaire le concernant.



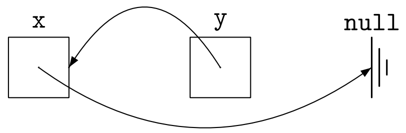

1) en entrée, la liste avec les variables qui ont pour valeur la tête.



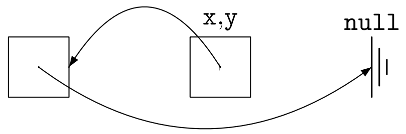

2) y prend la valeur du successeur de x, ce qu'on pourrait dire : y := x.1 (le premier champ de x).




3) alors, l'ancienne tête, la valeur de x, devient la queue, et pointe vers *nil*, ou : x.1 := *nil*.




4) l'ancien successeur (y) de la valeur de x doit maintenant avoir pour successeur x, ou y.1 := x.




5) la liste est retournée, toutes les variables prennent pour valeur sa nouvelle tête, ou x := y.

Précisément, le modèle $\rho_r$ pourrait être défini comme la fonction associant à 0, 1, 2, 3 et 4 les états mémoire ci-dessus dans l'ordre, puis rester constant.

## 2.2 Définition

Mon travail de stage a consisté en l'étude de la logique $\mathrm{LTL}^{\mathrm{mem}}$ ("LTL avec états mémoire"), dont la syntaxe et la sémantique m'ont été données par Stéphane Demri et Étienne Lozes, les encadrants de mon stage.

Il y a dans cette logique beaucoup de degrés de liberté ; on pourrait contraindre davantage au niveau du tas. Plus précisément, nous utilisons une sémantique sans grain : on ne définit pas ce qu'est le passage d'un état à un autre ; on a simplement une suite d'observations de la mémoire. Il est ainsi possible d'avoir pour modèle des suites d'états mémoire ne correspondant pas à l'idée qu'on se fait de l'exécution d'un programme. Par exemple, si $M_1$ est le premier état mémoire, et $M_2$ le second, un modèle peut être défini par $\rho : i \mapsto M_{i \bmod 2}$.

### 2.2.1 Syntaxe

C'est une logique à deux niveaux, l'un spatial et l'autre temporel. Il s'agit d'une extension de la logique LTL dont les formule atomiques sont les formules d'un fragment de la logique séparante, avec des comparaisons de variables à des instants différents. Soit $\mathtt{Var}$ un ensemble dénombrable de variables. On note $\mathrm{LTL}^{\mathrm{mem}}$ la logique définie par la grammaire des formules $\Phi$ :

– Expressions, pour $k$ un entier écrit en unaire, et $\mathtt{x} \in \mathtt{Var}$ :

$$\begin{array}{rll} e ::= & \mathsf{X}^k\mathtt{x} & \text{la valeur de } \mathtt{x}\ k \text{ instants plus tard} \\ \mid & \mathtt{null} & \text{la constante de valeur } \textit{nil} \end{array}$$

– Formules d'état, pour $n$ un entier, et $\mathtt{x}, \mathtt{x}_1, \ldots, \mathtt{x}_n \in \mathtt{Var}$ :

$$\begin{array}{rll} \mathcal{A} ::= & e = e & \text{les images des expressions par le store sont égales.} \\ \mid & \mathtt{x} \mapsto \mathtt{x}_1, \ldots, \mathtt{x}_n & \text{la cellule désignée par } \mathtt{x} \text{ dans le store contient le mot} \\ & & \text{de valeurs désigné par ce qui est à droite de la flèche.} \\ \mid & \mathtt{emp} & \text{la mémoire est vide, aucune cellule mémoire n'est allouée.} \\ \mid & \mathcal{A} * \mathcal{B} & \text{la “conjonction séparante” :} \\ & & \text{une partie de la mémoire vérifie} \mathcal{A}, \text{ et le reste } \mathcal{B}. \\ \mid & \mathcal{A} \mathrel{-\!\!*} \mathcal{B} & \text{la “baguette magique” : s'il existe une mémoire vérifiant } \mathcal{A}, \\ & & \text{alors elle permet de vérifier } \mathcal{B} \text{ dans le contexte actuel.} \\ \mid & \mathcal{A} \wedge \mathcal{B} & \text{la conjonction.} \\ \mid & \neg\mathcal{A} & \text{la négation.} \end{array}$$

– Formules temporelles :

$$\begin{array}{rll} \Phi ::= & \mathcal{A} & \text{l'état courant vérifie } \mathcal{A} \\ \mid & \mathsf{X}\Phi & \text{le “Next” : la formule } \Phi \text{ sera vraie à l'instant suivant.} \\ \mid & \Phi\mathsf{U}\Phi' & \text{le “Until” : la formule } \Phi \text{ sera vraie jusqu'à ce que } \Phi' \text{ le soit.} \\ \mid & \Phi \wedge \Phi' & \text{la conjonction.} \\ \mid & \neg\Phi & \text{la négation.} \end{array}$$

### 2.2.2 Sémantique

**Définition 2** *Soient $h_1$ et $h_2$ deux tas. Si $\mathtt{dom}(h_1) \cap \mathtt{dom}(h_2) = \emptyset$, on écrit $h_1 \perp h_2$, et l'opération $h_1 * h_2$ est définie : c'est l'*union disjointe *des deux fonctions.*

Soit $\rho$ un modèle et $h$ un tas. On écrit $\rho[i \leftsquigarrow h]$ le modèle $\rho$ dont le $i^{\text{ème}}$ tas $(h_i)$ est remplacé par $h$.

Définissons finalement la relation de satisfaction. Écrire $\rho, i \models \Phi$ signifie que le modèle $\rho \in \mathtt{Mod}$ satisfait la formule $\Phi$ à l'instant $i$. Alors :

- $\rho, i \models \mathsf{X}\Phi$ ssi $\rho, i+1 \models \Phi$.
- $\rho, i \models \Phi\mathsf{U}\Phi'$ ssi il existe $j \geq i$ tel que
  - $\rho, j \models \Phi'$ et
  - pour tout $i \leq k < j$, on a $\rho, k \models \Phi$.
- $\rho, i \models \mathsf{X}^k\mathtt{x} = \mathsf{X}^l\mathtt{y}$ ssi $s_{i+k}(\mathtt{x}) = s_{i+l}(\mathtt{y})$.

– $\rho, i \models \mathsf{X}^k\mathtt{x} = \mathtt{null}$ ssi $s_{i+k}(\mathtt{x}) = nil$.
– $\rho, i \models \mathtt{x} \mapsto \mathtt{x}_1, \ldots, \mathtt{x}_n$ ssi
  • $\mathtt{dom}(h_i) = \{s_i(\mathtt{x})\}$ et
  • $h_i(s_i(\mathtt{x})) = (s_i(\mathtt{x}_1), \ldots, s_i(\mathtt{x}_n))$.
– $\rho, i \models \mathtt{emp}$ ssi $\mathtt{dom}(h_i) = \emptyset$.
– $\rho, i \models \mathcal{A} * \mathcal{B}$ ssi il existe $h, h'$ tels que $h_i = h * h'$,
  • $\rho[i \leftarrow\!\!\rightsquigarrow h], i \models \mathcal{A}$ et
  • $\rho[i \leftarrow\!\!\rightsquigarrow h'], i \models \mathcal{B}$.
– $\rho, i \models \mathcal{A} \mathrel{-\!\!*} \mathcal{B}$ ssi pour tout $h$ tel que $h_i \perp h$,
  • $\rho[i \leftarrow\!\!\rightsquigarrow h], i \models \mathcal{A}$ implique $\rho[i \leftarrow\!\!\rightsquigarrow h_i * h], i \models \mathcal{B}$.

J'abrègerai parfois $\rho, 0 \models \Phi$ par $\rho \models \Phi$.

## 2.3 Problèmes étudiés

### 2.3.1 Model-checking et satisfaisabilité

Nous allons maintenant définir ces deux problèmes.

**Satisfaisabilité :** Voici le problème que nous nous posons ;
– entrée : une formule $\Phi$ de $\mathrm{LTL}^{\mathrm{mem}}$.
– sortie : "vrai" ssi il existe un modèle $\rho \in \mathtt{Mod}$ tel que $\rho \models \Phi$.

**Modèles** Le model-checking est l'étude de la satisfaction d'une formule par un modèle donnés, pour une relation de satisfaction.

Pour définir le model-checking dans notre cas, nous avons besoin de définir comment seront écrits nos modèles. En effet, l'algorithme ne peut prendre directement en entrée un modèle, qui est une suite infinie d'états.

Ainsi, un modèle pour le model-checking sera un automate de Büchi, dont les transitions sont étiquetées par des formules d'état, dont les expressions sont du type $\mathtt{x}$ ou $\mathsf{X}\mathtt{x}$ : pas plus de un "next", pas d'information sur les stores postérieurs au suivant. La définition d'un automate de Büchi se trouve en section 5.1

Un modèle pour le model-checking est un modèle d'une formule de $\mathrm{LTL}^{\mathrm{mem}}$ lorsqu'il existe un modèle concret, satisfaisant les conditions atomiques de chaque transition d'un calcul acceptant dans le modèle pour le model-checking, et étant un modèle de la formule.

**Model-checking :** Étant donnés un modèle pour le model-checking et une formule de $\mathrm{LTL}^{\mathrm{mem}}$, ce modèle est-il un modèle de la formule ?

Ce problème se réduit à la satisfaisabilité : il est possible de coder les contraintes, imposées par l'automate qui est le modèle pour le model-checking, dans une formule $\mathrm{LTL}^{\mathrm{mem}}$ par une méthode similaire à celle de [CS85] pour LTL seule.

### 2.3.2 Complexité

La seconde partie de ce mémoire est consacrée à la preuve en extension du théorème suivant, qui a représenté l'essentiel de mon travail de stage :

**Théorème 1** *Le problème de satisfaisabilité pour* $\mathrm{LTL}^{\mathrm{mem}}$ *est* PSPACE*-complet.*

Ou, autrement dit il existe un algorithme utilisant un espace polynomial qui, étant donnée une formule $\Phi$, détermine si il existe un modèle de la formule $\Phi$, un $\rho$ tel que $\rho \models \Phi$.

L'intérêt de ce résultat tient notamment au fait que LTL est déjà PSPACE-difficile ([CS85]), et la satisfaisabilité des formules d'état est aussi PSPACE-difficile ([CYO01]).

La réduction de la satisfaisabilité de LTL dans celle de $\mathrm{LTL}^{\mathrm{mem}}$ est prouvé plus loin dans le document, montrant la PSPACE-difficulté de la satisfaisabilité de $\mathrm{LTL}^{\mathrm{mem}}$ ; ce qui nous amènera ensuite à montrer, plus longuement, que le problème est inclus dans PSPACE.

# 3 Propriétés

## 3.1 Exemples

### 3.1.1 Propriétés locales

Venons en à l'usage des opérateurs de séparation. Ils traitent de l'union disjointe de deux tas : revenons d'abord sur cette opération $h_1 * h_2$ s'appliquant à des tas. La définition nous dit que les deux tas doivent avoir des domaines disjoints, autrement dit les cellules mémoire de l'un et l'autre ne se chevauchent pas. Ensuite, l'opération consiste à prendre l'union de ces deux ensembles de cellules. L'important, c'est que ces deux ensembles ($h_1$ et $h_2$) partitionnent le résultat de l'opération, $h = h_1 * h_2$, en deux régions indépendantes.

Cela nous permet de dire que la conjonction séparante raisonne spatialement : la formule $\mathcal{A} * \mathcal{B}$ est vraie quand $\mathcal{A}$ est vraie dans une partie de la mémoire, et $\mathcal{B}$ ailleurs. Ainsi, $(\mathtt{x} \mapsto \mathtt{x}') * (\mathtt{y} \mapsto \mathtt{y}')$ nous assure qu'il n'y a pas de problème d'aliasing entre $\mathtt{x}$ et $\mathtt{y}$, puisque $\mathtt{x} \mapsto \mathtt{x}'$ est vraie quelque part et $\mathtt{y} \mapsto \mathtt{y}'$ ailleurs. Prenons pour autre exemple la formule $\neg\mathtt{emp} * \neg\mathtt{emp}$ : elle stipule que les cellules mémoire peuvent être partitionnées en deux ensembles non vides, autrement dit qu'il y en a au moins deux ($|\mathtt{dom}(h)| \geq 2$). Plus tard cette formule sera notée $\mathtt{size}\ \geq\ 2$.

Enfin, la baguette magique quantifie universellement sur les tas. Par exemple, la formule $\mathtt{x} \mapsto nil \mathrel{-\!\!*} \perp$ est vraie pour $s_0, h_0$ quand tout tas dans lequel $\mathtt{x}$ est alloué à $nil$, et disjoint du tas courant $h_0$, satisfait $\perp$ avec $h_0$. Le sens est donc que s'il est possible d'allouer la cellule mémoire à l'adresse $s_0(\mathtt{x})$ à $nil$, alors on arrive à l'absurdité. Autrement dit, cette adresse est déjà allouée dans $h_0$, et un ensemble de cellules mémoire ne peut pas à la fois être disjoint de $h_0$ et avoir une cellule à cette adresse. Plus tard, cette formule sera appelée $\mathtt{alloc}\ \mathtt{x}$.

### 3.1.2 Dynamique des états mémoire

**Toujours, parfois :**
La formule $\top \mathsf{U} \Phi$ est vraie lorsque $\Phi$ sera vraie dans un instant futur. Nous dirons que parfois $\Phi$, et nous écrirons $\Diamond \Phi$ pour l'abréger.

La formule $\neg \Diamond \neg \Phi$ est vraie lorsque $\Phi$ est toujours vraie dans le futur. Nous dirons que toujours $\Phi$, et nous écrirons $\Box \Phi$ pour l'abréger.

Il est possible d'exprimer qu'une propriété $\Phi$ arrivera infiniment souvent, par $\Box \Diamond \Phi$ : pour tout instant futur, il existera un instant postérieur où $\Phi$ se réalisera. On peut ainsi dire qu'infiniment souvent la variable x est allouée : $\Box \Diamond (\mathtt{x} \mapsto nil \mathrel{-\!\!*} \bot)$.

**Parcours de liste.**
Définissons la formule :

$$\Phi_1 = (\mathtt{x} \mapsto \mathtt{y} \wedge \mathtt{y} = \mathsf{X}\mathtt{x}) \mathsf{U} (\mathtt{x} = \mathtt{null})$$

Elle peut être satisfaite par un modèle dont le tas est constant, et contient une liste de longueur arbitraire, et le store évolue d'un instant à l'autre avec x prenant la valeur de son successeur à l'instant précédent, et y le successeur x, et ce jusqu'au bout de la liste, quand x vaut *nil*.

**Parcours de liste ?**
Cependant, il est aussi possible de définir la formule suivante :

$$\Phi_2 = \Phi_1 \wedge \Box \neg (\neg \mathtt{emp} * \neg \mathtt{emp})$$

Qui signifie que la formule $\Phi_1$ est vraie alors que le tas n'a jamais un domaine de taille supérieure à 1, autrment dit qu'il n'existe jamais plus d'une cellule mémoire. Cette formule est satisfaisable, avec un parcours arbitrairement long pour x. Soit le modèle :

- $s_0(\mathtt{x}) = 0$ et $s_0(\mathtt{y}) = 1$, avec $h_1(0) = 1$ (et $\mathtt{dom}(h_1) = \{0\}$) ;
- $s_1(\mathtt{x}) = 1$ et $s_1(\mathtt{y}) = 0$, avec $h_1(1) = 0$ (et $\mathtt{dom}(h_1) = \{1\}$) ;
- $s_{2i}(\mathtt{x}) = 0$ et $s_{2i}(\mathtt{y}) = 1$, avec $h_{2i}(0) = 1$ ;
- $s_{2i+1}(\mathtt{x}) = 1$ et $s_{2i+1}(\mathtt{y}) = 0$, avec $h_{2i+1}(1) = 0$ ;
- $s_{2i_0}(\mathtt{x}) = 0$ et $s_{2i_0}(\mathtt{y}) = nil$, avec $h_{2i_0}(0) = nil$ ;
- $s_{2i_0+1}(\mathtt{x}) = nil$ et $\mathtt{dom}(h_{2i_0+1}) = \emptyset$.

Ce modèle, pour peu que l'on choisisse d'avoir des tas vides aux instants postérieurs à $2i_0+1$, satisfait la formule $\Phi_2$. Cet exemple illustre que l'évolution des états mémoire peut être surprenante.

## 3.2 Expressivité

### 3.2.1 Encoder LTL

Par LTL, nous désignons la logique dont les modèles sont des suites infinies d'ensembles de variables propositionelles de PROP, dont la syntaxe est, pour

$p \in$ PROP (variable propositionelle) :

$$\Phi ::= p \mid \mathsf{X}\Phi \mid \Phi \mathsf{U} \Phi' \mid \Phi \wedge \Phi \mid \neg \Phi$$

Et dont la sémantique est, si $M = (X_i)_{i \in \mathbb{N}} \in (\text{PROP})^{\mathbb{N}}$ est un modèle :
- $M, i \models p$ ssi $p \in X_i$,
- $M, i \models \mathsf{X}\Phi$ ssi $M, i+1 \models \Phi$,
- $M, i \models \Phi \mathsf{U} \Phi'$ ssi il existe $j_0 \geq i$ tel que $M, j_0 \models \Phi'$ et, pour tout $j \in [\![i, i+j_0]\!]$, on a $M, j \models \Phi$
- $M, i \models \Phi \wedge \Phi'$ ssi $M, i \models \Phi$ et $M, i \models \Phi'$,
- $M, i \models \neg\Phi$ ssi non $M, i \models \Phi$.

Le niveau temporel de notre logique a donc une syntaxe et une sémantique similaire.

**Lemme** *Pour toute formule de $\phi$ LTL, il existe une formule $\phi_{\text{LTL}^{\text{mem}}}$ de* LTL$^{\text{mem}}$ *satisfaisable ssi $\phi$ l'est ; de plus $\phi_{\text{LTL}^{\text{mem}}}$ peut être obtenue en espace logarithmique non-déterministe en la taille de $\phi$.*

Montrons le. Pour toute variable propositionelle $p \in$ PROP, on choisit une variable $\mathtt{x}_p \in \mathtt{Var}$. Ensuite, étant donnée une formule $\phi$ de LTL, il suffit d'introduire une nouvelle variable, mettons $\mathtt{x}_\top$, et la formule $\phi_{\text{LTL}^{\text{mem}}} = \phi[p \leftarrow \mathtt{x}_p = \mathtt{x}_\top] \wedge \Box(\mathtt{x}_\top = \mathsf{X}\mathtt{x}_\top)$ est satisfaisable dans LTL$^{\text{mem}}$ ssi $\phi$ l'est dans LTL : une variable propositionelle est considérée comme appartenant à l'ensemble $X_i$ d'un modèle de LTL ssi la variable associée $\mathtt{x}_p$ est égale à $\mathtt{x}_\top$ à l'instant $i$ dans un modèle de LTL$^{\text{mem}}$.

Cela se fait en espace logarithmique donc en espace logarithmique non déterministe.

**Proposition** *La satisfaisabilité pour* LTL$^{\text{mem}}$ *est* PSPACE*-difficile.*

Du lemme précédent, on peut conclure la proposition ci-dessus.

### 3.2.2 Variations possibles

Pour souligner les contours de l'expressivité de LTL$^{\text{mem}}$, nous allons présenter différentes extensions ou modifications envisageables. Certaines feront probablement à l'avenir l'objet d'une étude plus poussée.

**Première extension :**
Le langage de LTL$^{\text{mem}}$ ne permet pas de contraindre fortement deux tas successifs d'un modèle. Mais, sans alourdir la logique, il est possible d'introduire des formules du type : $\mathsf{X}\mathtt{x} \mapsto \mathtt{y}$, dont la sémantique serait : dans le tas courant, la valeur au prochain état de $\mathtt{x}$ pointe vers celle de $\mathtt{y}$. Cela peut aussi s'écrire $\mathsf{X}\mathtt{x} = \mathtt{z} \wedge \mathtt{z} \mapsto \mathtt{y}$ dans la syntaxe de LTL$^{\text{mem}}$, en ajoutant une variable.

**Fixer temporairement la mémoire :**

C'est l'extension qui me semble la plus intéressante, tout en restant dans l'esprit de $\mathrm{LTL}^{\mathrm{mem}}$. Il s'agirait d'introduire une formule, mettons †, pouvant être utilisé alternativement à une formule d'état dans une formule : la syntaxe serait

$$\Phi ::= \mathcal{A} \mid \dagger \mid \mathsf{X}\Phi \mid \Phi\mathsf{U}\Phi \mid \Phi \wedge \Phi \mid \neg\Phi$$

Et la sémantique de † serait $\rho, i \models \dagger$ ssi $h_i = h_{i+1}$.

Cela permettrait d'explorer la mémoire de façon plus complète entre chaque observation de la mémoire qui nous est livrée par le modèle, en fixant l'état de la mémoire jusqu'à ce qu'une condition de contrôle $\Phi$ soit vérifiée : $\dagger\mathsf{U}\Phi$. La temporalité de la logique serait utilisée à la fois pour que l'état mémoire évolue, et pour que chaque état mémoire puisse être exploré, ce qui serait utile au vu de notre sémantique sans grain. Par contre, l'exploration de ces états mémoire se ferait sur une partie de la mémoire de taille non bornée, ce qui rend l'approche choisie dans la preuve de complexité pour $\mathrm{LTL}^{\mathrm{mem}}$ qui suit inopérante. Et probablement augmente la complexité de la satisfaisabilité.

**Allocation dynamique de cellules mémoire :**

Une formule encore plus précise que † pourrait permettre de spécifier qu'entre le tas courant et le tas suivant, une seule chose a changé : une case mémoire a été créée, et la variable `x` a pour valeur cette case. Cette formule pourrait être nommée `new x`, et ajoutée dans la syntaxe de la même façon que †. Les problèmes de complexité sont cette fois plus importants : la satisfaisabilité de la logique devient indécidable, car il est possible d'y coder l'arrêt d'une machine de Minsky[1], par une technique tirée de [BFN06].

Pour ce codage, chaque compteur est représenté par une variable : $\mathtt{x}_1$ et $\mathtt{x}_2$. Un compteur vaut zéro lorsque la variable correspondante vaut *nil* : c'est-à-dire $\mathtt{x}_i = \mathtt{null}$. La valeur d'un compteur est la longueur du chemin menant de la case mémoire désignée par la variable à *nil*. Pour l'incrémenter, on écrit, pour une variable $\mathtt{y}$, $\mathtt{y} \mapsto \mathtt{x}_i \wedge \mathtt{y} = \mathsf{X}\mathtt{x}_i$. Ce que permet de faire le `new` , c'est de fixer les variables $\mathtt{x}_1$ et $\mathtt{x}_2$ sur des listes, pour que la mémoire garde la forme de deux compteurs. Pour cela, la mémoire est choisie comme vide à l'instant initial, et deux variables $\mathtt{z}_1$ et $\mathtt{z}_2$ sont ajoutées ; et à chaque instant, l'une et l'autre servent alternativement à construire deux listes :

$$\begin{array}{lll}
\mathtt{emp} \wedge \mathtt{y}' = nil \wedge & & \text{le tas est vide et ensuite} \\
\square([\mathtt{y}' = nil \Rightarrow & & \text{si } \mathtt{y}' = nil \text{ alors} \\
 & (\mathtt{z}_1 = \mathsf{X}\mathtt{y}'' \wedge \mathsf{X}(\mathtt{new}\ \mathtt{z}_1 \wedge \mathtt{z}_1 \mapsto \mathtt{y}'') & \quad \text{au prochain instant } \mathtt{z}_1 \text{ sera} \\
 & & \quad\ \text{une nouvelle case mémoire} \\
 & & \quad\ \text{pointant vers sa valeur actuelle} \\
 & \wedge\mathsf{X}(\mathtt{z}_1 = \mathsf{X}\mathtt{z}_1 \wedge \mathtt{y}' \neq nil))] & \quad \text{et la variable } \mathtt{z}_1 \text{ sera stable,} \\
 & & \quad\ \text{pendant que } \mathtt{z}_2 \text{ pourra changer ;} \\
\wedge[\mathtt{y}' \neq nil \Rightarrow & & \text{sinon,} \\
 & (\mathtt{z}_2 = \mathsf{X}\mathtt{y}'' \wedge \mathsf{X}(\mathtt{new}\ \mathtt{z}_2 \wedge \mathtt{z}_2 \mapsto \mathtt{y}'') & \quad \text{symétriquement pour } \mathtt{z}_2. \\
 & \wedge\mathsf{X}(\mathtt{z}_1 = \mathsf{X}\mathtt{z}_2 \wedge \mathtt{y}' = nil))])
\end{array}$$

[1] automate à deux compteurs avec incrémentation et test à zéro

Il reste alors à empêcher les compteurs de croître plus vite que la construction des listes, en ne faisant qu'une transition de l'automate tous les deux instants, automate codé par une formule de la logique par les méthodes classiques pour LTL.

**Logique d'exploration :**
Une autre logique, dont la syntaxe et la sémantique m'ont été proposées dès le début du stage par mes encadrants, en même temps que celle de $\mathrm{LTL}^{\mathrm{mem}}$, et dont je ne ferai que décrire les grandes lignes, est une piste pour un travail futur. Plutôt que de pouvoir fixer le tas à certains instants comme proposé deux paragraphes auparavant, il s'agirait de travailler sur un tas constant. Les modèles seraient donc des suites de stores associées à un unique tas. La temporalité permettrait d'explorer ce tas, avec des opérateurs temporels ("until") dans la portée d'opérateurs spatiaux (conjonction séparante).

## Deuxième partie
# Model-checking par automates

Nous allons maintenant nous attacher à prouver le théorème 1. Nous avons déjà montré que la logique LTL peut être codée dans $\mathrm{LTL}^{\mathrm{mem}}$. Cela signifie que la satisfaisabilité dans $\mathrm{LTL}^{\mathrm{mem}}$ est de complexité supérieure à celle de LTL, et donc au moins PSPACE-difficile, puisque la satisfaisabilité de LTL est PSPACE-complète.

Notons pour commencer que la méthode traditionnelle par automates de Büchi ne peut s'appliquer, puisque l'ensemble d'états mémoire est infini. Utilisant une méthode inspirée de [DD02], nous allons définir une abstraction des états mémoire dans la prochaine section. La suivante montrera que cette abstraction est bien choisie, et enfin nous devrons nous intéresser à des arguments de complexité.

## 4 Abstraction

Nous allons introduire un fragment classique de la logique séparante, emprunté à [Loz04], qui servira de base à la définition des modèles symboliques. Pour cela, nous allons utiliser les formules suivantes :

- $\mathtt{alloc}\ \mathtt{x} \stackrel{\text{def}}{\equiv} \mathtt{x} \mapsto \mathit{nil} \mathrel{-\!\!*} \bot$, qui signifie que la variable $\mathtt{x}$ est allouée.
- $\mathtt{size}\ \geq\ k \stackrel{\text{def}}{\equiv} \neg\mathtt{emp} * \cdots * \neg\mathtt{emp}$ $\quad$ ($k$ fois), qui signifie que la mémoire contient au moins $k$ cellules, puisque elle peut être découpée en $k$ parties en contenant au moins une.
- $\mathtt{x} \hookrightarrow \mathtt{x}_1, \ldots, \mathtt{x}_n \stackrel{\text{def}}{\equiv} \mathtt{x} \mapsto \mathtt{x}_1, \ldots, \mathtt{x}_n * \top$, qui désigne un pointeur pouvant être entouré d'autres cellules.

### 4.1 Modèles symboliques

#### 4.1.1 Un fragment classique

**Définition 3** *Une mesure* $\mu = (m, \eta, w, V)$ *est un élément de* $\mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathcal{P}_{fin}(\mathtt{Var})$*, où* $\mathcal{P}_{fin}(\mathtt{Var})$ *est l'ensemble des parties finies de* $\mathtt{Var}$*.*

**Définition 4** *On note* $\mathcal{L}_\mu$ *l'ensemble des formules* $\phi$ *engendré par la grammaire :*

$$e ::= \mathsf{X}^u\mathtt{x} \ \mid\ \mathtt{null}$$

$$\phi ::= \mathtt{x} \hookrightarrow \mathtt{x}_1, \ldots, \mathtt{x}_n \mid e = e' \mid \mathtt{alloc}\ \mathtt{x} \mid \mathtt{size}\ \geq\ k$$

*Avec* $u \leq m$*,* $n \leq \eta$*,* $k \leq w$ *et* $x \in V$ *si* $\mu = (m, \eta, w, V)$*.*

C'est un fragment de cardinal finie de la logique séparante.

### 4.1.2 La mesure relative à $\Phi$

Nous allons définir une mesure bornant le pouvoir discriminant d'une formule $\Phi$.

Ainsi, étant donnée une formule $\Phi$, soit $m_\Phi = max\{i,$ il existe $\mathtt{x}$ tel que $\mathsf{X}^i\mathtt{x}$ apparaît dans $\Phi\}$, caractérisant la taille de la fenêtre temporelle que la formule en question observe à un instant donné.

Ensuite, nous allons borner la taille de la mémoire des modèles ; pour cela, définissons d'abord la hauteur d'étoile :

$$
\begin{aligned}
spl(\mathcal{A}) = & 1 \text{ si } \mathcal{A} = \mathtt{x} \mapsto \mathtt{x}_1, \ldots, \mathtt{x}_a, \\
& 0 \text{ si } \mathcal{A} = \mathtt{emp}, \\
& max\{i,j\} + 1 \text{ si } \mathcal{A} = \mathcal{A}^1 * \mathcal{A}^2 \text{ et } spl(\mathcal{A}^1) = i, spl(\mathcal{A}^2) = j, \\
& max\{i,j\} \text{ si } \mathcal{A} = \mathcal{A}^1 \mathrel{-\!\!*} \mathcal{A}^2 \text{ et } spl(\mathcal{A}^1) = i, spl(\mathcal{A}^2) = j, \\
& max\{i,j\} \text{ si } \mathcal{A} = \mathcal{A}^1 \Rightarrow \mathcal{A}^2 \text{ et } spl(\mathcal{A}^1) = i, spl(\mathcal{A}^2) = j, \\
& -\infty \text{ sinon.}
\end{aligned}
$$

soit maintenant $w_\Phi = 2^{max\{spl(\mathcal{A}), \mathcal{A} \in \Phi\}}$, qui limite la taille de la mémoire explorable.

Enfin, définissons une limite $\eta_\Phi = max\{i, \mathtt{x} \mapsto \mathtt{x}_1 \ldots \mathtt{x}_i \in \Phi\}$, à la longueur des expressions vers lesquelles une variable peut pointer.

Cela permet d'obtenir, avec $V_\Phi$ l'ensemble des variables occurant dans $\Phi$, $\mu_\Phi = (m_\Phi, w_\Phi, \eta_\Phi, V_\Phi)$ la mesure relative à la formule $\Phi$. $\Phi$ peut être une formule d'état, que l'on préfèrera noter $\mathcal{A}$ ; la mesure relative s'écrira alors $\mu_\mathcal{A}$.

Ordonnons terme à terme les mesures.

**Définition 5** *On note* $(m, \eta, w, V \leqslant (m', \eta', w', V')$ *lorsque* $m \leq m'$, $\eta \leq \eta'$, $w \leq w'$ *et* $V \subseteq V'$.

### 4.1.3 Définition des modèles symboliques

Commençons par définir un modèle partiel, pour une mesure $\mu$. C'est un tas $h$, et $m+1$ stores $s_0, \ldots s_m$. Soit alors une formule d'état $\mathcal{A}$, de mesure $\mu_\mathcal{A} \leqslant \mu$.

**Définition 6** *Nous dirons que le modèle partiel* $s_0, \ldots, s_m, h$ *est un modèle de* $\mathcal{A}$ *et nous noterons* $s_0, \ldots, s_m, h \models_{\text{part}} \mathcal{A}$ *s'il existe des tas et stores remplaçant les "_ ",*

$$[(s_0, h), (s_1, \_), \ldots, (s_m, \_), (\_, \_), \ldots], 0 \models \mathcal{A}$$

Notons que cette définition pourrait de façon équivalente être formulée : "[...] si, *pour tous tas et stores* remplaçant les "_", [...]", puisque ces derniers ne seront pas consultés pour déterminer la satisfaction de $\mathcal{A}$ ; en effet, $m_\mathcal{A} \leq m$.

Maintenant, la consistance maximale est une propriété s'appliquant aux ensembles de formules de $\mathcal{L}_\mu$. [2]

**Définition 7** *Un ensemble $X \subseteq \mathcal{L}_\mu$ est maximalement consistant si il existe un modèle partiel, constitué de $m+1$ stores $s_0 \dots s_m$ et un tas $h$, tel que, pour tout $\mathcal{A} \in \mathcal{L}_\mu$ :*

$$s_0, \dots, s_m, h \models_{\text{part}} \mathcal{A} \text{ ssi } \mathcal{A} \in X$$

Un ensemble maximalement consistant sera aussi appelé une *lettre*.

Nous noterons $\Sigma_\mu$ l'ensemble des parties maximalement consistantes de $\mathcal{L}_\mu$.

**Définition 8** *Finalement, un modèle symbolique est une suite d'éléments de $\Sigma_\mu$, un élément de*

$$M_s \stackrel{def}{\equiv} \mathbb{N} \to \Sigma_\mu.$$

**Lien avec les modèles concrets :**

Nous nous ramenons des modèles concrets, suites de tas et de stores, à des modèles symboliques, suites d'ensembles de formules, par la fonction suivante :

**Définition 9** *Soit $\mu$ une mesure. La fonction d'abstraction $Abs_\mu$ est définie par :*

$$\begin{array}{rrcl} Abs_\mu : & \mathtt{Mod} & \to & M_s \\ & \rho & \mapsto & (Abs_\mu(\rho) : i \mapsto \{\mathcal{A} \in \mathcal{L}_\mu : \rho, i \models \mathcal{A}\}) \end{array}$$

C'est-à-dire que le modèle symbolique associé à un modèle concret est la suite des formules de $\mathcal{L}_\mu$ satisfaites à chaque instant par ce modèle concret, les autres étant considérées comme fausses.

Nous devons montrer qu'un tel ensemble $Abs_\mu(\rho)(i)$ est maximalement consistant, c'est-à-dire est bien une lettre de $\Sigma_\mu$. Mais pour cela, il suffit de remarquer que, pour le cas de la $i^{\text{ème}}$ lettre, les $m$ stores $s_i \dots s_{i+m}$ et le tas $h_i$ permettent de vérifier la définition. Une pénible énumération des cas permettrait d'être rigoureux.

## 4.2 Correction de l'abstraction

### 4.2.1 Satisfaction symbolique

Nous allons utiliser un modèle symbolique, qui es donc une suite d'éléments de $\Sigma_\mu$, pour définir une distribution de vérité sur les formules de mesure inférieure à celle de la formule $\Phi$.

[2]Remarquons que la terminologie "maximalement consistant" n'est pas parfaitement appropriée : l'ensemble est bien consistant, mais il reste possible de lui ajouter une formule de pointeur, il n'est donc pas vraiment maximal. Par contre, toute formule qui n'est pas dans l'ensemble est considérée comme fausse : un ensemble maximalement consistant $X$ induit une distribution de vérité sur les formules de $\mathcal{L}_\mu$ à chaque instant, avec $\mathcal{A}$ est vraie ssi $\mathcal{A} \in X$ ; ce qui justifie partiellement le mot "maximal".

**Définition 10** *Soit une mesure $\mu$.*

*Tout d'abord, nous disons qu'un ensemble maximalement consistant $X \in \Sigma_\mu$ satisfait symboliquement une formule d'état $\mathcal{A}$ de mesure inférieure à $\mu$, ce que nous écrirons $X \models_{\mathrm{s}} \mathcal{A}$, lorsque tout modèle partiel $(s_0, \ldots, s_m, h)$ de $X$ satisfait $\mathcal{A}$ (c'est-à-dire $(s_0, \ldots, s_m, h) \models_{\mathrm{part}} \mathcal{A}$).*

*Nous pouvons alors définir inductivement la* satisfaction symbolique $\models_\mu$ *d'une formule quelconque $\phi$ par un modèle symbolique $\sigma$.*

- *Pour le cas des formules d'état, $\sigma, i \models_\mu \mathcal{A}$ ssi $\sigma(i) \models_{\mathrm{s}} \mathcal{A}$.*
- *Ensuite, pour les autres formules $\wedge$, $\neg$, $\mathsf{U}$ et $\mathsf{X}$, la satisfaction symbolique est définie inductivement exactement comme la satisfaction (non symbolique). Ainsi :*
  - *Si $\phi = \phi_1 \wedge \phi_2$, alors $\sigma, i \models_\mu \phi_1 \wedge \phi_2$ ssi $\sigma, i \models_\mu \phi_1$ et $\sigma, i \models_\mu \phi_2$.*
  - *Si $\phi = \mathsf{X}\phi'$, alors $\sigma, i \models_\mu \mathsf{X}\phi'$ ssi $\sigma, i+1 \models_\mu \phi'$.*
  - *Et les cas de $\neg$ et $\mathsf{U}$ sont similaires.*

Ce n'est que maintenant que nous pouvons énoncer ce lemme :

**Lemme 1** *Soit $\mu = \mu_\Phi$. Alors,*
*$\Phi$ est satisfaisable ssi il existe un modèle symbolique $\sigma$ tel que*

- *$\sigma$ est un modèle symbolique de $\Phi$ ($\sigma \models_\mu \Phi$),*
- *$\sigma$ a un modèle concret ($\exists \rho, Abs_\mu(\rho) = \sigma$).*

Il dépend assez immédiatement du lemme suivant.

**Lemme 1.1** *Si $\mu_\phi \leqslant \mu$, alors $\rho, i \models \phi$ ssi $Abs_\mu(\rho), i \models_\mu \phi$.*

Que l'on démontre par induction sur $\phi$ :
**Démonstration:**
Pour l'étape inductive de l'induction, les satisfactions symbolique et non symbolique étant définies de la même façon, elles se comportent identiquement. Par exemple, dans les deux cas, symbolique et non symbolique, un modèle satisfait $\neg\phi$ à l'instant $i$ ssi ce modèle ne satisfait pas $\phi$ à l'instant $i$. Reste le cas d'initialisation de l'induction.

Si $\phi$ est une formule d'état, notons la $\mathcal{A}$. Alors $Abs_\mu(\rho), i \models_\mu \mathcal{A}$ ssi $Abs_\mu(\rho)(i) \models_{\mathrm{s}} \mathcal{A}$, ssi tout modèle partiel de $Abs_\mu(\rho)$ en $i$ satisfait cette formule. Or $s_i, \ldots, s_{i+m}$ et $h_i$ extraits de $\rho$ sont (clairement) un modèle partiel de $Abs_\mu(\rho)$. Donc, comme $s_i, \ldots, s_{i+m}, h_i \models_{\mathrm{s}} \mathcal{A}$ ssi $\rho \models \phi$, nous avons l'équivalence $Abs_\mu(\rho), i \models_\mu \mathcal{A}$ ssi $\rho \models \phi$ ; à condition de montrer que :tout modèle partiel de $Abs_\mu(\rho)$ en $i$ satisfait $\mathcal{A}$ ssi $s_i, \ldots, s_{i+m}, h_i \models_{\mathrm{s}} \mathcal{A}$. Si l'on considère que tout modèle partiel de $Abs_\mu(\rho)$ satisfait exactement les formules de $Abs_\mu(\rho)$ parmi celles de $\mathcal{L}_\mu$, ou plus précisément que tout modèle complétant un modèle partiel de $Abs_\mu(\rho)$, en mettant des tas et des stores à la place des "_" de la définition, aura pour abstraction $Abs_\mu(\rho)$, c'est ce que nous dit le lemme 1.2 (*si deux modèles ont la même abstraction relativement à la mesure $\mu$, alors ils satisfont exactement les mêmes formules d'état de mesure inférieure à $\mu$*). $\square$

Venons-en à la preuve du lemme 1.

**Démonstration:**
S'il existe un tel modèle symbolique, alors son modèle concret satisfait exactement les formules de mesure plus petite que $\mu_\Phi$ qu'il satisfait symboliquement, par le lemme 1.1. Ce modèle concret satisfait donc $\Phi$, dès lors satisfaisable.

Si $\Phi$ est satisfaisable, soit $\rho$ l'un de ses modèles concrets. Le modèle symbolique $Abs_\mu(\rho)$ satisfait alors symboliquement $\Phi$ par le lemme 1.1 et le fait que $\mu_\Phi \leqslant \mu_\Phi$. $\square$

Nous allons maintenant nous intéresser au lemme 1.2.

### 4.2.2 Portée de la mesure

Ici se trouve une adaptation somme toute peu intéressante d'un résutat très proche de [Loz04].

Nous allons définir une équivalence modulo une certaine mesure $\mu$ de deux modèles $\rho$ et $\rho'$ pour $\mathrm{LTL}^{\mathrm{mem}}$.

**Définition 11** *$\rho \simeq_{\mu,i} \rho'$ ssi $Abs_\mu(\rho)(i) = Abs_\mu(\rho')(i)$.*

$\simeq_{\mu,i}$ est une relation d'équivalence sur $(\mathcal{S} \times \mathcal{H})$.

Nous voudrions montrer :

**Lemme 1.2** *Si $\rho \simeq_{\mu,i} \rho'$ et $\mu_\mathcal{A} \leqslant \mu$ alors $\rho, i \models \mathcal{A}$ ssi $\rho', i \models \mathcal{A}$.*

Pour cela, nous aurons besoin de trois lemmes sur la relation d'équivalence. Avant tout, définissons une notation.

**Définition 12** *Nous noterons $\mu/2$ la mesure $(m, \eta, w/2, V)$ si $\mu = (m, \eta, w, V)$, c'est-à-dire la mesure dont la limite à la taille de la mémoire explorable est divisée par deux.*

Voici les trois lemmes, prouvés en annexe 8 :

**Lemme 1.2.1 (Distributivité)** *Si $\rho \simeq_{\mu,i} \rho'$ et s'il existe $h^1$ et $h^2$ tels que $h^1 * h^2 = h_i$ (le $i^{\grave{e}me}$ tas du modèle $\sigma$), alors : il existe $h'^1$ et $h'^2$ tels que $h'^1 * h'^2 = h'_i$ (le $i^{\grave{e}me}$ tas du modèle $\sigma'$) et pour $u \in \{1,2\}$ on a $\rho[i \leftsquigarrow h^u] \simeq_{\mu/2,i} \rho'[i \leftsquigarrow h'^u]$.*

La démonstration se fait en construisant $h'^1$ et $h'^2$, à partir de tous les autre éléments donnés, en séparant la construction des tas entre la partie accessible par des variables, et la partie non accessible.

**Lemme 1.2.2** *Si $\rho \simeq_{\mu,i} \rho'$ et $h$ est un tas quelconque, alors il existe $h'$ tel que $\rho[i \leftsquigarrow h] \simeq_{\mu,i} \rho'[i \leftsquigarrow h']$.*

La démonstration est du même type que celle du lemme de distributivité. Voici un dernier lemme décrivant le bon comportement de $\simeq$,

**Lemme 1.2.3 (Congruence)** *Si $\rho \simeq_{\mu,i} \rho'$, $h^1 \perp h_i$, $h^2 \perp h'_i$, et $\rho[i \leftsquigarrow h^1] \simeq_{\mu,i} \rho'[i \leftsquigarrow h^2]$ alors $\rho[i \leftsquigarrow h_i * h^1] \simeq_{\mu,i} \rho'[i \leftsquigarrow h'_i * h^2]$.*

À l'aide de ces lemmes, il est possible de prouver la propriété qui nous importe, pour une formule d'état $\mathcal{A}$, le lemme 1.2 : deux modèles équivalents à l'instant $i$ satisfont les mêmes formules d'état à cet instant, ce qui se prouve par induction sur les formules satisfaites.

**Pourquoi le fragment est-il classique ?**
Voici le contexte dans lequel Étienne Lozes a introduit ce fragment de la logique séparante.

Soit $X$ un sous-ensemble de $\mathcal{L}_\mu$, alors

$$\mathcal{A}_X = \bigwedge_{\psi \in X} \psi \wedge \bigwedge_{\psi \in \mathcal{L}_\mu \setminus X} \neg\psi$$

est une formule d'état. Nous pouvons l'appeler formule classique associée à $X$.

Il y a un nombre fini de classes d'équivalences pour $\simeq_{\mu,i}$ étant donnés $\mu$ et $i$. Dès lors, une formule $\mathcal{A}$ de mesure inférieure à $\mu$ peut être écrite comme la disjonction des formules classiques associées aux abstractions des modèles la satisfaisant. C'est-à-dire, soit $C$ un ensemble de modèles contenant exactement un modèle de chaque classe d'équivalence de $\simeq_{\mu,i}$, alors on définit :

$$class_{\mathcal{A}} = \bigvee_{\rho \in C : \rho, i \models \mathcal{A}} \mathcal{A}_{Abs_\mu(\rho)}$$

La formule obtenue est équivalente à $\mathcal{A}$ : aucun modèle ne peut l'en différencier.

# 5 Automates de Büchi

Les automates de Büchi permettent de reconnaître des langages de mots infinis. Leur expressivité est équivalente à celle de MSO, c'est-à-dire supérieure à celle de LTL.

On s'en servira pour vérifier des propriétés sur des suites infinies de lettres. Plus précisément, nous allons les utiliser pour vérifier des propriétés sur les modèles symboliques : les séquences vérifiant les propriétés du lemme 1 forment un langage reconnaissable par un automate de Büchi.

## 5.1 Définition

Un automate de Büchi étendu est un quintuplet $\mathbb{A} = (\Sigma, Q, i, \mathcal{F}, \delta)$ où :
- $\Sigma$ est un alphabet fini,
- $Q$ est un ensemble fini d'états,
- $I$ est un sous-ensemble de $Q$, les états initiaux,
- $\mathcal{F}$, la condition d'acceptation, est un ensemble $(F_1, \dots F_n)$ d'ensembles d'états : les $F_i$ sont les ensembles finaux.
- $\delta$ est la table de transition, un sous-ensemble de $Q \times \Sigma \times Q$.

**Définition 13** *Soit un automate de Büchi* $\mathbb{A} = (\Sigma, Q, i, (F_1, \dots F_n), \delta)$.

*Un calcul acceptant dans* $\mathbb{A}$ *pour le mot* $(a_i)_{i\in\mathbb{N}}$ *est une suite d'états* $(q_i)_{i\in\mathbb{N}}$ *telle que*

- *pour tout* $i$, $(q_i, a_i, q_{i+1}) \in \delta$,
- $q_0 \in I$,
- *pour chaque* $j \in [\![1; n]\!]$ *il existe une infinité d'entiers* $i$ *tels que* $q_i \in F_j$.

On dira d'un mot pour lequel existe un calcul acceptant dans $\mathbb{A}$ qu'il est accepté par $\mathbb{A}$. L'ensemble des mots pour lesquels il existe un calcul acceptant dans $\mathbb{A}$ est le langage de cet automate, que l'on note $\mathrm{L}(\mathbb{A})$. Étant donné un automate $\mathbb{A}$, décider du vide de son langage, c'est-à-dire savoir si $\mathrm{L}(\mathbb{A}) = \emptyset$, se fait en NLOGSPACE. Nous utiliserons ce résultat dans la section sur la complexité.

## 5.2 Automate de réalisation

Cet automate accepte les modèles symboliques, qui sont des mots, admettant un modèle concret. En particulier, comme il n'y aura pas qu'une seule lettre dans le modèle symbolique, on va vouloir vérifier que celles-ci s'enchaînent bien, de proche en proche. Nous montrerons ensuite que cette condition suffit à l'existence d'un modèle concret ayant pour abstraction le modèle reconnu par l'automate.

### 5.2.1 Définition

Définissons l'automate $\mathbb{A}^*_\Phi$, l'*automate de réalisation*. Il dépend de la mesure $\mu_\Phi$, de la formule $\Phi$.

L'alphabet est l'ensemble $\Sigma_\mu$ des parties maximalement consistantes de $\mathcal{L}_\mu$.

Les états $Q$ sont toutes les lettres, plus un état initial $i$.

Une transition $(X, Y, Z)$ est dans la table de transition $\delta$ ssi $Y = Z$ $X$ et $Y$ vérifient pour $u, v < m$ la condition $(A)$ :

$\mathsf{X}^{u+1}\mathtt{x} = \mathsf{X}^{v+1}\mathtt{y} \in X$ ssi $\mathsf{X}^{u}\mathtt{x} = \mathsf{X}^{v}\mathtt{y} \in Y$.

Quant à l'état initial, $(i, Y, Y) \in \delta$ pour toute lettre.

La condition d'acceptation $\mathcal{F}$ n'est constituée que d'un ensemble $F$, égal à l'ensemble des états de l'automate.

### 5.2.2 Correction

Nous souhaitons être certains de l'efficacité de cet automate pour la tâche qu'il doit accomplir : vérifier l'existence d'un modèle concret.

**Lemme 2** *Un modèle symbolique* $\sigma$ *est accepté par* $\mathbb{A}^*_\Phi$ *ssi il admet un modèle concret* $\rho$ *tel que* $Abs_\mu(\rho) = \sigma$.

Ce lemme est démontré en annexe 9.

## 5.3 Automate de contraintes temporelles

Il reconnaît les modèles symboliques satisfaisant symboliquement la formule dont on cherche à connaître la satisfaisabilité. La méthode utilisée pour s'assurer des conditions temporelles est déjà connue par [VW94, DD02], nous nous attarderons peut-être plus sur la satisfaction des formules d'état.

### 5.3.1 Définitions préliminaires

Soit $sfc(\Phi)$ l'ensemble des sous-formules de la formule $\Phi$, étant exclues les sous-formules d'état ($\Phi$ est dans cet ensemble ; et $\phi_\epsilon$, $\epsilon \in \{1;2\}$, y est dès que $\neg\phi_\epsilon$ ou $\phi_1 \wedge \phi_2$ ou $\mathsf{X}\phi_\epsilon$ ou $\phi_1 \mathsf{U} \phi_2$ y est aussi). On définit maintenant l'ensemble $cl(\Phi)$ :

$$smallcl(\Phi) = sfc(\Phi) \cup \{\mathsf{X}(\phi \mathsf{U} \phi'), \phi \mathsf{U} \phi' \in sfc(\Phi)\}$$
$$cl(\Phi) = smallcl(\Phi) \cup \{\neg\phi, \phi \in smallcl(\Phi)\}$$

Enfin, un atome de $cl(\Phi)$ est un sous-ensemble $q$ de $cl(\Phi)$ tel que :

$$\neg\phi \in q \text{ ssi non } \phi \in q$$
$$\phi \wedge \phi' \in q \text{ ssi } \phi \in q \text{ et } \phi' \in q$$
$$\phi \mathsf{U} \phi' \in q \text{ ssi } \phi' \in q \text{ ou } (\phi \in q \text{ et } \mathsf{X}(\phi \mathsf{U} \phi') \in q)$$

### 5.3.2 L'automate

Définissons l'automate $\mathbb{A}^s_\Phi$, l'automate de contraintes temporelles

L'alphabet est le même que celui de $\mathbb{A}^*_\Phi$ : l'ensemble des parties maximalement consistantes de $\mathcal{L}_\mu$.

Les états (dont l' ensemble est $Q$) sont les atomes de $cl(\Phi)$.

Une transition $(q, a, q')$ existe (dans la table de transitions $\delta$) ssi

- pour toute formule d'état $\mathcal{A} \in cl(\Phi)$, $\mathcal{A} \in q$ ssi $a \models_s \mathcal{A}$ ; et
- pour toute formule $\mathsf{X}\phi \in cl(\Phi)$, $\mathsf{X}\phi \in q$ ssi $\phi \in q'$.

Les états initiaux sont les atomes contenant $\Phi$.

Soit $\{\phi_1 \mathsf{U} \phi'_1, \ldots, \phi_l \mathsf{U} \phi'_l\}$ l'ensemble des formules de $cl(\Phi)$ de la forme $\phi \mathsf{U} \phi'$. La condition d'acceptation est : $\mathcal{F} = (F_1, \ldots, F_l)$, où

$$F_i = \{q, \neg(\phi_i \mathsf{U} \phi'_i) \in q \text{ ou } \phi'_i \in q\}$$

### 5.3.3 Correction

Nous devons montrer le lemme :

**Lemme 3** *Soit $(a_i)_{i\in\mathbb{N}}$ un modèle symbolique. Alors ce modèle est reconnu par $\mathbb{A}^s_\Phi$ si et seulement si $(a_i)_{i\in\mathbb{N}} \models_\mu \Phi$.*

La preuve de ce lemme, utilisant une méthode classique, se trouve en annexe 10.

# 6 Complexité

Nous allons maintenant montrer que les automates définis dans la précédente section permettent de décider en espace polynomial de la satisfaisabilité de $\Phi$. Tout d'abord, nous allons coder les ensembles de formules que sont les lettres de façon plus succinte, puis nous nous intéresserons à la complexité des transitions des automates.

## 6.1 Codage

Nous souhaitons maintenant nous assurer que les objets obtenus par l'abstraction sont manipulables, utilisables sans complexité excessive par rapport à notre souhait de rester dans PSPACE. En effet, les parties de $\mathcal{L}_\mu$, ainsi que les éléments de l'ensemble $M_s$, peuvent contenir un nombre exponentiel de formules. Cependant, l'information contenue dans ces formules est redondante dans le cas des éléments de $M_s$. Nous allons donc définir un codage de ces parties maximalement consistantes, puis nous nous assurerons que ce codage satisfait des propriétés permettant de le manipuler dans un algorithme PSPACE

Définissons pour commencer une propriété des parties de $\mathcal{L}_\mu$, que nous appelerons propriété $P$.

**Propriété:**
Un ensemble $X$ de formules de $\mathcal{L}_\mu$ est maximalement consistant ssi

- *P1 :* "=" est une relation d'équivalence, et :
  - si $\mathtt{alloc}\ \mathtt{x} \in X$ et $\mathtt{x} = \mathtt{y} \in X$ alors $\mathtt{alloc}\ \mathtt{y} \in X$.
  - si $\mathtt{x} \hookrightarrow \mathtt{x}_1, \ldots, \mathtt{x}_n \in X$ et $\mathtt{x} = \mathtt{y} \in X$ alors $\mathtt{y} \hookrightarrow \mathtt{x}_1, \ldots, \mathtt{x}_n \in X$.
- *P2 :* si $\mathtt{x} \hookrightarrow \mathtt{x}_1, \ldots, \mathtt{x}_n \in X$ alors $\mathtt{alloc}\ \mathtt{x} \in X$.
- *P3 :* $\mathtt{x} \hookrightarrow \mathtt{x}_1, \ldots, \mathtt{x}_n \in X$ lorsque : $\mathtt{x} \hookrightarrow \mathtt{y}_1, \ldots, \mathtt{y}_m \in X$ ssi
  - $m$ est égal à $n$ et
  - les formules $\mathtt{x}_i = \mathtt{y}_i$ appartiennent à $X$ pour tout $i \leq m$.
- *P4 :* il exite un $k_0$ tel que :
  - $\mathtt{size}\ \geq\ k$ appartient à $X$ pour $k \geq k_0$ et n'y appartient pas sinon ;
  - $k_0$ est égal à $w$ ou plus grand que le nombre de variables non égales allouées, c'est-à-dire plus grand que le plus grand $n$ tel qu'il existe $\mathtt{x}_1 \ldots \mathtt{x}_n$ non égales avec $\mathtt{alloc}\ x_i \in X$ pour $i \leq n$.

**Démonstration:**
Si $X$ est maximalement consistant, il en existe un modèle partiel $(s_0 \ldots s_m, h)$. Alors

- l'égalité y est une relation d'équivalence, et :
  - si une variable est allouée ($s_0(\mathtt{x}) \in \mathtt{dom}(h)$), toute variable égale ($s_0(\mathtt{x}) = s_0(\mathtt{y})$) l'est aussi ($s_0(\mathtt{y}) \in \mathtt{dom}(h)$).
  - si une variable pointe vers quelque chose, un autre variable égale pointe vers cette même chose, c'est-à dire, si $s_0(\mathtt{x}) = s_0(\mathtt{y})$ alors $h(s_0(\mathtt{x})) = h(s_0(\mathtt{y}))$.
- une variable pointant vers quelque chose est allouée, c'est-à-dire, si $h(s_0(\mathtt{x})) = \_$, alors $s_0(\mathtt{x}) \in \mathtt{dom}(h)$.

– $h$ étant une fonction dont le domaine est inclus dans $\mathtt{Val}$,
lorsque $\mathtt{x} \hookrightarrow \mathtt{x}_1, \ldots, \mathtt{x}_n \in X$, on a $h(s_0(\mathtt{x})) = s_0(\mathtt{x}_1), \ldots, s_0(\mathtt{x}_n)$,
et $\mathtt{x} \hookrightarrow \mathtt{y}_1, \ldots, \mathtt{y}_m \in X$ ssi :
  - $m$ est égal à $n$ et
  - $s_0(x_i) = s_0(y_i)$ pour $i \leq m$.

– il exite un entier $k$ plus grand que $|\mathtt{dom}(h)|$.
on prend $k_0$ le plus petit de $k_0$ et $w$.

Inversement, supposons qu'un sous-ensemble $X$ de $\mathcal{L}_\mu$ vérifie ces contraintes. Alors, construisons un modèle partiel. Soit $i$ le nombre de classes d'équivalences de l'égalité, bien définies par *P1*. Numérotons ces classes de 1 à $i$.

Choisissons arbitrairement $i$ éléments distincts $l_1 \ldots l_i$ de $\mathtt{Loc}$. Définissons l'image par $s_k$ de la variable $\mathtt{x}$ par $l_j$ si $j$ est le numéro de la classe d'équivalence de $\mathsf{X}^k\mathtt{x}$. Cela suffit à définir $s_0 \ldots s_m$. Cela permet de satisfaire les $e = e'$ voulus, et pas d'autre.

Maintenant, si un élément $\mathtt{x}$, ou plus précisément $\mathsf{X}^0\mathtt{x}$, de la $j^{\text{ème}}$ classe d'équivalence pointe vers des variables des classes $j_1 \ldots j_m$, c'est-à-dire $\mathtt{x} \hookrightarrow \mathtt{x}_1, \ldots, \mathtt{x}_m \in X$ avec $\mathtt{x}_u$ dans la $u^{\text{ème}}$ classe et dans aucune autre, tout cela par *P3* ; alors il en est de même pour tous les éléments de cette classe grâce aux sous-propriétés de *P1*, et on définit $h(l_j) = l_{j_1} \ldots l_{j_m}$. Cela permet de satisfaire les $\mathtt{x} \hookrightarrow \mathtt{x}_1, \ldots, \mathtt{x}_m$ voulus.

De même, soit un élément $\mathtt{x}$ de la $j^{\text{ème}}$ classe d'équivalence, qui ne pointe vers rien de défini : aucune formule $\mathtt{x} \hookrightarrow \mathtt{x}_1, \ldots, \mathtt{x}_n$ n'est vraie. Alors il en est de même pour toute sa classe d'équivalence. Soit $l \in \mathtt{Loc}$ distinct des $l_u$ Il reste à définir : si $\mathtt{alloc}\ x \in X$ alors $h(l_j) = l$ ; et, sinon, $h$ n'est pas défini en $l_j$. Cela permet de satisfaire les $\mathtt{alloc}\ \mathtt{x}$ voulus.

Enfin, soit $p = \max\{k, \mathtt{size}\ \geq\ k \in X\} - |\mathtt{dom}(h)|$. Si $p$ est négatif ou nul, c'est que $|\mathtt{dom}(h)| \geq w$ et $\max\{k, \mathtt{size}\ \geq\ k \in X\} \geq w$, les formules $\mathtt{size}\ \geq\ k$ satisfaites sont donc celles voulues. Sinon, soient $p$ éléments distincts de $\mathtt{Loc}$, différents de $l_1 \ldots l_i$ ; nommons les $l'_1 \ldots l'_p$. Alors définissons $h(l'_u) = nil$. Cela permet de satisfaire les $\mathtt{size}\ \geq\ k$ voulus.

Ce modèle permet d'assurer la consistance maximale, c'est-à-dire que les stores et le tas construits satisfont exactement les formules de $X$, et $X \in \Sigma_\mu$. Notons qu'il n'y a pas plus de formules satisfaites que ce qui est souhaité, ce qu'assure le choix de $l$ et des $l'_u$ différents des adresses déjà utilisées.
□

**Le codage**

Ainsi, une fois l'égalité codée par une partition des expressions $\mathtt{Var} \times \{0 \ldots m\}$, l'ensemble des variables aux instants accessibles auxquelles il faut ajouter $\mathtt{null}$[3], "$\hookrightarrow$" peut être vu comme une fonction des classes de variables dans les mots de classes de variables (de longueur au plus $\eta$), et codée en espace polynomial.

Pour être précis, il suffit, d'abord, de coder l'égalité, par un ensemble de paires d'expressions c.à.d. de variables et instants, pour ensuite d'écrire un ta-

[3] Il m'arrive de ne pas traiter le cas de $\mathtt{null}$, puisqu'il peut être considéré comme une variable spéciale, toujours constante.

bleau de taille $n$, où $\mathtt{Var} = \{\mathtt{x}_1, \ldots, \mathtt{x}_n\}$ et donc $n$ est le nombre de variables, dont la case $j$, correspondant à $\mathtt{x}_j$, est remplie par, selon les cas :

- une première valeur spéciale ("$Nalloc$") si l'expression $\mathtt{x}_j$ n'est pas allouée,
- une seconde valeur spéciale ("$alloc$") si c'est alloué mais ne pointe vers rien de défini ($\mathtt{alloc}\ \mathtt{x}_j$, mais non $\mathtt{x}_j \hookrightarrow \_$),
- un mot de variables $\mathtt{x}_{j_1}, \ldots, \mathtt{x}_{j_u}$ (des noms de cases du tableau) de longueur bornée par $\eta$, qui sont choisies comme des représentantes de leur classe ; la représentante de sa classe est la première dans l'ordre $\mathtt{x}_1, \ldots \mathtt{x}_n$.

Enfin, un entier plus petit que $w$ représente la taille de la mémoire, le "$k_0$", codé en binaire dans un espace polynomial.

Pour résumer, notre représentation, ou codage, d'une lettre est un élément de l'ensemble $\mathcal{X}$ suivant :

$$\mathcal{P}([(\mathtt{Var} \times \{0 \ldots m\}) \cup \mathtt{null}]^2) \times [\mathtt{Var} \to (\{Nalloc, alloc\} \bigcup_{i \leq \eta} \{\mathtt{Var}\}^i)] \times \{0 \ldots w\}$$

L'important, c'est que notre codage est de taille polynomiale, ce que l'ensemble ci-dessus prouve. Nous pouvons énoncer cela en deux propositions.

**Proposition 1** *Un élément de $\mathcal{X}$ peut être représenté en espace polynomiale en la taille de $\Phi$ si $\mu = \mu_\Phi$.*

**Proposition 2** *Si $X \in \Sigma_\mu$ alors il existe un élément de $\mathcal{X}$ le représentant.*

Nous souhaitons aussi pouvoir vérifier qu'un codage représente une lettre.

**Proposition 3** *Si $C \in \mathcal{X}$ alors on peut savoir en espace polynomial si il existe un élément de $\Sigma_\mu$ que $C$ représente.*

**Démonstration:**
Soit donc $C$ un "codage" : $(P, f, t) \in \mathcal{X}$. Nous vérifions alors les conditions suivantes :

- $P$ représente une relation d'équivalence : si $(\mathtt{x}_i, \mathtt{x}_j) \in P$ alors $(\mathtt{x}_j, \mathtt{x}_i) \in P$, si $(\mathtt{x}_i, \mathtt{x}_j) \in P$ et $(\mathtt{x}_j, \mathtt{x}_u) \in P$ alors $(\mathtt{x}_i, \mathtt{x}_u) \in P$, et pour tout $\mathtt{x}_i$ $(\mathtt{x}_i, \mathtt{x}_i) \in P$.
- $f$ est cohérent avec l'égalité : si $(\mathtt{x}_i, \mathtt{x}_j) \in P$ alors $f(\mathtt{x}_i) = f(\mathtt{x}_j)$.
- $f$ choisit les bons représentants : si $f(\mathtt{x}_j) = \mathtt{x}_{j_1}, \ldots, \mathtt{x}_{j_u}$ alors pour chaque $j_i$, il n'existe pas de $\alpha \leq j_i$ tel que $(\mathtt{x}_\alpha, \mathtt{x}_{j_i}) \in P$.
- la taille de la mémoire est plus grande que le nombre de classes de variables allouées : $t \leq \mathrm{card}(\{\mathtt{x}_i \in \mathtt{Var}, f(\mathtt{x}_i) \neq Nalloc$ et $(\mathtt{x}_i, \mathtt{x}_j) \notin P$ pour tout $j < i\})$, ou égale à $w$.

Chacune de ces conditions peut être vérifiée en temps donc en espace polynomial.

Il existe alors un ensemble de formules $X$ de $\mathcal{L}_\mu$ correspondant :

- $\mathtt{x}_i = \mathtt{x}_j \in X$ ssi $(\mathtt{x}_i, \mathtt{x}_j) \in P$
- $\mathtt{size} \geq k \in X$ ssi $k \leq t$,
- $\mathtt{alloc}\ \mathtt{x}_j \in X$ ssi $f(\mathtt{x}_j) \neq Nalloc$,
- $\mathtt{x}_j \hookrightarrow \mathtt{x}_{j_1}, \ldots, \mathtt{x}_{j_u} \in X$ (ce qui est possible puisque $u \leq \eta$) ssi, pour $\mathtt{x}_{i_1}, \ldots, \mathtt{x}_{i_u}$ les variable représentant respectivement $\mathtt{x}_{j_1}, \ldots, \mathtt{x}_{j_u}$, $f(\mathtt{x}_j) = \mathtt{x}_{i_1}, \ldots, \mathtt{x}_{i_u}$.

Les conditions de la propriété $P$(caractérisant les ensembles maximalement consistants) sont effectivement vérifiées. Ce que je ne montrerai pas. Cela permet d'affirmer que $X$ est bien une lettre de $\Sigma_\mu$. $\square$

Enfin, on veut pouvoir accéder à l'information codée.

**Proposition 4** *Soit $C \in \mathcal{X}$ représentant un élément $X$ de $\Sigma_\mu$, et $\mathcal{A} \in \mathcal{L}_\mu$. Il est possible, connaissant $C$, de savoir si $\mathcal{A} \in X$ en espace polynomial.*

**Démonstration:**
Il suffit de suivre la procédure indiquée dans la précédente preuve pour connaître l'ensemble de formules correspondant à un codage, en ne s'intéressant cette fois-ci qu'à une formule dont on vérifie qu'elle apparaît dans l'énumération. Ce qui peut se faire en espace polynomial. $\square$

Par conséquent, nous pouvons travailler en ne nous souciant que de la définition initiale de la consistance maximale : nous retenons simplement qu'une lettre de $\Sigma_\mu$ peut être codée en espace polynomial, et quelques autres propositions pour leur utilisation.

Nous avons une représentation de nos lettres, mais un modèle symbolique ne sera plus considéré que comme une suite d'ensembles maximalement consistants de formules de $\mathcal{L}_\mu$.

## 6.2 Automates

Il faut que leurs transitions se calculent en espace polynomial et que leurs états se stockent en espace polynomial.

### 6.2.1 Réalisation

**Lemme 4** *Une transition de $\mathbb{A}^*_\Phi$ peut se vérifier en espace polynomial en la taille de $\Phi$, et un état de cet automate peut être stocké en espace polynomial.*

**Démonstration:**
Notons qu'un état peut être stocké en espace polynomial, et une lettre aussi. L'automate a donc une taille au plus exponentielle. Quant aux transitions, elles se vérifient aisément à partir du codage défini dans la précédente section : chacune des lettres contient une partition de $\texttt{Var} \times [\![0; m]\!]$ représentant l'égalité. Pour vérifier la condition souhaitée, il suffit de comparer la troncature de la première (chronologiquement) à $\texttt{Var} \times [\![1; m]\!]$ et de la seconde à $\texttt{Var} \times [\![0; m-1]\!]$, pour vérifier qu'elles sont "identiques", par exemple en enlevant 1 au second élément de chaque paire $(\texttt{x}, i)$ de la partition tronquée à $\texttt{Var} \times [\![1; m]\!]$ de la première lettre. Cela se fait en temps polynomial, et donc en espace polynomial. $\square$

### 6.2.2 Contraintes temporelles

**Lemme 5** *Une transition de $\mathbb{A}^s_\Phi$ peut se vérifier en espace polynomial, et un état de cet automate peut être stocké en espace polynomial.*

**Démonstration:**

- taille des états : $\Phi$ a un nombre polynomial de sous-formules. Prendre le "next" des "until" reste polynomial, et ajouter la négation aussi.
- calcul des transitions : le lemme suivant (5.1) prouve que l'on peut vérifier en espace polynomial la première condition pour qu'une transition soit possible ("*pour toute formule d'état* $\mathcal{A} \in cl(\Phi)$, $\mathcal{A} \in q$ *ssi* $a \models_{\mathrm{s}} \mathcal{A}$"), en énumérant toutes les formules d'état de $cl(\Phi)$, et les testant. Et la seconde condition ("*pour toute formule* $\mathsf{X}\phi \in cl(\Phi)$, $\mathsf{X}\phi \in q$ *ssi* $\phi \in q'$") se vérifie en temps donc en espace polynomial en énumérant les formules de l'état d'origine, qui est de taille polynomiale, et vérifiant que pour chacune de celles de la forme $\mathsf{X}\phi$, la formule $\phi$ apparaît dans l'état d'arrivée.

□

Nous devons alors prouver le lemme 5.1. La preuve en extension se trouve en annexe 11.

**Lemme 5.1** *Soit* $a \in \Sigma_\mu$. *Il est possible de vérifier en espace polynomial la validité de "*$a \models_{\mathrm{s}} \mathcal{A}$*", pour* $a$ *et* $\mathcal{A}$ *données.*

L'outil principal de cette preuve est une forme d'union disjointe sur les modèles symboliques.

**Définition 14** *Soit* $a$, $a_1$ *et* $a_2$ *trois éléments de* $\Sigma_\mu$. *Nous écrivons* $a = a_1 * a_2$ *lorsqu'il existe deux modèles partiels* $s_0 \dots s_m, h^1$ *et* $s_0 \dots s_m, h^2$, *de respectivement* $a_1$ *et* $a_2$, *tels que* $s_0 \dots s_m, h^1 * h^2$ *soit un modèle partiel de* $a$.

Nous devrons nous assurer que cette union disjointe pour modèles symbolique est utilisable dans un algorithme en espace polynomial. Voici les deux lemmes que nous utiliserons à ce sujet.

**Lemme 5.1.1** *Il est possible de tester en espace polynomial si* $a = a' * a''$.

**Lemme 5.1.2** *Soit une lettre* $a$. *Il existe* $a'$ *et* $a''$ *tels que* $a = a' * a''$ *et* $a' \models \mathcal{A}'$ *et* $a'' \models \mathcal{A}''$ *ssi* $a \models \mathcal{A}' * \mathcal{A}''$.

## 6.3 Complexité globale

Il est possible de décider en NPSPACE du "non-vide" d'un automate de Büchi généralisé de taille exponentielle défini implicitement et dont les transitions se calculent en espace polynomial, en prenant un parcours non déterministe dans l'automate de longueur $|\mathcal{F}| \times |Q|$[4], stockant un état choisi de façon non déterministe, et vérifiant que le parcours choisi passe à nouveau par cet état après une boucle passant par un état de chacun des ensembles de la condition d'acceptation : il existe un calcul acceptant s'il existe un tel parcours. Ce qui nous donne un algorithme non déterministe. On en déduit par le théorème de Savitch,

[4]le nombre d'ensembles dans la condition finale multiplié par le nombre d'états de l'automate

PSPACE=NPSPACE, qu'il existe un algorithme PSPACE pour décider du "non-vide" d'un automate. Et comme PSPACE=co-PSPACE, il existe un algorithme PSPACE décidant du vide d'un automate de Büchi généralisé.

En prenant l'intersection des deux automates que nous avons construit, nous obtenons un automate de Büchi généralisé acceptant un mot ssi la formule $\Phi$ est satisfaisable, comme le montrent les lemmes 1, 2 et 3. Et comme il est possible de calculer une transition en espace polynomial, grâce aux lemmes 4 et 5, la satisfaisabilité dans $\mathrm{LTL}^{\mathrm{mem}}$ est PSPACE. C'est le théorème 1.

# 7 Perspectives

Nous avons montré que la composition des formalismes de LTL et la logique de séparation est pertinente, puisqu'elle conduit à un formalisme décidable, dans une classe de complexité optimale par rapport à ce qui pouvait être espéré au vu de la complexité des ingrédients choisis.

L'obtention de ce résultat de complexité nous conforte donc dans l'utilisation d'une combinaison de logiques exprimant des contraintes temporelles et sur la mémoire, sachant qu'à l'exception du travail présenté dans [DKR04], il n'existe pas d'autre utilisation d'une logique temporelle pour spécifier des propriétés sur la mémoire.

Dans mon travail, la complexité montrée est voulue comme un premier pas vers un langage plus expressif. Les travaux existants sur la logique LTL montrent que ce résultat peut être étendu à d'autres opérateurs temporels, imposant des conditions sur le passé, ou encore calculant des points fixes. Par contre, l'ajout d'expressivité sur le plan de la mémoire, pour obtenir des propriétés plus contraignantes comme par exemple la constance du tas ou encore exprimer explicitement l'allocation de mémoire, peut parfois mener à l'indécidabilité. L'avenir sera donc de trouver un compromis entre expressivité et complexité.

# Références


[BCO05] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. *APLAS*, 3780 :52–68, 2005.

[BFL04] S. Bardin, A. Finkel, and J. Leroux. Faster acceleration of counter automata in practice. *TACAS*, pages 576–590, March 2004.

[BFLS04] S. Bardin, A. Finkel, E. Lozes, and A. Sangnier. From pointer systems to counter systems using shape analysis. *AVIS*, 2004.

[BFN06] S. Bardin, A. Finkel, and D. Nowak. Towards symbolic verification of programs handling pointers. *AVIS*, 2006. à paraître.

[BHV04] A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract regular model-checking. *CAV*, pages 372–386, 2004.

[BIL03] M. Bozga, R. Iosif, and Y. Lakhnech. Storeless semantics and alias logic. *PEPM'03*, pages 55–65, 2003.

[CC00] H Comon and V. Cortier. Flatness is not a weakness. *14 Int. Workshop Computer Science Logic*, 1862 :262–276, 2000.

[CS85] E. M. Clarke and A. P. Sistla. The complexity of propositional linear temporal logic. *JACM*, 32(3) :733–749, 1985.

[CYO01] C. Calcagno, H. Yang, and P. W. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. *APLAS*, pages 289–300, 2001.

[DD02] S. Demri and D. D'Souza. An automata-theoretic approach to constraint LTL. *FST&TCS'02, Kanpur*, 2256 :121–132, 2002.

[DEG06] J. V. Deshmukh, E. A. Emerson, and P. Gupta. Automatic verification of parametrized data structures. *TACAS*, pages 27–41, 2006.

[DKR04] D. Distefano, J. P. Katoen, and A. Rensink. Who is pointing when to whom ? *FSTTCS*, pages 250–262, 2004.

[HIV06] P. Habermelh, R. Iosif, and T. Vojnar. Automata-based verification of programs with tree updates. *TACAS*, pages 350–364, 2006.

[JJKS97] J. L. Jensen, M. E. Jorgensen, N. Klarlund, and M. I. Schwartzbach. Automatic verification of pointer programs using monadic second order logic. *PDLI*, pages 226–236, 1997.

[LAMS04] T. Lev-Ami, R. Manevich, and M. Sagiv. Tvla : A system for generating abstract interpreters. *IFIP Congress Topical Sessions*, pages 367–376, 2004.

[Loz04] E. Lozes. Separation logic preserves the expressive power of classical logic. *Proceedings of the 2nd Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE'04)*, 2004.

[MS01] A. Moller and M. I. Schwartzbach. The pointer assertion logic engine. *PDLI*, pages 221–231, 2001.

[Pnu77] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society Press, 1977.

[Rey02] J. C. Reynolds. Separation logic : a logic for shared mutable data structures. *LICS*, pages 55–74, 2002.

[SRW98] M. Sagiv, T. W. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Program. Lang. Syst.*, 20(1) :1–50, 1998.

[VW94] M. Vardi and P. Wolper. Reasoning about infinite computations. 115 :1–37, 1994.

[YRS+05] G. Yorsh, A. M. Rabinovich, M. Sagiv, A. Meyer, and A. Bouajjani. A logic of reachable patterns in linked data structures. *FoSSaCS*, pages 94–110, 2005.

# Troisième partie

# Annexe : preuves

## 8 Bon comportement de $\simeq_{\mu,i}$

Nous voulons montrer trois lemmes préliminaires, pour finalement prouver le lemme 1.2.

**Distributivité** Commençons par le lemme de distributivité :

*Si $\rho \simeq_{\mu,i} \rho'$ et s'il existe $h^1$ et $h^2$ tels que $h^1 * h^2 = h_i$ (le $i^{\text{ème}}$ tas du modèle $\sigma$), alors : il existe $h'^1$ et $h'^2$ tels que $h'^1 * h'^2 = h'_i$ (le $i^{\text{ème}}$ tas du modèle $\sigma'$) et pour $u \in \{1,2\}$ on a $\rho[i \leftsquigarrow h^u] \simeq_{\mu/2,i} \rho'[i \leftsquigarrow h'^u]$.*

**Démonstration:**

Nous souhaitons donc construire $h'^1$ et $h'^2$, le reste étant donné.

Tout d'abord, notons que les égalités sont conservées d'un modèle à l'autre puisque le store[5] est conservé, quel que soit la construction des tas $h'^u$. Les classes d'équivqlence entre variables restent les mêmes, et $\rho[i \leftsquigarrow \_]$ satisfait les mêmes "$e_1 = e_2$" que $\rho$. Occupons nous des autres expressions atomiques.

Revenons donc à la construction des $h'^u$. Posons $h^u = def^u * garb^u$. $def^u$, a pour domaine de définition la partie de celui de $h^u$ correspondant par $s_i$ à des variables : $\mathtt{Im}(s_{i\restriction V}) \cap \mathtt{dom}(h^u)$, où $V$ est tiré de la mesure $\mu$. Et $garb^u$ a pour domaine $\mathtt{dom}(h^u) \backslash \mathtt{Im}(s_{i\restriction V})$, c'est-à-dire ce qui reste. Nous allons donc construire symétriquement $def'^u$ et $garb'^u$, de domaines de définition disjoints.

Pour $def'^u$, commençons par noter $A^u = s_i^{-1}(\mathtt{dom}(h^u))$, l'ensemble des variables allouées dans $\rho[i \leftsquigarrow h^u]$, à l'instant $i$. Alors $def'^u$ est définie comme la restriction de $h'_i$ à $s'_i(A^u)$. Nous avons donc décidé que $\rho'[i \leftsquigarrow h'^u], i \models \mathtt{alloc\ x}$ serait vrai exclusivement lorsque $s'_i(\mathtt{x}) \in \mathtt{dom}(def'^u)$, c'est-à-dire lorsque $\mathtt{x} \in A^u$. Or, cela équivaut à $s_i(\mathtt{x}) \in \mathtt{dom}(def^u)$, ce qui est vrai ssi $s_i(\mathtt{x}) \in \mathtt{dom}(h^u)$, autrement dit $\rho[i \leftsquigarrow h^u], i \models \mathtt{alloc\ x}$. Les $def'^u$ permettent donc de conserver les assertions concernant "$\mathtt{alloc}$ ". De même, $\rho'[i \leftsquigarrow h'^u], i \models \mathtt{x} \hookrightarrow \mathtt{x}_1 \dots \mathtt{x}_\alpha$ sera vrai lorsque $def'^u(s'_i(\mathtt{x})) = def'^u(\mathtt{x}_1) \dots def'^u(\mathtt{x}_\alpha)$. C'est-à-dire lorsque $\mathtt{x} \in A^u$ et $h'_i(s'_i(\mathtt{x})) = h'_i(\mathtt{x}_1) \dots h'_i(\mathtt{x}_\alpha)$, ce qui est vrai ssi $h'_i(s'_i(\mathtt{x})) = h'_i(\mathtt{x}_1) \dots h'_i(\mathtt{x}_\alpha)$, ssi $h^u(s'_i(\mathtt{x})) = h^u(\mathtt{x}_1) \dots h^u(\mathtt{x}_\alpha)$, ssi $\rho[i \leftsquigarrow h^u], i \models \mathtt{x} \hookrightarrow \mathtt{x}_1 \dots \mathtt{x}_\alpha$ Les $def'^u$ permettent donc de conserver les assertions concernant "$\hookrightarrow$".

Il faut maintenant définir les $garb'^u$. Il reste une partie du domaine de $h'_i$ qui n'est couverte ni par celui de $def'^1$, ni par celui de $def'^2$. Il faut la répartir intelligemment entre $garb'^1$ et $garb'^2$, de façon à satisfaire les "$\mathtt{size} \ \geq \ k$". Pour cela, observons l'abstraction de $h^1$ et $h^2$ (ou plutôt de $\rho[i \leftsquigarrow h^u]$). Soient $k^1$ et $k^2$ les limites respectives pour $k$ de validité de "$\mathtt{size} \ \geq \ k$" de $\rho[i \leftsquigarrow h^1]$ et $\rho[i \leftsquigarrow h^2]$ ; formellement, $k^u = \min(|\mathtt{dom}(h^u)|, w/2)$, ce qui est aussi $k^u = \max\{k, k \leq w/2$ et $\rho[i \leftsquigarrow h^u] \models \mathtt{size} \ \geq \ k\}$. Soit $k^0$ celle de $h_i$. Soient enfin $l^u$ les tailles des domaines des $def'^u$, égales à celles des $def^u$. Trois cas sont possibles.

---

[5] les stores en réalité : $s'_i \dots s'_{i+m}$

- Si $k^1$ et $k^2$ sont tous deux inférieurs à $w/2$, alors $k^0 = k^1 + k^2$ est inférieur à $w$, et $k^u \geq l^u$ car $k^u$ représente précisément la taille de la mémoire. On peut choisir $k^1 - l^1$ éléments de $\mathtt{dom}(h'_i)$ au hasard, et décider que le domaine de $garb'^1$ est l'ensemble de ces éléments. La taille du domaine de $h'^1 = def'^1 * garb^1$ est alors $k^1$. $garb'^2$ sera défini sur la partie du domaine de $h'_i$ non encore traitée, ce qui nous amènera bien à $k^2$.
- Si $k^1$ et $k^2$ sont tous deux égaux à $w/2$, alors $k^0$ est égal à $w$. Si $k^u \geq l^u$, on met alors $w/2 - l^u$ éléments dans le domaine de $garb'^u$ (ce qui est possible car $k^0$ est suffisament grand). Ensuite, dans tous les cas, s'il en reste, on les ajoute, mettons, au domaine de $garb'^1$.
- Enfin, si seul l'un d'eux est inférieur à $w/2$, disons $k^1$, on met le nombre exact ($k^1 - l^1$, positif car $k^1$ représente précisément la taille de la mémoire) d'adresses qu'il faut dans $garb'^1$ pour avoir un domaine de $h'^1$ de taille égale à celui de $h^1$. Et , peu importe le sort de $k^0$ par rapport à $w$, toutes les autres adresses sont placées dans le domaine de $garb'^2$, ce qui permet de dépasser $w/2$, car $k^0$ est commun à $h_i$ et $h'_i$, et $k^1$ à $h^1$ et $h'^1$.

Il résulte de tout cela que les abstractions de $\rho[i \leftarrow\!\!\sim h^u]$ et $\rho'[i \leftarrow\!\!\sim h'^u]$ vont être les mêmes en $i$. $\square$

**Lemme 1.2.2** Cette fois-ci, nous voulons montrer :
*Si $\rho \simeq_{\mu,i} \rho'$ et $h$ est un tas quelconque, alors il existe $h'$ tel que $\rho[i \leftarrow\!\!\sim h] \simeq_{\mu,i} \rho'[i \leftarrow\!\!\sim h']$.*

**Démonstration:**
De même que pour $h^u$ dans le précédent lemme, nous définissons $def$ et $garb$ correspondant à $h$, et nous allons en déduire $def'$ et $garb'$ correspondant à $h'$.

La construction de $def'$ est assez proche de la précédente démonstration. Cependant, comme nous ne partons pas d'un tas défini, il faut plus expliciter. Notons pour commencer que l'ensemble des variables est subdivisé en $r$ classes d'équivalences $A_1 \dots A_r$ : les classes d'équivalence pour "=", c'est-à-dire pour la relation $\mathtt{x}R\mathtt{y}$ ssi $s_i(\mathtt{x}) = s_i(\mathtt{y})$, conservée d'un modèle à l'autre (de $\rho$ à $\rho'$, mais aussi de $\rho[i \leftarrow\!\!\sim h]$ à $\rho'[i \leftarrow\!\!\sim h']$, quel que soit le choix de $h'$). Ainsi, dès que $\mathtt{alloc\ x}$ est vraie, alors, pour tout $\mathtt{y}R\mathtt{x}$, on a $\mathtt{alloc\ y}$. Dès lors, si $l_1 \dots l_r$ sont les images communes aux éléments de $A_1 \dots A_r$ par $s'_i$, nous pouvons définir : $def(l_i) = l_{i_1} \dots l_{i_n}$ ssi $\exists(\mathtt{x}, \mathtt{x}_1, \dots, \mathtt{x}_n) \in A_i \times A_{i_1} \dots A_{i_n}$ tels que $\rho[i \leftarrow\!\!\sim h], i \models \mathtt{x} \hookrightarrow \mathtt{x}_1, \dots, \mathtt{x}_n$, ce qui, grâce à la conservation des classes d'équivalence de $\rho[i \leftarrow\!\!\sim h]$ à $\rho'[i \leftarrow\!\!\sim \_]$, équivaut à $\forall(\mathtt{x}, \mathtt{x}_1, \dots, \mathtt{x}_n) \in A_i \times A_{i_1} \dots A_{i_n}$, $\rho[i \leftarrow\!\!\sim h], i \models \mathtt{x} \hookrightarrow \mathtt{x}_1, \dots, \mathtt{x}_n$.

Les "=", "$\hookrightarrow$" et "$\mathtt{alloc}$ " sont ainsi conservés de $\rho[i \leftarrow\!\!\sim h]$ à $\rho'[i \leftarrow\!\!\sim h']$.

La construction de $garb'$ est plus simple : soit $t$ la taille du domaine de $garb$, c'est-à-dire le nombre d'éléments du domaine de $h$ n'étant pas dans l'image de $s_i$ ; soient aussi $l'_1 \dots l'_t$ des adresses distinctes de $l_1 \dots l_r$, et distinctes deux à deux. Alors on définit $garb'(l_j) = nil$ pour tout $j$. Cela permet de satisfaire la conservation des "$\mathtt{size} \geq$ ".

Les abstractions de $\rho[i \leftarrow\!\!\sim h]$ et $\rho'[i \leftarrow\!\!\sim h']$ seront donc les mêmes. $\square$

**Congruence** Enfin, voici la preuve du lemme de congruence :

*Si $\rho \simeq_{\mu,i} \rho'$, $h^1 \perp h_i$, $h^2 \perp h'_i$, et $\rho[i \leftsquigarrow h^1] \simeq_{\mu,i} \rho'[i \leftsquigarrow h^2]$ alors $\rho[i \leftsquigarrow h_i * h^1] \simeq_{\mu,i} \rho'[i \leftsquigarrow h'_i * h^2]$.*

**Démonstration:**

Comme d'habitude, les "$=$" se conservent. Les "$\hookrightarrow$" et les "`alloc` " sont les mêmes des deux côtés : la réunion des deux tas orthogonaux ne permet pas d'interférence, et l'ensemble des formules satisfaites est l'union des formules satisfaites par chacun des tas. Enfin, la contrainte de taille est la même des deux côtés si les deux tas ont un domaine plus petit que $w$ (venant de $\mu$), et si l'un des deux dépasse, leur union dépassera aussi. D'où le lemme. $\square$

**Lemme 1.2** Ces propriétés permettent de finalement prouver le lemme 1.2 :

*Si $\rho \simeq_{\mu,i} \rho'$ et $\mu_{\mathcal{A}} \leqslant \mu$ alors $\rho, i \models \mathcal{A}$ ssi $\rho', i \models \mathcal{A}$.*

**Démonstration:**

Nous voulons montrer que : si $\rho \simeq_{\mu,i} \rho'$ et $\mu_{\mathcal{A}} \leqslant \mu$ alors $\rho, i \models \mathcal{A}$ ssi $\rho', i \models \mathcal{A}$.

Supposons donc, étant donnés $\rho$, $\rho'$, $\mu$ et $\mathcal{A}$, que $\rho \simeq_{\mu,i} \rho'$ et $\mu_{\mathcal{A}} \leqslant \mu$. Nous allons montrer le lemme par induction sur $\mathcal{A}$, et pour toute mesure $\mu$ telle que $\mu_{\mathcal{A}} \leqslant \mu$.

Commençons par les cas atomiques.

Si $\mathcal{A}$ est $e = e'$, avec $e = \mathsf{X}^i\mathtt{x}$ et $e' = \mathsf{X}^j\mathtt{y}$, alors, puisque $\mu_{\mathcal{A}} \leqslant \mu$, on a $\mathtt{x} \in \mathtt{Var}_{\mathcal{A}}$, $\mathtt{x}' \in \mathtt{Var}_{\mathcal{A}}$, $i \leq m_\mu$ et $j \leq m_\mu$. Ce qui signifie que $\mathcal{A} \in \mathcal{L}_\mu$ ; et donc, puisque $\rho \simeq_{\mu,i} \rho'$ (c'est-à-dire $Abs_\mu(\rho)(i) = Abs_\mu(\rho')(i)$), $\mathcal{A} \in Abs_\mu(\rho)(i)$ ssi $\mathcal{A} \in Abs_\mu(\rho')(i)$. Par définition de $Abs_\mu$ cela signifie : $\rho, i \models \mathcal{A}$ ssi $\rho', i \models \mathcal{A}$.

Si $\mathcal{A}$ est $\mathtt{x} \mapsto \mathtt{x}_1, \ldots, \mathtt{x}_n$, alors similairement au cas de l'égalité, on obtient $\mathtt{x} \hookrightarrow \mathtt{x}_1, \ldots, \mathtt{x}_n \in \mathcal{L}_\mu$. De plus, $spl(\mathcal{A}) \geq 1$, et donc $w_\mu \geq 2$, ce dont on tire : $\mathtt{size} \geq 1 \in \mathcal{L}_\mu$ et $\mathtt{size} \geq 2 \in \mathcal{L}_\mu$. On a ainsi $\mathtt{x} \hookrightarrow \mathtt{x}_1, \ldots, \mathtt{x}_n \in Abs_\mu(\rho)(i)$ et $\mathtt{size} \geq 2 \notin Abs_\mu(\rho)(i)$ ssi $\mathtt{x} \hookrightarrow \mathtt{x}_1, \ldots, \mathtt{x}_n \in Abs_\mu(\rho')(i)$ et $\mathtt{size} \geq 2 \notin Abs_\mu(\rho')(i)$. Et donc $\rho, i \models \mathtt{x} \hookrightarrow \mathtt{x}_1, \ldots, \mathtt{x}_n \wedge \neg\mathtt{size} \geq 2$ ssi $\rho', i \models \mathtt{x} \hookrightarrow \mathtt{x}_1, \ldots, \mathtt{x}_n \wedge \neg\mathtt{size} \geq 2$. Or $\mathtt{x} \hookrightarrow \mathtt{x}_1, \ldots, \mathtt{x}_n \wedge \neg\mathtt{size} \geq 2$ et $\mathtt{x} \mapsto \mathtt{x}_1, \ldots, \mathtt{x}_n$ ont la même sémantique.

Si $\mathcal{A}$ est $\mathtt{emp}$, alors $spl(\mathcal{A}) \geq 0$ et donc $w_\mu \geq 1$. Il s'ensuit $\rho, i \models \neg\mathtt{size} \geq 1$ ssi $\rho', i \models \neg\mathtt{size} \geq 1$. Or $\neg\mathtt{size} \geq 1$ et $\mathtt{emp}$ ont la même sémantique.

Nous en arrivons à l'étape inductive de l'induction.

Si $\mathcal{A}$ est $\mathcal{A}_1 * \mathcal{A}_2$, alors il faut appliquer le lemme de "distributivité". Si $\rho, i \models \mathcal{A}_1 * \mathcal{A}_2$, cela signifie qu'il existe deux tas $h^1$ et $h^2$ tels que $h = h^1 * h^2$ et $\rho[h_i \leftsquigarrow h^u], i \models \mathcal{A}_u$. Or d'après le lemme de distributivité, il existe $h'^1$ et $h'^2$ tels que $h' = h'^1 * h'^2$ et $\rho[i \leftsquigarrow h^u] \simeq_{\mu/2,i} \rho'[i \leftsquigarrow h'^u]$. De cette dernière propriété et de $\mu_{\mathcal{A}_u} \leqslant \mu_{\mathcal{A}}/2 \leqslant \mu/2$, on peut déduire $\rho'[i \leftsquigarrow h'^u], i \models \mathcal{A}_u$ par hypothèse d'induction. Ce qui nous amène, comme $h'_i = h'^1 * h'^2$, et donc $\rho'[i \leftsquigarrow h'^1 * h'^2] = \rho'$, à $\rho', i \models \mathcal{A}_1 * \mathcal{A}_2$. Un raisonnement parfaitement symétrique montrerait l'autre implication : si $\rho', i \models \mathcal{A}$ alors $\rho, i \models \mathcal{A}$.

Si $\mathcal{A}$ est $\mathcal{A}_1 \mathrel{-\!\!*} \mathcal{A}_2$, il faut appliquer le lemme de "congruence". Supposons $\rho, i \models \mathcal{A}_1 \mathrel{-\!\!*} \mathcal{A}_2$ ; cela signifie que pour tout tas $h$ tel que $h \perp h_i$, si $\rho[i \leftsquigarrow h], i \models \mathcal{A}_1$ alors $\rho[i \leftsquigarrow h_i * h], i \models \mathcal{A}_2$. Et donc, soit $h'$ tel que $\rho'[i \leftsquigarrow h'], i \models \mathcal{A}_1$. Par le lemme

1.2.2, il existe un tas $h$ tel que $\rho'[i \leftarrow\!\!\sim h'] \simeq_{\mu,i} \rho[i \leftarrow\!\!\sim h]$, d'où $\rho[i \leftarrow\!\!\sim h], i \models \mathcal{A}_1$ par hypothèse d'induction et sachant que $\mu_{\mathcal{A}_1} \leqslant \mu_{\mathcal{A}} \leqslant \mu$. Ce dont on déduit $\rho[i \leftarrow\!\!\sim h_i * h], i \models \mathcal{A}_2$, comme $\rho, i \models \mathcal{A}_1 \twoheadrightarrow\!\!* \mathcal{A}_2$. Et par le lemme de congruence, on obtient : $\rho'[i \leftarrow\!\!\sim h'_i * h'], i \models \mathcal{A}_2$. Finalement, $\rho', i \models \mathcal{A}_1 \twoheadrightarrow\!\!* \mathcal{A}_2$. L'autre implication se montre symétriquement.

Enfin, le cas des connecteurs booléens est trivial. □

# 9 Correction de l'automate de réalisation

Nous allons montrer le lemme 2 :

*Un modèle symbolique $\sigma$ est accepté par $\mathbb{A}^*_\Phi$ ssi il admet un modèle concret $\rho$ tel que $Abs_\mu(\rho) = \sigma$.*

**Démonstration:**

Supposons que $\sigma$ admette un modèle concret. Alors la contrainte souhaitée (la "condition $(A)$") est vérifiée, et donc que le modèle symbolique associé à ce modèle concret est accepté par l'automate. En effet, les égalités que l'on veut conserver correspondent au même objet mathématique : un store. Ainsi, dire que la $i^{\text{ème}}$ lettre contient la formule $\mathsf{X}^{i_1}\mathtt{x}_1 = \mathsf{X}^{i_2}\mathtt{x}_2$, c'est dire que dans le modèle concret ayant pour abstraction le modèle symbolique considéré les stores vérifient $s_{i+i_1}(\mathtt{x}_1) = s_{i+i_2}(\mathtt{x}_2)$ ; or cela signifie aussi que la $(i+1)^{\text{ème}}$ lettre contient la formule $\mathsf{X}^{i_1-1}\mathtt{x}_1 = \mathsf{X}^{i_2-1}\mathtt{x}_2$, en supposant $i_1$ et $i_2$ non nuls, seul cas intéressant ; et donc la condition $(A)$ est vérifiée.

Maintenant, supposons qu'un modèle symbolique $\sigma$ soit accepté par l'automate $\mathbb{A}^*_\Phi$. Montrons par récurrence l'existence d'un modèle concret : nous allons montrer par récurrence sur $i$ la propriété : il existe $s_0 \dots s_{i+m}$ et $h_0, \dots, h_i$ tels que pour tout $u \leq i$, $h_u, s_u \dots s_{u+m}$ est un modèle partiel de la $i^{\text{ème}}$ lettre de $\sigma$. Tout d'abord, l'initialisataion est donnée par la définition des lettres : il existe $h_0, s_0 \dots s_m$ un modèle partiel de la première lettre. Construisons maintenant les stores et tas suivants. Soient $h_0, \dots, h_i$ et $s_0 \dots s_{i+m}$ les stores et tas déjà construits au $i^{\text{ème}}$ stade de la récurrence. Construisons $h_{i+1}$ et $s_{i+m+1}$. Soit $h', s'_0 \dots s'_m$ un modèle partiel de la $(i+1)^{\text{ème}}$ lettre.

Commençons par $s_{i+m+1}$. Par la condition $(A)$, $s_{i+1}, \dots s_{i+m}$ et $s'_0, \dots s'_{m-1}$ ont les mêmes classes d'équivalence pour l'égalité. Soient $A_k$ les classes d'équivalence en question, et $l_k$ l'image des éléments de cette classe. Pour toute variable $\mathtt{x}$, si il existe $(\mathtt{y}, j) \in A_k$ tel que $s'_j(\mathtt{y}) = s'_m(\mathtt{x})$, alors on définit $s_{i+m+1}(\mathtt{x}) = l_k$. Sinon, soient $B_k$ les classes d'équivalence pour l'égalité dans $s'_m$, entre les variables ne rentrant pas dans le cas précédent. Soient $l'_k$ des adresses distinctes n'appartenant à l'image d'aucun de $s_{i+1}, \dots s_{i+m}$, en nombre égal aux classes d'équivalence. Alors on définit pour tout élément $\mathtt{x}$ de $B_k$ : $s_{i+m+1}(\mathtt{x}) = l'_k$. Nous obtenons grâce à ce store les formules d'égalité de la $(i+1)^{\text{ème}}$ lettre.

Maintenant, $h_{i+1}$.[6] Nous ne nous préoccuppons que de $s'_0 (\approx s_{i+1})$, et du tas $h'$.

[6]Le raisonnement est très proche de celui fait pour la démonstration du lemme 1.2.2. Cependant, formaliser le rapprochement des deux preuves n'est pas aisé.

- Tout d'abord, définissons la fonction $tr$ : pour tout $\mathtt{x}$, $tr(s'_0(\mathtt{x})) = s_{i+1}(\mathtt{x})$, et pour $l \notin Im(s'_0)$, $tr(l) = nil$ ; puis la fonction $Tr$ : pour tout $\mathtt{x}_1, \ldots, \mathtt{x}_n$, $Tr(s'_0(x_1), \ldots, s'_0(x_n)) = (tr(x_1), \ldots, tr(x_n))$. Définissons pour toute variable $\mathtt{x}$ telle que $h'$ est défini en $s_0(x)$, $h_{i+1}(tr(\mathtt{x})) = Tr(h'(s'_0(\mathtt{x})))$. Cela permet d'obtenir les formules "`alloc` " et "$\hookrightarrow$" souhaitées de la $(i+1)^{\text{ème}}$ lettre.
- Ensuite, notons $u$ le nombre d'adresses du domaine de définition de $h'$ n'apparaissant pas dans l'image de $s'_0$, c'est-à-dire la taille de "$garb$". Soient $u$ adresses, fraîches relativemant à $s_{i+1}, \ldots, s_{i+m+1}$ ; on définit $h_{i+1}$ comme associant $nil$ à ces adresses. Ce tas n'est défini à aucun autre endroit, ce qui permet de vérifier les formules "`size` $\geq$ $k$".

Le modèle concret construit a pour abstraction le modèle abstrait accepté par l'automate. Ce qui prouve le lemme. $\square$

# 10 Correction de l'automate de contraintes temporelles

Voici la preuve du lemme 3 :

*Soit $(a_i)_{i\in\mathbb{N}}$ un modèle symbolique. Alors ce modèle est reconnu par $\mathbb{A}^s_\Phi$ si et seulement si $(a_i)_{i\in\mathbb{N}} \models_\mu \Phi$.*

Nous allons prouvé les implications du "ssi" l'une après l'autre.

**($\Rightarrow$)** Supposons que $(a_i)_{i\in\mathbb{N}}$ soit reconnu par $\mathbb{A}^s_\Phi$. Soit $(q_i)_{i\in\mathbb{N}}$ un calcul acceptant correspondant. Nous allons montrer par induction sur les formules de $cl(\Phi)$ que, quel que soit $j$, $\phi \in q_j$ ssi $(a_i)_{i\in\mathbb{N}}, j \models_\mu \phi$.

Soit donc $\phi \in cl(\Phi)$, et $j \in \mathbb{N}$.

- Si $\phi$ est une formule d'état, $\phi = \mathcal{A}$, alors par définition de l'automate, $\phi \in q_j$ ssi $a_j \models_{\mathrm{s}} \mathcal{A}$, et donc par définition de la satisfaction symbolique, ssi $(a_i)_{i\in\mathbb{N}}, j \models_\mu \mathcal{A}$.
- Si $\phi = \phi_1 \wedge \phi_2$, alors comme $a_j$ est un atome, $\phi \in a_j$ ssi ($\phi_1 \in a_j$ et $\phi_2 \in a_j$), et par induction, ssi $\{(a_i)_{i\in\mathbb{N}}, j \models_\mu \phi_1$ et $(a_i)_{i\in\mathbb{N}}, j \models_\mu \phi_2\}$, enfin par définition de la statisfaction symbolique, ssi $(a_i)_{i\in\mathbb{N}}, j \models_\mu \phi_1 \wedge \phi_2$.
- Si $\phi = \neg\phi'$, le raisonnement est similaire à celui de la conjonction, cidessus.
- Si $\phi = \mathsf{X}\phi'$, alors par définition des transitions, $\mathsf{X}\phi \in a_j$ ssi $\phi \in a_{j+1}$, et par induction, ssi $(a_i)_{i\in\mathbb{N}}, j+1 \models_\mu \phi'$, et enfin ssi $(a_i)_{i\in\mathbb{N}}, j \models_\mu \mathsf{X}\phi'$
- Si $\phi = \phi_r \mathsf{U} \phi'_r$, selon la numérotation des "until", attardons nous plus sur ce cas.

Supposons $\phi \in q_j$. $(q_i)_{i\in\mathbb{N}}$ est un calcul acceptant, qui passe donc infiniment souvent par les états de $F_r$, donc il existe $j' > j$ tel que $q_{j'} \in F_r$ ; soit alors $j_0$ le plus petit entier supérieur à $j$ tel que $q_{j_0} \in F_r$. Par récurrence sur $e$ de $j$ à $j_0 - 1$, montrons que $\{\mathsf{X}(\phi_r \mathsf{U} \phi'_r), \phi_r\} \subseteq q_e$. Si $j = j_0$, il n'y a rien à montrer ; sinon, $e < j_0$, et donc $q_e \notin F_r$. Pour $e = j$, $\phi_r \mathsf{U} \phi'_r \in q_e$, et comme l'état est un atome, $\{\mathsf{X}(\phi_r \mathsf{U} \phi'_r), \phi_r\} \subseteq q_e$ ou $\phi'_r \in q_e$, mais puisque l'état n'est pas dans

$\phi_r'$, $\phi_r' \notin q_e$, et la propriété est vérifiée. Si c'est vrai pour $e < j_0 - 1$, comme $\mathsf{X}(\phi_r \mathsf{U} \phi_r') \in q_e$, on a $\phi_r \mathsf{U} \phi_r' \in q_{e+1}$, et par le même raisonnement que pour $e = j$, on en déduit que la propriété $\{\mathsf{X}(\phi_r \mathsf{U} \phi_r'), \phi_r\} \subseteq q_e$ est vérifiée. Enfin, la récurrence nous donne : $\mathsf{X}(\phi_r \mathsf{U} \phi_r') \in q_{j_0 - 1}$, d'où $\phi_r \mathsf{U} \phi_r' \in q_{j_0}$, et $\neg(\phi_r \mathsf{U} \phi_r') \notin q_{j_0}$, ce dont on déduit, l'état étant dans $F_r$, que $\phi_r' \in q_{j_0}$. Nous avons donc à la fois, pour $j \leq e < j_0$, $\phi_r \in q_e$ et $\phi_r' \in q_{j_0}$ ; ce dont l'induction nous permet de conclure que pour $j \leq e < j_0$, $(a_i)_{i\in\mathbb{N}}, e \models_\mu \phi_r$ et $(a_i)_{i\in\mathbb{N}}, j_0 \models_\mu \phi_r'$. Et par définition de la satisfaction symbolique, $(a_i)_{i\in\mathbb{N}}, j \models_\mu \phi_r \mathsf{U} \phi_r'$.

Supposons $(a_i)_{i\in\mathbb{N}}, j \models_\mu \phi_r \mathsf{U} \phi_r'$. Alors il existe $j_0$ supérieur à $j$ tel que pour $j \leq e < j_0$, $(a_i)_{i\in\mathbb{N}}, e \models_\mu \phi_r$ et $(a_i)_{i\in\mathbb{N}}, j_0 \models_\mu \phi_r'$. L'induction nous dit que $\phi_r' \in q_{j_0}$, et donc $\phi_r \mathsf{U} \phi_r' \in q_{j_0}$. Une récurrence pour $e$ de $j_0$ à $j$ nous dirait que $\phi_r \mathsf{U} \phi_r' \in q_e$ pour tout $e$ de cet intervalle, et en particulier pour $j$.

De toute cette induction on déduit que $\Phi \in q_0$ ssi $(a_i)_{i\in\mathbb{N}}, 0 \models_\mu \Phi$. Or $q_0$ est le premier état d'un calcul acceptant, c'est donc un état initial. Et donc $\Phi \in q_0$, d'où $(a_i)_{i\in\mathbb{N}}, 0 \models_\mu \Phi$.

**($\Leftarrow$)** Supposons que $(a_i)_{i\in\mathbb{N}}, 0 \models_\mu \Phi$. Nous allons montrer que la suite d'états $(q_i)_{i\in\mathbb{N}}$ définie comme suit est un calcul acceptant :

$$q_j = \{\phi \in cl(\Phi), (a_i)_{i\in\mathbb{N}}, j \models_\mu \phi\}$$

Pour cela nous devons montrer que pour tout $j$, $(q_j, a_j, q_{j+1}) \in \delta$. Ce qui est vrai lorsque :

- pour toute formule d'état $\mathcal{A}$ de $cl(\Phi)$, $\mathcal{A} \in q_j$ ssi $a_j \models \mathcal{A}$. Or cela est assuré par la définition de $q_j$.
- pour toute formule $\mathsf{X}\phi \in cl(\Phi)$, $\mathsf{X}\phi \in q_j$ ssi $\phi \in q_{j+1}$. Or par la définition de la satisfaction symbolique, $(a_i)_{i\in\mathbb{N}}, j \models_\mu \mathsf{X}\phi$ ssi $(a_i)_{i\in\mathbb{N}}, j+1 \models_\mu \phi$. Ce qui se traduit par la propriété souhaitée, par définition de $(q_i)_{i\in\mathbb{N}}$.

Nous devons de plus montrer que l'on passe infiniment souvent par chacun des ensembles d'états finaux $F_k$. Soit donc $k$ quelconque, montrons que pour une infinité d'entiers $j$, $\neg(\phi_k \mathsf{U} \phi_k') \in q_j$ ou $\phi_k' \in q_j$. Cela est vrai si pour tout $i$, il existe $j > i$ tel que $\neg(\phi_k \mathsf{U} \phi_k') \in q_j$ ou $\phi_k' \in q_j$. Soit donc $i_0$. Si $\neg(\phi_k \mathsf{U} \phi_k') \notin q_i$, alors $\phi_k \mathsf{U} \phi_k' \in q_{i_0}$. Montrons qu'il existe $j > i_0$ tel que $\phi_k' \in q_j$, ce qui permettra de conclure. Puisque $\phi_k \mathsf{U} \phi_k' \in q_{i_0}$, on a $(a_i)_{i\in\mathbb{N}}, i_0 \models_\mu \phi_k \mathsf{U} \phi_k'$. Et par définition de la satisfaction symbolique, il existe $j > i_0$ tel que $(a_i)_{i\in\mathbb{N}}, j \models_\mu \phi_k'$[7]. Ce qui implique : $\phi_k' \in q_j$.

Enfin, notons que l'état $q_0$ contient $\Phi$, et est donc un état initial. Ce qui conclut la liste des conditions définissant un calcul acceptant.

# 11 Model-checking symbolique

Nous voulons montrer le lemme 5.1 :

*Soit $a \in \Sigma_\mu$. Il est possible de vérifier en espace polynomial la validité de "$a \models_s \mathcal{A}$", pour $a$ et $\mathcal{A}$ données.*

[7] et pour tout $u$ entre $i_0$ et $j$... mais cette propriété ne nous intéresse pas ici

Pour cela, nous devons d'abord prouver les deux lemmes assurant que l'union disjointe pour modèles symbolique est utilisable dans un algorithme en espace polynomial.

**Test de l'union disjointe symbolique** Tout d'abord le lemme 5.1.1 :

*Il est possible de tester en espace polynomial si $a = a' * a''$.*

**Démonstration:**

Pour tester si $a = a' * a''$, il suffit de vérifier les critères suivants :

- Les égalités entre les variables sont exactement les mêmes pour $a$, $a'$ et $a''$.
- Soient $k$, $k'$ et $k''$ les limites pour $u$ de validité de $\texttt{size} \geq u$ pour respectivement $a$, $a'$ et $a''$. Alors $k = k' + k''$ si $k < w$ (le $w$ de la mesure), et $k' + k'' \geqslant w$, $k' \geqslant w$ ou $k'' \geqslant w$ sinon.
- de toutes les formules $\texttt{alloc}\ x$ de $\mathcal{L}_\mu$, $\texttt{alloc}\ x \in a$ ssi $\texttt{alloc}\ x$ est dans exactement l'un de $a'$ ou $a''$.
- de toutes les formules "$\hookrightarrow$" de $\mathcal{L}_\mu$, l'une est dans $a$ ssi elle est dans exactement l'un de $a'$ ou $a''$.

Cela se fait en espace polynomial (je ne détaille pas).

Il nous reste à prouver la correction des critères.

Supposons que ces critères sont vérifiés par $a$, $a'$ et $a''$. Soit $s_1, \ldots, s_n, h$ un modèle partiel de $a$. Commençons par noter que comme, par le premier critère, les égalités sont exactement les mêmes dans les trois lettres, utiliser $s_1, \ldots, s_n$ dans les modèles partiels de $a'$ et $a''$ est parfaitement cohérent. Il nous faut construire $h^1$ et $h^2$ correspondant respectivement aux lettres $a'$ et $a''$ pour obtenir deux modèles partiels. On définit $h^i(s_u(\texttt{x})) = h(s_u(\texttt{x}))$ ssi $\texttt{alloc x} \in a_i$. Ensuite, soit $v_0$ la limite de validité de $\texttt{size} \geq v$ pour $a'$, et soit $X^1$ un ensemble de $v_0$ adresses de $\texttt{dom}(h)$ n'appartenant pas à $\texttt{Im}(h_{|\texttt{Var}})$, c'est-à-dire non accessibles par une variable. Soit aussi $X^2 = (\texttt{Im}(h) \backslash \texttt{Im}(h_{|\texttt{Var}})) \backslash X^1$, le reste des adresses non accessibles. On définit, pour tout $l \in X^i$, $h^i(l) = h(l)$. Remarquons maintenant que $s_1, \ldots, s_n, h^i$ est un modèle partiel de $a_i$ : comme indiqué précédemment, les égalités vérifiées sont celles souhaitées. De plus, la première partie de la définition de $h^i$ permet de vérifier exactement les formules "$\hookrightarrow$" et "$\texttt{alloc}$ " de $a_i$. Enfin, la seconde partie de cette définition permet de vérifier les formules "$\texttt{size} \geq k$". Nous avons donc bien des modèles partiels des lettres souhaitées.

Il nous faut aussi montrer que $h = h^1 * h^2$. Le troisième critère assure que $\texttt{alloc x} \in a'$ ssi $\texttt{alloc x} \in a$ et $\texttt{alloc x} \notin a''$ (et symétriquement, relativement à $'$ et $''$), ce qui, grâce à la première partie de la définition de $h$, se traduit par $\texttt{x} \in \texttt{dom}(h^i)$ ssi $\texttt{x} \in \texttt{dom}(h^{1-i})$. De cela ,et des choix disjoints d'adresses dans la seconde partie de la définition de $h^i$, on déduit : $h^1 \perp h^2$. Enfin, par définition des $h^i$, $\texttt{dom}(h^1) \cup \texttt{dom}(h^2) = \texttt{dom}(h)$ et pour toutes adresses $l$ et $l'$, $h^i(l) = l'$ implique $h(l) = l'$. On a donc bien $h^1 * h^2 = h$.

Réciproquement, supposons que $a = a' * a''$. La satisfaction des critères pourrait se déduire par une énumération lourde. $\square$

**Séparation symbolique**

Maintenant le lemme :

*Soit une lettre a. Il existe $a'$ et $a''$ tels que $a = a' * a''$ et $a' \models \mathcal{A}'$ et $a'' \models \mathcal{A}''$ ssi $a \models \mathcal{A}' * \mathcal{A}''$.*

**Démonstration:**

Soit $a$ une lettre.

Supposons qu'il existe $a'$ et $a''$ tels que $a = a' * a''$ et $a' \models_{\mathrm{s}} \mathcal{A}'$ et $a'' \models_{\mathrm{s}} \mathcal{A}''$. Montrons que $a \models_{\mathrm{s}} \mathcal{A}' * \mathcal{A}''$. Par définition de $a = a' * a''$, il existe deux modèles partiels $s_0, \ldots, s_n, h'$ et $s_0, \ldots, s_n, h''$ de $a'$ et $a''$ tels que $s_0, \ldots, s_n, h' * h''$ est un modèle partiel de $a$. Puisque $s_0, \ldots, s_n, h'$ est un modèle partiel de $a'$, on a $s_0, \ldots, s_n, h' \models_{\mathrm{part}} \mathcal{A}'$ et $s_0, \ldots, s_n, h'' \models_{\mathrm{part}} \mathcal{A}''$. Ce qui signifie que pour tout remplacement des "_", $((s_0, h'), \ldots, (s_n, \_), (\_, \_), \ldots), 0 \models \mathcal{A}'$ et $((s_0, h''), \ldots, (s_n, \_), (\_, \_), \ldots), 0 \models \mathcal{A}''$. Cela implique, par définition de la satisfaction, que : $((s_0, h' * h''), \ldots, (s_n, \_), (\_, \_), \ldots), 0 \models \mathcal{A}' * \mathcal{A}''$. Et donc $s_0, \ldots, s_n, h' * h'' \models_{\mathrm{part}} \mathcal{A}' * \mathcal{A}''$.

Réciproquement, supposons que $a \models \mathcal{A}' * \mathcal{A}''$. Alors il existe $s_0, \ldots, s_n, h$ un modèle partiel de $a$ tel que $s_0, \ldots, s_n, h'' \models_{\mathrm{part}} \mathcal{A}' * \mathcal{A}''$. Ainsi, il existe $h'$ et $h''$ deux tas tels que $s_0, \ldots, s_n, h' \models_{\mathrm{part}} \mathcal{A}'$ et $s_0, \ldots, s_n, h'' \models_{\mathrm{part}} \mathcal{A}''$ et $h = h' * h''$. Et donc les lettres $a'$ et $a''$ associées aux modèles partiels $s_0, \ldots, s_n, h'$ et $s_0, \ldots, s_n, h''$ satisfont les définitions de $a = a' * a''$, $a' \models \mathcal{A}'$ et $a'' \models \mathcal{A}''$. $\square$

**Le lemme 5.1 :**

Pour démontrer ce lemme,

*Soit $a \in \Sigma_\mu$. Il est possible de vérifier en espace polynomial la validité de "$a \models_{\mathrm{s}} \mathcal{A}$", pour $a$ et $\mathcal{A}$ données.*

**Démonstration:**

Il suffit d'utiliser l'algorithme suivant,

$MC(a, \mathcal{A})$ :

- pour $\mathcal{A} = \mathtt{x} \mapsto \mathtt{x}_1, \ldots \mathtt{x}_n$, l'algorithme accepte ssi $\mathtt{x} \hookrightarrow \mathtt{x}_1, \ldots \mathtt{x}_n \in a$ et $\mathtt{size} \ \geq \ 1 \in a$ et $\mathtt{size} \ \geq \ 2 \notin a$.
- pour $\mathcal{A} = emp$, l'algorithme accepte ssi $\mathtt{size} \ \geq \ 1 \notin a$.
- pour $\mathcal{A} = (\mathtt{x} = \mathtt{x}')$, l'algorithme accepte ssi $\mathtt{x} = \mathtt{x}' \in a$.
- pour $\mathcal{A} = \neg\mathcal{A}'$, l'algorithme accepte ssi $MC(a, \mathcal{A}')$ n'accepte pas.
- pour $\mathcal{A} = \mathcal{A}' \wedge \mathcal{A}''$, l'algorithme accepte ssi $MC(a, \mathcal{A}')$ accepte et $MC(a, \mathcal{A}'')$ accepte.
- pour $\mathcal{A} = \mathcal{A}' * \mathcal{A}''$, l'algorithme énumère tous les $(a', a'')$ tels que $a = a' * a''$, teste pour chacun d'entre eux $MC(a', \mathcal{A}')$ et $MC(a'', \mathcal{A}'')$. Il accepte ssi l'un de ces couples de lettres est tel que les deux appels récursifs sont acceptants.
- pour $\mathcal{A} = \mathcal{A}' \mathrel{-\!\!*} \mathcal{A}''$, l'algorithme énumère tous les $(a', a'')$ tels que $a' = a * a''$, teste pour chacun d'entre eux $MC(a'', \mathcal{A}')$ et $MC(a', \mathcal{A}'')$. Il accepte ssi chacun de ces couples de lettres est tel que les deux appels récursifs sont acceptants.

L'algorithme utilise un espace polynomial, car la pile est de hauteur polynomiale, et même linéaire puisque la longueur de la formule décroît strictement à chaque appel récursif, et la mémoire nécessaire pour chaque appel récursif, c'est-à-dire sans compter ses propres appels qui en résulteront, est polynomiale. En effet, pour les appels sur des formules atomiques, il suffit de tester si une lettre contient un nombre fixé de formules, ce qu'on peut faire en PSPACE par la proposition 4 ; et pour $*$ et $-\!\!*$, chaque paire de lettres peut être stockée en espace polynomial puisque une lettre seule peut l'être, il est possible d'énumérer toutes ces paires en espace polynomial puisqu'elles ont un codage de taille polynomial et puisqu'il est possible de vérifier en espace polynomial si un codage est une lettre par la proposition 3 (dès lors, on énumère les paires de codages et vérifie que ce sont des paires de lettres), et il est possible de vérifier en espace polynomial que la conjonction séparante de deux lettres est ce qu'on souhaite par le lemme 5.1.1.

Montrons que l'algorithme est correct par induction. Pour les cas atomiques, par exemple $\texttt{emp}$ : $a \models_{\mathrm{s}} \texttt{emp}$ ssi il existe un modèle partiel de $a$ satisfaisant $\texttt{emp}$, ce qui est vrai ssi ce modèle partiel ne satisfait pas la formule du fragment classique $\texttt{size} \geq 1$, ce qui a lieu ssi $\texttt{size} \geq 1 \notin a$, finalement ssi $MC(a, \texttt{emp})$ est acceptant. Le raisonnement est similaire pour les autres cas atomiques. Pour les booléens, prenons l'exemple de $\neg\mathcal{A}$ : $a \models_{\mathrm{s}} \neg\mathcal{A}$ ssi non $a \models_{\mathrm{s}} \mathcal{A}$, par induction ssi $MC(a, \mathcal{A})$ n'est pas acceptant, ce qui est vrai ssi $MC(a, \neg\mathcal{A})$ est acceptant. Le cas de la conjonction est similaire. Enfin, pour les opérateurs $*$ et $-\!\!*$, prenons l'exemple de la conjonction séparante : l'algorithme accepte ssi il existe $(a', a'')$ avec $a = a' * a''$ tels que $MC(a', \mathcal{A}')$ et $MC(a'', \mathcal{A}'')$ sont acceptants, par induction vrai ssi (il existe $(a', a'')$ avec $a = a' * a''$ tels que) $a' \models_{\mathrm{s}} \mathcal{A}'$ et $a'' \models_{\mathrm{s}} \mathcal{A}''$, ssi $a \models_{\mathrm{s}} \mathcal{A}' * \mathcal{A}''$ grâce au lemme 11. Le même lemme permet de conclure le raisonnement similaire que l'on pourrait faire pour $-\!\!*$. $\square$