Model-Checking et Abstraction

Laure DANTHONY

juin & juillet 2001


## Résumé

Dans ce contre-rendu du stage MIM1 réalisé du 5 juin au 21 juillet au Laboratoire Spécification et Vérification (LSV) de l'ENS de Cachan, nous présentons les travaux effectués dans le domaine du Model-Checking symbolique. Après une courte mais néanmoins nécessaire introduction de la problématique et des notations, nous abordons le cœur du problème qui consiste à éviter les explosions combinatoires. Pour cela, nous introduisons la notion mathématique d'*abstraction* et montrons comment appliquer cette notion à la résolution de problèmes liés aux gros systèmes.



## Mots-clés

Model Checking symbolique - Logiques temporelles CTL et PLTL - Model Checker - Cadence SMV - State-explosion problem - Abstraction - Problèmes d'ascenseurs.


*Une version de ce rapport ainsi que les transparents de la soutenance sont disponibles à l'adresse :*

`http://www.ens-lyon.fr/~ldanthon/stageMIM1/stageMIM1_fst.html`


LSV, CNRS UMR 8643, ENS de Cachan, 61 av. du Pdt Wilson, 94235 CACHAN Cedex, FRANCE
Tél :+33 1 47 40 24 04, Fax :+33 1 47 40 24 64

**Directeur de stage** : Philippe Schnoebelen, `phs@lsv.ens-cachan.fr`

# Table des matières

# Introduction

De grands échecs retentissants comme la destruction du premier exemplaire d'Ariane 5 en 1996 montrent la nécessité de vérifier nombre de gros systèmes ou logiciels. Plusieurs possibilités complémentaires apparaissent alors :

- le *test*, qui est indispensable et permet de découvrir nombre d'erreurs, mais il ne peut être exhaustif,
- la *démonstration assistée*, qui est en principe capable de démontrer ou d'infirmer toute propriété imaginable, mais qui est souvent difficile à mettre en œuvre,
- le *model-checking*, enfin, qui combine méthode exhaustive et une grande partie automatique.

En quoi consiste le model-checking ? Il s'agit tout d'abord de *modéliser* le système à vérifier par un automate, ou plus généralement par un réseau d'automates communiquants. Ensuite il faut formaliser les propriétés qui sont à vérifier à l'aide par exemple d'une *logique temporelle*, et enfin on utilise un outil informatique, un *model-checker*, qui permet de vérifier que le système satisfait bien les propriétés énoncées.

Dans ce stage, nous avons exploré une partie du vaste domaine de recherche que constitue le Model Checking, le Model Checking symbolique, et particulièrement la notion d'**abstraction** qui, nous le verrons, est une façon de repousser les limites des model-checkers. L'objectif était triple :

- se familiariser avec la problématique et les outils de Model Checking,
- comprendre et formaliser mathématiquement la notion d'abstraction en SMV, un Model Checker symbolique,
- caractériser la méthode (en termne d'ergonomie et de puissance) en codant un exemple de très grande taille.

Dans ce rapport, nous introduisons tout d'abord succintement les notions nécessaires à la compréhension de cette étude. Ensuite nous présenterons la notion d'abstraction, de manière théorique puis son incarnation en SMV, le model-checker utilisé. Nous ferons un point sur les gains de temps et d'espace induits par l'utilisation de cette méthode. Enfin, nous détaillerons la méthodologie utilisée pour cette étude.

Tous les codes pourront être trouvés en annexe.

# 1 Principes et techniques du model-checking

Le but de cette section est de réaliser une présentation succinte du cadre de l´étude, des notations afin de comprendre l'algorithme utilisé par le model-checker SMV. Cette présentation est plutôt informelle. Pour plus de précisions et de définitions mathématiques, suivre les liens . . .

## 1.1 Logiques temporelles

Lorsque l'on essaie d'écrire des propriétés concernant un système, on s'aperçoit vite des lourdeurs de l'écriture mathématique usuelle. Par exemple, si l'on veut formaliser la propriété : tout appel à la machine finira par être satisfaite, il faut écrire :

$$\forall t, \exists t' > t, app(t) \Rightarrow ok(t'),$$

et les propriétés peuvent rapidement devenir illisibles.

Une **logique temporelle** est une logique utilisée dans les énoncés faisant intervenir l'ordonnancement du temps. Sa syntaxe explicite est facilement utilisable et les énoncés sont aisément lisibles en langue "naturelle" car les combinateurs ont un sens explicite particulièrement agréable. Nous présentons ici quelques types de logiques temporelles utilisées dans notre étude.

### 1.1.1 La logique temporelle CTL$^*$

En CTL$^*$ , le temps est un temps discret, comme représenté dans la figure 1. Cela signifie qu'entre $t$ et $t + 1$, il se passe un laps de temps qui est indivisible.

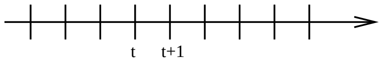



FIG. 1 – Le temps de CTL*

Les formules de la logique CTL* sont des formules booléennes usuelles (connecteurs $\vee$, $\wedge$, $\neg$, $\Rightarrow$, $\Leftrightarrow$) auxquelles on a ajouté des connecteurs temporels ($\mathbf{X}, \mathbf{F}, \mathbf{G}, \mathbf{U}, \ldots$) et des connecteurs de chemin ($\mathbf{A}, \mathbf{E}$), selon la sémantique suivante :
- la sémantique des connecteurs de la logique propositionnelle est celle utilisée habituellement.
- $\mathbf{X}P$ ($\mathbf{X}$ pour *next*) énonce que l´état suivant vérifie $P$,
- $\mathbf{F}P$ ($\mathbf{F}$ pour *future*) énonce qu'un état futur vérifie $P$,
- $\mathbf{G}P$ énonce que tous les états futurs (*de l´exécution courante*) vérifient $P$,
- $P\mathbf{U}Q$ ($\mathbf{U}$ pour *until*) énonce que $Q$ sera vérifiée un jour, et en attendant $P$ est vraie,
- $\mathbf{A}P$ énonce que *toutes les exécutions* partant de l´état courant vérifient $P$,
- $\mathbf{E}P$ énonce qu'*il existe une exécution* partant de l´état courant vérifiant $P$.

EXEMPLES 1.1
- `G (pluie` $\Rightarrow$ `F beau_temps)`,
- `(F forgeron)` $\Rightarrow$ `forger U forgeron`.

*sont des proverbes bien connus !*

REMARQUE 1.1 Les connecteurs ne sont pas tout à fait indépendants, par exemple $\mathbf{G}\varphi \equiv \neg\mathbf{F}\neg\varphi \ldots$

C'est la possibilité de combiner des différents connecteurs qui fait la richesse d'expression de CTL* , par exemple, **GF** permet d'exprimer qu'une propriété est vraie un nombre infini de fois pour une exécution donnée.

### 1.1.2 Satisfaction des formules CTL*

L'objectif étant de vérifier si un système (un automate, un programme, ...) vérifie telle ou telle propriété, nous donnons dans ce paragraphe une définition de la satisfaction d'une formule lors d'une exécution d'un système $\mathcal{A}$ :

DÉFINITION 1.1
On appelle **structure de Kripke** sur $Q$ ou **automate** le quadruplet $\mathcal{A} =< Q, T, q_0, l >$ où :
- $Q$ désigne l'ensemble des états,
- $T \subseteq Q \times Q$ l'ensemble des transitions,
- $q_0$ l´état initial,
- $l$ une fonction d'étiquetage (si $q$ est un état, $l(q)$ désigne l'ensemble des propositions atomiques vérifiées par $q$).

DÉFINITION 1.2
On appelle **exécution** de $\mathcal{A}$ toute suite maximale de transitions de $\mathcal{A}$. Une exécution ne commence pas forcément par l'état $q_0$.

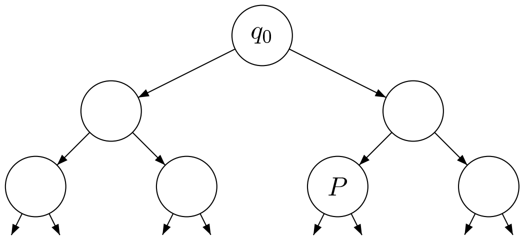
DÉFINITION 1.3
Soit une exécution $\sigma$ de l'automate $\mathcal{A}$ . Alors la satisfaction d'une formule $\varphi$ au temps $i$ de l'exécution $\sigma$ est définie inductivement de la façon suivante :
- $\sigma, i \vDash P$ ssi $P \in l(\sigma(i))$.
- $\sigma, i \vDash \neg\varphi$ ssi il n'est pas vrai que $\sigma, i \vDash \varphi$
- $\sigma, i \vDash \varphi \wedge \psi$ ssi $\sigma, i \vDash \varphi$ et $\sigma, i \vDash \psi$
- $\sigma, i \vDash \mathbf{X}\varphi$ ssi $i \leq |\sigma|$ et $\sigma, i+1 \vDash \varphi$
- $\sigma, i \vDash \mathbf{F}\varphi$ ssi il existe $j$, tel que $i \leq j \leq |\sigma|$ et $\sigma, j \vDash \varphi$.

- $\sigma, i \models \mathbf{G}\varphi$ ssi pour tout $j$, tel que $i \leq j \leq |\sigma|$, on a $\sigma, j \models \varphi$.
- $\sigma, i \models \varphi\mathbf{U}\psi$ ssi il existe $j$, $i \leq j \leq |\sigma|$ tel que $\sigma, j \models \psi$, et pour tout $k$ tel que $i \leq k < j$, $\sigma, k \models \varphi$.
- $\sigma, i \models \mathbf{E}\varphi$ ssi il existe une exécution $\sigma'$ telle que $\sigma(0)\dots\sigma(i) = \sigma'(0)\dots\sigma'(i)$ et $\sigma', i \models \varphi$.
- $\sigma, i \models \mathbf{A}\varphi$ ssi pour toute exécution $\sigma'$ telle que $\sigma(0)\dots\sigma(i) = \sigma'(0)\dots\sigma'(i)$, on ait $\sigma', i \models \varphi$.

On note $\mathcal{A}, \sigma, i \models \varphi$ si **au temps** $i$ **de l'exécution** $\sigma$ de $\mathcal{A}$, la formule $\varphi$ est vérifiée. Enfin, on note $\mathcal{A} \models \varphi$ si $\mathcal{A}, \sigma, 0 \models \varphi$ pour toute exécution $\sigma$ de $\mathcal{A}$.

REMARQUE 1.2 On omet souvent le contexte $\mathcal{A}$, comme dans la définition précédente.

EXEMPLE 1.2 *Si les exécutions de $\mathcal{A}$ sont représentées par l'arbre suivant :*




*alors* $\mathcal{A} \models \mathbf{EF}P$.

REMARQUE 1.3 On désigne par **logique du temps arborescent** une logique qui, comme CTL$^*$ , présente la possibilité de quantifier librement sur les chemins possibles.

### 1.1.3 Les logiques CTL et PLTL

Dans cette section, nous présentons les logiques CTL et PLTL , qui sont des restrictions de la logique CTL$^*$ . Nous verrons que l'utilisation de CTL au lieu de CTL$^*$ présente quelques avantages. En outre, c'est la logique utilisée par le model-checker SMV, qui est notre logiciel de travail.

DÉFINITION 1.4 ( **CTL** )
CTL est le fragment obtenu à partir de CTL$^*$ quand on exige que chaque utilisation d'un combinateur temporel ($\mathbf{X}, \mathbf{F}, \mathbf{U}, \dots$) soit immédiatement sous la portée d'un quantificateur de chemin (**A** ou **E**). On peut donc considérer que les combinateurs dont dispose CTL sont **EX**, **AX**, **E_U_**, **A_U_** et leurs dérivés, comme par exemple **EF**, ...

REMARQUE 1.4 Ce critère est uniquement syntaxique : il peut en effet exister deux formules équivalentes dans CTL$^*$ dont l'une est une formule CTL , et l'autre non.

En CTL , cette obligation de quantification limite sensiblement le pouvoir d'expression. Il est en effet impossible d'emboîter plusieurs combinateurs temporels tout en continuant à faire référence à “une exécution”. Dès lors, les formules CTL sont des *formules d'état* : leur satisfaction ne dépend pas de l'exécution courante, mais uniquement de l'état courant. Cependant, les lacunes de CTL seront vites occultées par le fait que son model-checking est plus efficace, ce que nous allons voir dans la prochaine section.

DÉFINITION 1.5 ( **PLTL** )
PLTL est le fragment obtenu à partir de CTL$^*$ en enlevant les connecteurs de chemin **A** et **E**.

REMARQUE 1.5 Il faut noter qu'il existe des propriétés CTL qui ne sont pas PLTL et vice-versa. L'utilisation d'une logique ou de l'autre est en fait déterminée selon les cas. Par exemple, PLTL est plus pratique dans certains cas, par exemple lorsque l'on souhaite pouvoir vérifier certaines propriétés “à la volée”, mais pas forcément de manière exhaustive. Plus de développements sont donnés dans [SBL$^+$99].

En Cadence SMV, les propriétés peuvent être exprimées en CTL et en PLTL . Dans la majorité des cas, les propriétés sont énoncées en PLTL par l'utilisateur, et elles sont parfois converties lors de la vérification en propriétés CTL .

## 1.2 Algorithmes de base

### 1.2.1 L'algorithme de base pour CTL

Dans cette section nous introduisons dans ses grandes lignes l'algorithme de model-checking pour CTL , qui joue un rôle fondamental dans le domaine de la vérification. En effet, c'est un algorithme performant (il est linéaire en chacune des entrées : l'automate et la formule CTL $\varphi$ à vérifier).

Tout repose sur le fait que les formules CTL sont des formules d'états : cela permet de se limiter à la question : "quels états vérifient la formule $\varphi$ ?" au lieu de raisonner en terme d'exécution.

L'idée est de coder une sous-procédure `marquage` qui, pour chaque état $q$ de $\mathcal{A}$ et pour chaque sous-formule $\psi$ de $\varphi$, va marquer si $q$ satisfait ou non la formule $\psi$. A la fin, pour chaque état et pour chaque sous-formule, `q.psi` vaut `true` si $q \models \psi$, `false` sinon. On utilise évidemment une procédure récursive qui utilise la satisfaction de sous-formules pour étudier la satisfaction de la formule étudiée. Quand ce marquage est achevé, il suffit de vérifier la valeur de `q_0.phi` pour savoir si $\mathcal{A} \models \varphi$ ou non.

Une présentation plus détaillée de cet algorithme est présente dans le chapitre 3 de [SBL$^+$99]. Une analyse simple montre que cet algorithme termine dans le cas où $\mathcal{A}$ est un automate fini, et qu'il est correct. De plus, sa complexité est en $O(|\mathcal{A}| \times |\varphi|)$.

### 1.2.2 Un algorithme pour PLTL

Un algorithme pour PLTL peut être trouvé dans le chapitre 3 de [SBL$^+$99]. Sa complexité est en $O(|\mathcal{A}| \times 2^{|\varphi|})$. Il utilise la synchronisation d'automates. Son exposé n'est pas utile pour la suite de l'étude.

## 1.3 Explosion combinatoire et Model Checking symbolique

### 1.3.1 State explosion problem

Les deux algorithmes présentent l'inconvénient du *state explosion problem* : en effet, ils nécessitent la construction explicite de l'automate $\mathcal{A}$ qui peut être très gros. En fait, le problème n'est pas tant la taille de la structure à vérifier (car les algorithmes sont linéaires en $|\mathcal{A}|$) mais plutôt le fait que l'on utilise des notations assez compliquées pour décrire ces mêmes automates, notations qui sont potentiellement de taille exponentielle en $|\mathcal{A}|$.

On voit dès lors la nécessité de ne pas construire explicitement l'automate, et l'objet de la section suivante sera de montrer quelques méthodes pour résoudre ce problème.

### 1.3.2 Model-checking symbolique

Dans le paragraphe précédent, nous avons vu l'absolue nécessité de ne pas construire entièrement l'automate des états. Le **model-checking symbolique** désigne toute méthode qui ne construit pas explicitement cet automate, mais au contraire cherche à le représenter *symboliquement*, c'est-à-dire par exemple code de manière concise un très grand nombre d'états à la fois, et cherche à les manipuler par "paquets".

Dans le chapitre 4 de [SBL$^+$99], on peut trouver un codage symbolique des ensembles de variables et des transitions très satisfaisante en taille. Ce sont les BDDs (*binary decision diagram*). Les opérations boolénnes (et, ou, il existe) s'y implémentent de façon efficace, c'est-à-dire de manière quadratique (*pour une seule opération*). Cette représentation est celle choisie par les concepteurs de SMV.

## 1.4 Cadence SMV : présentation du model-checker

Cadence SMV est un **model-checker**, c'est-à-dire un outil informatique qui permet de faire de la vérification de formules sur des sytèmes automatiquement. Dans cette section, nous allons présenter succintement les possibilités de SMV, en les illustrant par un exemple simple. Plus tard, nous évaluerons les compétences du logiciel sur des exemples de très grande taille.

### 1.4.1 Caractéristiques du model-checker

Cadence SMV est un model-checker symbolique pour CTL et PLTL sur des réseaux d'automates avec variables partagées qui utilise les BDDs pour la vérification. C'est le premier outil de model-checking qui a permis de traiter des exemples de très grande taille : par exemple il a permis la vérification d'un circuit électronique à $10^{1300}$ états.

CMU SMV a été développé par K. L. McMillan sous la direction de E. M. Clarke à l'université de Carnegie-Mellon (Pittsburgh, USA). Une version plus récente, Cadence SMV, a été developpée ensuite par la société Cadence. C'est cette dernière version qui a été utilisée dans cette étude. Cadence SMV, logiciel auquel nous ferons référence par la suite uniquement par "SMV" est disponible sur le Web à l'adresse `http://www-cad.eecs.berkeley.edu/~kenmcmill/smv.html`, où l'on trouve aussi les manuels utilisateur [McM99b] et [McM99a].

Une utilisation très simple de SMV peut se résumer par les trois étapes suivantes :
- dans un premier temps, on code l'automate à vérifier, c'est à dire que l'on déclare ses variables d'états et comment elles évoluent au cours du temps,
- dans un second temps, on énonce les formules que l'on désire vérifier selon la syntaxe CTL ou PLTL ,
- dans un troisième temps, on laisse faire la machine : on lance VW, l'interface graphique de SMV, on demande au logiciel de vérifier telle ou telle propriété, après calcul, la propriété est validée avec `true`, ou invalidée avec `false` *et un contre-exemple.*

La majorité des personnes qui utilisent SMV le font selon la progression précédente, car les autres possibilités du logiciel ne sont pas bien détaillées dans la documentation. La limitation de cette utilisation vient de l'explosion du nombre d'états : on peut toujours trouver un système assez gros (ou bien augmenter la taille du système en cours de traitement) pour saturer la mémoire de SMV. Ceci dit, dans nombre de cas, la puissance des ordinateurs actuels combinée avec l'utilisations d'outils algorithmiques performants (par exemple les BDDs) dans le code du logiciel, permettent une vérification efficace.

### 1.4.2 Un exemple simple de vérification

Dans cet section, nous allons vérifier un ascenseur simple : lorsque l'utilisateur appuie sur un bouton d'appel (à un des étages), la lumière d'appel s'allume et la cabine se déplace vers cet étage .

- Les variables utilisées sont `cabine` (localisation de la cabine), `dir` (la direction de la cabine), `b_appel` (un bouton d'appel pour chaque étage, prend la valeur 1 si il est appuyé, la valeur 0 sinon), `l_appel` (une lumière d'appel pour chaque étage).
- Le code de marche est assez clair : si la cabine est à l'arrêt et qu'on l'apelle, alors elle prend la direction de l'appel, *etc.*
- Les propriétés à vérifier sont du type : "si on appuie sur un bouton, la cabine finira bien par arriver",*etc.*

On trouvera le code de cet ascenseur, en langage SMV dans l'annexe `ex1.smv`.

*Et la vérification ?*

Lorsque l'on demande "Verify-All", SMV met environ deux secondes à répondre que toutes les propriétés énoncées (16 en tout) sont vraies pour le codage de l'ascenseur [1].

[1]Bien noter qu'il y a 288 configurations possibles des variables.

# 2 Une première idée de l'abstraction

## 2.1 Introduction

Pour vérifier un modèle de système à $n$ variables d'états, la construction est exponentielle en ce nombre de variables. On voit donc directement la nécessité de réduire le nombre de variables.

Deux idées simples peuvent permettre dans certains cas de résoudre le problème :

- Une première idée est d'utiliser les éventuelles symétries du problème. Par exemple, si on dispose d'un tableau de 30 valeurs booléennes dont il s'agit de vérifier la consistence, et qui ont toutes un comportement identique, il suffit de n'en tester qu'une pour savoir le comportement de toutes. SMV offre cette possibilité. Pour cela, il suffit de déclarer les cases du tableau à l'aide de la commande `scalarset` (voir le paragraphe 4.4 de [McM99a]). Tous les éléments déclarés avec ce type ont alors le même comportement (si bien sûr ils sont dans le même environnement).
- Une deuxième idée est d'utiliser le raisonnement par induction. Cela est très intéressant pour montrer une propriété sur un grand nombre d'entiers. SMV offre aussi cette possibilité qui est bien détaillée dans le paragraphe 4.7 de [McM99a].

Mais dans certains cas ces idées simples ne suffisent pas et il faut faire appel à des méthodes plus complexes, les méthodes d'abstraction.

L'abstraction est une méthode simplificatrice. Son esprit est de regrouper certaines variables afin de vérifier leur comportement en une seule fois. Dans les faits, le système abstrait est "plus gros" que le système initial, dans le sens où il présente plus de comportements possibles, mais il est plus simple à vérifier car ces comportements sont moins complexes.

Nous allons dans cette section donner des idées informelles de ce qu'est l'abstraction et donner un premier exemple de la validation d'une abstraction en SMV.

Sans plus de détails mathématiques, on suposera dans cette section $M_1 \sqsubseteq M_2$, ce qui désigne que $M_2$ est une **abstraction** de $M_1$ ou que $M_1$ est un **raffinement** de $M_2$. De manière informelle, cela signifie que $M_2$ a été obtenu à partir de $M_1$ en augmentant le nombre de ses comportements.

## 2.2 Buts de la méthode, généralités

Lorsqu'un système ($M_1$) est trop complexe pour pouvoir être vérifié par une approche directe, on désire pouvoir vérifier des propriétés sur un système plus simple, tout en garantissant que la satisfaction d'une formule sur ce système ($M_2$) implique la satisfaction de la formule sur le système initial($M_1$). Le travail de SMV va alors être divisé en deux parties :

- la vérification que le deuxième système est bien une abstraction du premier, ce qui garantira le transfert des propriétés,
- la vérification de ces propriétés sur le système abstrait.

Nous verrons plus tard que le transfert des propriétés est garanti par l'utilisation de la logique PLTL . Moralement, $M_2 \vDash \varphi \in$ PLTL signifie $\forall \sigma, M_2, \sigma \vDash \varphi$, et donc comme l'ensemble des chemins de $M_1$ est inclus dans l'ensemble des chemins de $M_2$, on obtient $M_1 \vDash \varphi$.
Nous verrons aussi que pour avoir la même propriété de transfert dans le cas de la logique CTL, il faut se restreindre aux formules de sûreté, de manière intuitive on peut justifier ceci par l'argument suivant : l'augmentation du nombre de comportements a pour effet une diminution du nombre de propriétés de sûreté vérifiées.

## 2.3 L'abstraction en SMV

Ici nous montrons comment coder les parties relatives à $M_1$, $M_2$, et comment utiliser SMV pour vérifier la justesse de l'abstraction.

- Dans un premier temps, on code le système $M_1$ (déclaration des variables, leur initialisation et comportements) en SMV. Nous appellerons comme K. McMillan cette portion de code **l'implémentation**.

- Une abstraction du système $M_1$, contenant les mêmes variables d'état, peut ensuite être codée à l'intérieur d'un bloc délimité par des crochets et désigné par le mot clef `layer`. A l'intérieur de ce bloc, des instructions du type initialisation et description de transitions *de ces mêmes variables d'état* sont autorisées pour décrire le comportement de cette abstraction $M_2$.
- La vérification de la justesse (**validation**) de l'abstraction peut ensuite être effectuée à l'aide de VW. Le logiciel détermine à l'ouverture du fichier quelles sont les variables dont le comportement a été redéfini dans le layer. Par exemple `x//toto` signifie que la variable `x` est redéfinie dans le layer nommé `toto`, et qu'il faut vérifier sa compabilité avec son implémentation.

## 2.4 Sur l'exemple asc-essai.smv

Les fichiers `asc-essai-a.smv`, `asc-essai-b.smv` et `asc-essai-c.smv` sont un exemple d'utilisation de l'abstraction en SMV, sur un système simple.

### 2.4.1 Le code de $M_1$ et de $M_2$

La première partie du code de `asc-essai-a.smv` réalise l'implémentation d'un ascenseur élémentaire dont la cabine oscille entre les étages 0 et MAX, éteignant les voyants lumineux allumés lors du passage à l'étage concerné [2]. Rien ne dirige l'évolution des variables `boutons` (les utilisateurs de l'ascenseur font ce qu'ils veulent !), et les `lumieres` s'allument (*i.e.* se mettent en position 1) lorsque leur bouton correspondant a été appuyé (position 1 aussi). L'évolution de la variable `cabine` est régie par l'évolution de la variable `direction`. Enfin, les boutons et les lumières sont initialisés à 0, et la direction initiale est soit haut, soit bas.

Le code contenu dans le layer nommé `M2` est une abstraction de l'implémentation dans la mesure où plus de comportements sont possibles. Plus rien ne régit en effet le comportement de la variable `direction`, la variable `cabine` a un comportement trivial dans la mesure où à tout moment, elle possède n'importe quelle valeur entre 0 et MAX, la seule contrainte est que la lumière de l'étage où se trouve la cabine est éteinte. Le nouvel automate obtenu contient plus de transitions que le précédent, puisque sa description autorise des transitions auparavant interdites.

Passons maintenant à la validation de l'abstraction par le model-checker VW. A l'ouverture du fichier `asc-essai-a.smv`, VW nous dit que les variables `lumieres[0]`, `lumieres[1]`, `lumieres[2]` , `lumieres[3]` ont été redéfinies dans `M2`. La commande `Prop - Verify All` permet de vérifier toutes ces définitions à la fois. Et voici le résultat :

```
Model checking results
======================
lumieres[0]//M2.........................................................true
lumieres[1]//M2.........................................................true
lumieres[2]//M2.........................................................true
lumieres[3]//M2.........................................................true
```

L'abstraction $M_2$ de notre ascenseur $M_1$ donne donc bien une définition des variables `lumieres[0]`, `lumieres[1]`, `lumieres[2]`, `lumieres[3]` compatible avec l'implémentation . Il est donc maintenant possible de vérifier des propriétés (attention cependant à leur type, voir à ce sujet une des sections précédentes) concernant ces variables selon la définition qu'en donne le code de $M_2$.

### 2.4.2 Vérification de propriétés élémentaires

Dans le fichier `asc-essai-b.smv`, nous avons rajouté des propriétés PLTL qu'il s'agit de vérifier. Puisque nous disposons d'une abstraction, nous allons tenter de les vérifier dans le cadre de cette abstraction. Sélectionnons donc la propriété `ok1[2]` qui énonce que si la cabine est au deuxième étage, alors le bouton du deuxième étage n'est pas allumé. Vérifions que cette propriété est vraie en selectionnant `Prop - Verify {ok1[2]}`, sans surprise, on obtient `true`.

[2] C'est une façon de modéliser le fait que les requêtes ont bien été satisfaites

Mais qu'avons-nous au juste vérifié ? En fait, par défaut, VW vérifie une formule selon la définition des variables la plus abstraite possible. Le "raisonnement" utilisé pour définir quel layer utiliser pour la définition d'une variable peut être visualisé en sélectionnant `Abstraction - Explain layer` . Dans le cas de la propriété `ok1[2]`, si on sélectionne la variable `lumiere[2]`, on obtient ainsi :

```
Choosing a layer for lumieres[2]:
No layer is specified for lumieres[2].
Thus, I will look for the highest layer that assigns lumieres[2].
The layers assigning lumieres[2] are: , M2 and the implementation.
The highest usable layer is layer M2.
```

Ici donc VW a utilisé la définition `lumiere[2]//M2` pour vérifier la propriété `ok1[2]`.

Il faut remarquer que l'on peut vérifier `ok1[2]` selon la définition `lumiere[2]//M2` *sans avoir auparavant validé l'abstraction*. Les deux phases : validation de l'abstraction et vérification des propriétés sont donc totalement indépendantes.

Une dernière remarque pour souligner un défaut de SMV : il semble impossible de forcer la vérification d'une propriété dans le cadre des définitions de l'implémentation, si un layer a changé les définitions des variables . Nous verrons plus loin une solution "empirique" à ce problème.

## 2.5 Abstractions successives

Il est possible en SMV de définir plusieurs layers de plus en plus abstraits. L'utilisation du mot-clef `refines` permet alors de spécifier à SMV quel layer contient l'abstraction la plus poussée. Ainsi, dans le fichier `asc-essai-c.smv`, la phrase `M2a refines M2b ;` permet de dire que le layer `M2b` est une abstraction plus poussée de $M_1$ que `M2a`.

Le comportement de VW est alors le suivant :

- il essaie de valider les deux abstractions de $M_1$ indépendamment.
- la vérification des propriétés s'effectue alors à l'aide des définitions des variables *données par le layer le plus abstrait possible*, pour chaque variable. Si le logiciel ne sait pas quel layer choisir, il utilise l'implémentation.

Si l'on veut forcer l'utilisation de tel layer pour la définition de telle variable, on utilise le mot-clef `using`. Par exemple, à l'intérieur du fichier `asc-essai-c.smv`, la phrase `using lumieres//M2a prove ok1[3];` permet de forcer la définition des `lumiere[i]` selon le layer `M2a` lors de la vérification de la propriété `ok1[3]`.

Pour forcer la vérification d'une propriété dans le cadre des définitions de l'implémentation, il suffit alors de copier le code de l'implémentation à l'intérieur d'un layer `impl` et de vérifier les propriétés à l'aide de `var//impl`.

# 3 Bases théoriques pour la vérification par abstraction

Dans ce paragraphe, nous donnons une définition plus formelle de la méthode d'abstraction dans le cas géneral et donnons des idées de son intérêt. Nous montrons que la définition de l'abstraction selon SMV est un peu différente et donnerons une idée de la méthode de validation.

## 3.1 Une définition formelle de l'abstration

La définition donnée ici s'inspire largement de [CGL94].

Considérons un système de transitions : $M = (D, I, R)$ , où :
- $D$ est l'ensemble des états, c'est-à-dire, si $x_1, x_2, \ldots, x_m$ désignent les variables du système, de domaines respectifs $D_1, D_2, \ldots, D_m$, alors $D = D_1 \times D_2 \times \ldots \times D_m$,
- $I \subseteq D$ est l'ensemble des états initiaux de $M$,
- $R \subseteq D \times D$ l'ensemble des relations (ou transitions) entre les états.

On se donne alors une famille de surjections $h_i)_{1 \leq i \leq n}$ :

$$\begin{array}{rccc} h_i : & D & \rightarrow & \widehat{D_i} \\ & (d_1, \ldots d_m) & \mapsto & \widehat{d_i} \end{array}$$

qui induit la fonction suivante :

$$\begin{array}{rccc} h : & D & \rightarrow & \widehat{D} = \prod_{i=1}^{n} \widehat{D_i} \\ & (d_1, \ldots d_m) & \mapsto & \big(h_1(d_1, \ldots, d_m) \ldots, h_n(d_1, \ldots, d_m)\big) \end{array}$$

DÉFINITION 3.1
Soit $\widehat{M} = (\widehat{D}, \widehat{I}, \widehat{R})$ un système de transitions sur $\widehat{D}$, on dit que $\widehat{M}$ est **une abstraction** de $M$ par $h$, ce que l'on note $M \sqsubseteq_h \widehat{M}$ si :

1. $\forall d \in D,\ d \in I \Rightarrow \widehat{d} \in \widehat{I}$ (où $\widehat{d} = h(d)$),
2. $\forall d_1, d_2 \in D,\ d_1 R d_2 \Rightarrow \widehat{d_1} \widehat{R}\ \widehat{d_2}$ (où $\widehat{d_1} = h(d_1)$ et $\widehat{d_2} = h(d_2)$).

En termes clairs, cela signifie que le comportement de $\widehat{d}$ (initialisations puis transitions) "représente" le comportement de son(ses) antécédent(s) par la fonction $h$. En effet, si $d_{init}$ est un état initial des variables, alors $\widehat{d_{init}}$ est un état initial dans l'abstraction ; si $d_1$ et $d_2$ sont des états des variables reliés par une transition, alors $\widehat{d_1}$ et $\widehat{d_2}$ sont aussi reliés dans l'abstraction.

DÉFINITION 3.2
La donnée de $h$ et de $\widehat{M}$ sera par abus de langage appelé **abstraction**.

L'intérêt est que dans ce processus on a pu faire décroître le nombre des variables, en regroupant des variables dont le comportement est (presque-) similaire, ou bien dont le détail du comportement est inutile pour la compréhension (et la vérification) du système.

Nous verrons plus tard comment une abstraction peut être réalisée en SMV.

## 3.2 De l'intérêt de l'abstraction

La question majeure est donc : à quoi sert une abstraction ? En fait, le processus d'abstraction garantit la transmission des propriétés des propriétés PLTL et des propriétés CTL de sûreté (une propriété de sûreté est une propriété qui garantit que, sous certaines conditions, une certaine chose ne se produit jamais) :

- Une démontration de la transmission des propriétés de l'abstraction vers le raffinement dans le cas de la logique $\forall$ CTL* (c'est-à-dire CTL* sans le quantificateur de chemin $\exists$) peut être trouvée dans [CGL94]. Ce théorème a directement pour corollaire la transmission des propriétés PLTL puisque celles-ci n'utilisent aucun quantificateur de chemin.
- En revanche, seules les propriétés de sûreté pour CTL sont de manière certaine transmises de l'abstraction vers l'implémentation car il y a alors quantification pour tous les chemins. Mais les propriétés comme l'absence de blocage ne pourront être prouvées sur le modèle abstrait directement.

Remarque 3.1 La non-satisfaction d'une propriété par le modèle abstrait n'apporte rien à la connaissance de l'implémentation. Le théorème cité plus haut ne comporte en effet qu'une implication.

## 3.3 L'abstraction selon SMV

Le model-checker SMV n'utilise pas cette définition de l'abstraction. A l'aide de tests, nous avons pu nous apercevoir de deux modifications majeures :
- Les domaines $D$ et $\widehat{D}$ sont les mêmes, et les fonctions $h_i$ sont les fonctions identité.
- La vérification de l'abstraction ne s'effectue que sur les états atteignables.

Si la deuxième restriction est une modication sérieuse de la définition, la première peut se voir comme une facilité d'écriture qu'ont choisie les concepteurs de SMV. En effet, utiliser la fonction identité est plus commode pour le code, comme nous le verrons par la suite, et de plus la validation de l'abstraction se fait plus facilement aussi. De plus, on peut toujours se ramener au cas $h = \mathrm{Id}$ par le petit jeu d'écriture suivant :

$$\begin{array}{ccc}
(x_1, \dots x_m) \in D & \xrightarrow{(h_i)_{1\leq i \leq n}} & (y_1, \dots, y_n) \in \widehat{D} \\
\Big\downarrow {\scriptstyle extension} & & \Big\uparrow {\scriptstyle projection} \\
(x_1, \dots x_m, y_1, \dots, y_n) & \xrightarrow[\mathrm{Id}]{} & (x_1, \dots x_m, y_1, \dots, y_n)
\end{array}$$

En effet, on peut se ramener au cas $h = Id$ en ajoutant aux variables de départ (les $x_i$) les variables d'arrivée (les $y_i$) selon leurs définitions données par les fonctions $h_i$. De même on ajoute aux variables d'arrivée les variables de départ, à l'aide de leur définition initiale. Valider une abstraction $\widehat{M}$ d'ensemble de variables $\widehat{D}$ revient alors à valider cette même abstraction sur l'ensemble $D \times \widehat{D}$ (on a juste rajouté des variables "inutiles").

En conséquence, la définition de l'abstraction de SMV, si elle est un peu différente, n'offre pas de problème majeur car la sémantique raisonne le plus souvent en terme d'états atteignables [3].

*Et concrétement, comment réalise-t-on une abstraction en SMV ?*

En SMV, c'est l'utilisateur qui réalise l'abstraction :

- Toutes les variables sont définies dans la partie de code appelée **implémentation** . On définit aussi leur comportement, c'est-à-dire que l'on code les ensembles $I$ et $R$.
- A l'intérieur du bout de code nommé **layer**, on définit les ensembles $\widehat{I}$ et $\widehat{R}$ de l'abstraction en fonction des variables de l'abstraction.

Comment réalise-t-on le codage des ensembles $I$, $R$, $\widehat{I}$ et $\widehat{R}$ en SMV ? On utilise les fonctions `init` et `next`, dont la syntaxe est peu intuitive. Le paragraphe suivant donne une interprétation de ces fonctions en termes de formules.

## 3.4 Une formulation en termes logiques

Pour plus de commodités lors de la manipulation des variables de l'abstraction, nous formalisons ici l'idée d'abstraction (selon SMV) en termes de formules du premier ordre. Nous verrons que cette formulation se

[3] Se restreindre aux états atteignables revient en fait à corriger la définition $D = \prod D_i$ : on prend alors $M = (Q, I, R)$ où $Q =_{def} R^*(I)$ (alors $Q \subseteq D$).

prête aussi mieux à la validation de l'abstraction et montrons que la vérification variable par variable est licite, et présente certains avantages pour le logiciel.

### 3.4.1 Correspondance SMV/théorie

Le tableau suivant résume la sémantique des fonctions SMV `init` et `next` en termes mathématiques, puis en terme de formules boolénnes :

| SMV | MATH | FORMULE |
|---|---|---|
| `init(x):=f(x1,...,xn)` | $x(t=0) = f(x_1, \ldots, x_n)$ | $\varphi^x_{init}(x_1, \ldots, x_n)$ |
| `next(x):=g(x1,...,xn,x'1, ... x'n)` | $x(t+1) = g(x_1(t), \ldots, x_n(t), x_1(t+1), \ldots, x_n(t+1))$ | $\psi^x_R(x_1, \ldots, x_n, x'_1, \ldots, x'_n)$ |

REMARQUE 3.2 Rappelons que les définitions circulaires sont interdites.

L'implémentation comme l'abstraction se codent à l'aide des `init` et `next` selon la sémantique donnée, alors :

$$I =_{def} \{(a_1, \ldots, a_n) | a_1, \ldots a_n \vDash \bigwedge_{i \in [\![1,n]\!]} \varphi^{x_i}_{init}(a_1, \ldots, a_n)\}$$

et :

$$R =_{def} \{(a_1, \ldots, a_n, a'_1, \ldots, a'_n) | a_1, \ldots a_n, a'_1, \ldots, a'_n \vDash \bigwedge_{i \in [\![1,n]\!]} \psi^{x_i}_R(a_1, \ldots, a_n, a'_1, \ldots, a'_n)\}.$$

La définition de $\widehat{I}$ et $\widehat{R}$ est similaire.

REMARQUES 3.3 (SUR LES DÉFINITIONS IMPLICITES)

- Lorsque l'on ne code pas une définition d'une variable $x$ dans l'implémentation : alors SMV considère $\varphi^x_{init} \equiv true$ et/ou $\psi^x_R \equiv true$.
- Lorsque l'on ne précise pas l'initialisation/les transitions d'une certaine variable dans l'abstraction, SMV considère que la variable a le même comportement que dans l'implémentation en ce qui concerne son initialisation/ses transitions.

### 3.4.2 Validation de l'abstraction

Suivant la formalisation précédente, vérifier la SMV-abstraction $\widehat{M}$ consiste à vérifier les deux assertions suivantes :

1. $I \subseteq \widehat{I}$
2. $R \cap \{(\overrightarrow{a}, R(\overrightarrow{a'})), \overrightarrow{a} \text{ atteignable}\} \subseteq \widehat{R}$.

REMARQUE 3.4 On pourrait se demander pourquoi on ne vérifie pas plutôt à la place de la deuxième propriété : $R \subseteq \widehat{R}$. L'exemple du paragraphe suivant nous montrera qu'une SMV-abstraction ne vérifie pas forcément cette implication. Plus tard nous verrons aussi comment nous nous sommes aperçu de cette restriction.

En fait, en SMV on vérifie les variables séparément. Vérifier $x_i // \widehat{M}$ signifie vérifier :

$(1_i)$ $\forall a_i \in I_{|x_i}, \bigwedge_{j \in [\![1,n]\!]} \varphi^{x_j}_{init}(x_1, \ldots, a_i, \ldots x_n) \Rightarrow \varphi^{x_i}_{\widehat{init}}(x_1, \ldots, a_i, \ldots x_n)$.

$(2_i)$ $\forall a_i \in R^*_{|x_i}(x^0_i), \forall a'_i \in R_{|x_i}(a_i), \bigwedge_{j \in [\![1,n]\!]} \psi^{x_j}_R(x_1, \ldots, a_i, \ldots, x_n, x'_1, \ldots, a'_1, \ldots, x'_n)$
$\Rightarrow \psi^{x_i}_{\widehat{R}}(x_1, \ldots, a_i, \ldots, x_n, x'_1, \ldots, a'_1, \ldots, x'_n)$, où $x^0_i$ désigne l'ensemble des valeurs initiales de la variable $x_i$.

REMARQUE 3.5 Les $a_i$ sont fixés, les $x_i$ sont des variables libres ...Nous garderons cette convention dans la suite.

Nous allons montrer qu'il est licite de vérifier les variables séparément, grâce à la proposition suivante :

PROPOSITION 3.1 $\forall i, x_i // \widehat{M} \Longleftrightarrow I \subseteq \widehat{I}$ *et* $R \cap \{(\overrightarrow{a}, R(\overrightarrow{a'})), \overrightarrow{a}$ *atteignable*$\} \subseteq \widehat{R}$.

PREUVE :

- Montrons que $\forall i, (1_i) \Leftrightarrow I \subseteq \widehat{I}$ : on a

$$\Big(\forall i, (1_i)\Big) \Leftrightarrow \Big(\forall \overrightarrow{a} \in I, \bigwedge_{j \in [\![1,n]\!]} \varphi^{x_j}_{init}(\overrightarrow{a}) \Rightarrow \varphi^{x_i}_{\widehat{init}}(\overrightarrow{a})\Big)$$

  de façon évidente.

- Montrons que $\forall i, (2_i) \Leftrightarrow R \cap \{(\overrightarrow{a}, R(\overrightarrow{a})), \overrightarrow{a}$ atteignable$\} \subseteq \widehat{R}$ : on a

$$(2_i) \Leftrightarrow \forall a_i \in R^*(I)_{|x_i}, \forall a'_i \in R_{|x_i}(a_i), \bigwedge_{j \in [\![1,n]\!]} \psi^{x_j}_R(x_1, \ldots, a_i, \ldots, x_n, x'_1, \ldots, a'_1, \ldots, x'_n)$$

$$\Rightarrow \psi^{x_i}_{\widehat{R}}(x_1, \ldots, a_i, \ldots, x_n, x'_1, \ldots, a'_1, \ldots, x'_n),$$

  ou encore, comme $R^*(I)_{|x_i} = R^*_{|x_i}(I_{|x_i})$ :

$$(2_i) \Leftrightarrow \forall \overrightarrow{a} \in R^*(I), \forall \overrightarrow{a'} \in R(\overrightarrow{a}), \bigwedge_{j \in [\![1,n]\!]} \psi^{x_j}_R(\overrightarrow{a}, \overrightarrow{a'}) \Rightarrow \psi_{\widehat{R}}(\overrightarrow{a}, \overrightarrow{a'}).$$

  Finalement, $\Big(\forall i, (2_i)\Big) \Leftrightarrow \Big(R \cap \{(\overrightarrow{a}, R(\overrightarrow{a})), \overrightarrow{a}$ atteignable$\} \subseteq \widehat{R}\Big)$.

■

REMARQUE 3.6 Il est très intéressant de vérifier les variables séparément, notamment pour situer une éventuelle erreur faite lors de l'abstraction.

**Important !** Une SMV-abstraction est une abstraction qui *se restreint aux états atteignables.*

### 3.5 Un exemple simple

Considérons le code suivant :

```
-----------------------------------------------------------------------------------------
module main()
{
/* declaration des variables*/

   x,y,z : boolean ;
   z := x & y ;

/* implementation : M */

   init(x) := 0 ;
   init(y) := 0 ;

   next(x) := !y ;
   next(y) := !x ;

/*abstraction*/
```

```
   layer M_chapeau :
   {
      init(z) := 0 ;
      next(z) := !z ;
   }
}
-------------------------------------------------------------------------------------
```

Si on ne considère que les états atteignables, on voit que l'on a codé les automates des figures 2 et 3.

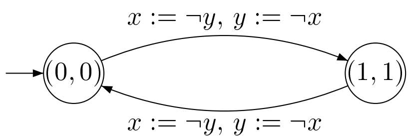



FIG. 2 – L'automate M...

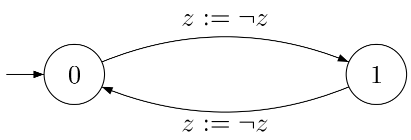



FIG. 3 – ... et son abstraction $\widehat{M}$

Il s'agit ici de vérifier que $\widehat{M}$ réalise bien une abstraction de $M$. Pour celà, il faut vérifier que :

1. $(x_0 = 0) \wedge (y_0 = 0) \wedge (z_0 = x_0 \wedge y_0) \Rightarrow (z_0 = 0)$
2. $(x' = \neg y) \wedge (y' = \neg x) \wedge (z' = x' \wedge y') \Rightarrow (z' = \neg z)$

... sur les automates restreints aux états atteignables!

Et en effet, avec la commande `Prop - Verify z//M_chapeau`, VW répond :

```
Model checking results
======================
z//M_chapeau..............................................................true
```

REMARQUE 3.7 On voit bien que l'on ne vérifie que les états atteignables car $(x = 0, y = 0, z = 1, x' = 1, y' = 1, z' = 1)$ est un contre-modèle de la deuxième implication (mais l´état $(x = 0, y = 0, z = 1)$ n'est pas atteignable).

Les autres variables de $\widehat{M}$, $x$ et $y$, étant par ailleurs trivialement validées, l'abstraction $\widehat{M}$ de $M$ est donc finalement valide. On peut donc vérifier sans problème des propriétés dans le système $\widehat{M}$.

## 3.6 Un exemple plus conséquent

### 3.6.1 But de l'étude, présentation du système

On décide d'étudier ici le gain offert par l'utilisation de l'abstraction en terme de temps et d'espace. Nous codons dans un fichier `time1.smv` [4] une version d'ascenseur à deux cabines dont le comportement assure la venue d'une cabine, essaie de minimiser le déplacement des cabines et aussi le temps de travail de l'ascenseur.

[4]Le nom des fichiers n'est pas très judicieusement choisi, voir les commentaires qui suivent :-)

Dans un fichier `time2.smv` nous ajoutons le layer `spec_cab1` qui est une abstraction du comportement de la cabine numéro 1.

Le but de l'exemple [5] est de voir si il est possible en pratique de trouver des abstractions qui à la fois :
- font gagner du temps et de l'espace (*i.e.* sont assez simplificatrice),
- ne rendent pas les propositions faussent (*i.e.* ne sont pas trop simplificatrices).

### 3.6.2 Vérification des propriétés, avec ou sans abstraction

Remarquons en premier lieu que le coût de la validation de l'abstraction n'est pas nul, par exemple la vérification de `pos1//spec_cab1` prend $0,27$s, de même que la validation de chaque `l_appel[i]//spec_cab1`.

Comparons ensuite les temps de vérification des propriétés (et le nombre d'itérations effectuées), avec et sans abstraction, dans *des cas favorables*, c'est-à-dire lorsque SMV renvoie `true`, garantissant ainsi dans le deuxième cas comme dans le premier la satisfaction de ces mêmes propriétés par l'ascenseur :

| Prop | `time1.smv` | `time2.smv` |
|---|---|---|
| `dir_ok1` | 0.27 s / 14 it. | 0,26 s / 7 it. |
| `hyp_stop` | 18,27 s / 637 it. | 3,09 s / 342 it. |
| `ok1[2]` | 1,49 s / 10 it. | 1,41 s / 6 it. |

et en terme de nœuds de BDD alloués (idée de complexité spatiale) :

| Prop | `time1.smv` | `time2.smv` |
|---|---|---|
| `dir_ok1` | 17740 | 4113 |
| `hyp_stop` | 239887 | 38347 |
| `ok1[2]` | 10036 | 4052 |

Ainsi l'abstraction permet bien un gain de temps non négligeable lors de la vérification des propriétés. Dans la section suivante, nous verrons qu'elle permet de vérifier des propriétés dans des cas où sans abstraction la machine ne peut réaliser cette vérification, faute de place.

## 3.7 Un exemple encore plus conséquent

Le code de cet ascenseur a été inspiré du code d'une *Lift-Machine* de manière abstraite (cf [Abr96] pages 358 à 369). Pour les indications de complexité spatiale et temporelle, on utilisera 2 cabines et 4 étages.

### 3.7.1 Présentation du code

On code dans le fichier `big_lift.smv` un système de `NB_LIFT` ascenseurs dans un immeuble qui compte `FLOOR = TOP - GROUND` 1+ étages. Les mécanismes internes de l'ascenseur sont supposés fiables et on ne s'y intéresse pas. Les spécifications requises sont les suivantes :
- Chaque cabine (`LIFT`) possède autant de boutons qu'il y a d'étages. Lorsque l'on appuie dessus, ils demandent le déplacement de la cabine vers l'étage désiré.
- Chaque étage (sauf `GROUND` et `TOP`) possède deux boutons : `UP` et `DOWN` qui permettent d'appeler une cabine de cet étage pour aller dans direction indiquée (`UP` ou `DOWN`).
- Quand une cabine n'a plus de requête à satisfaire, elle reste immobile à l'étage de sa destination finale (propriété désirée).
- Toutes les requêtes finiront par être satisfaites, que ce soient des requêtes d'appel de cabine ou des requêtes de déplacement vers un autre étage (propriété désirée).

[5] et du suivant !

Nous choisissons de coder le fonctionnement de cet ascenseur de la manière suivante :
- Les tableaux `moving`, `floors`, `dirs` contiennent les informations sur les cabines : état, étage, directions courants.
- Les tableaux `attracted_up`, `attracted_dn`, `can_continue_up`, `can_continue_dn` permettent de déterminer le comportement des différentes cabines (arrêt, changement de direction ou status-quo).
- Les tableaux `tab_in` et `tab_out` contiennent les requêtes utilisateur. Par exemple, `tab_in[i][up]=1` signifie qu'un utilisateur a demandé une cabine à partir de l'étage $i$ pour aller vers le haut. De manière similaire, `tab_out[i][j]=1` signifie qu'un utilisateur à l'intérieur de la cabine $i$ veut se diriger vers l'étage $j$.

REMARQUE 3.8 Nous avons volontairement choisi de ne pas coder de variables de type `bouton_allume` ou `bouton_appuye`, le codage sous forme matricielle des requêtes nous semblant plus approprié.

Détaillons maintenant le comportement des variables :

- **Evolution "forcée" par le système : invariants :** Le système impose un certain nombre d'invariants qu'il est commode de supposer acquis par la commande `assume` : ainsi il faut imposer le fait que l'étage `GROUND` ne contienne pas de bouton vers le bas, donc qu'une requête de type $(GROUND, dn)$ est impossible. De même, il est impossible, à l'intérieur d'une cabine arrêtée à un certain étage, de demander l'évolution vers cet étage, *etc.* Cette partie du code est commentée par `/* assumptions */`.

- **Evolution des variables booléennes du système selon les requêtes :** Ces variables sont très utiles pour déterminer le déplacement ultérieur d'un cabine donnée :
  - variables `attracted_up` (une par cabine) : de manière informelle, une cabine $i$ est "attirée vers le haut" (`attracted_up[i]=1`) si elle vérifie au moins une des deux conditions suivantes : des passagers de la cabine désirent atteindre un étage situé au dessus de la cabine, ou bien il existe des passagers dans des étages au dessus ;
  - variables `can_continue_up` (une par cabine) : une cabine peut continuer son chemin vers le haut si elle est attirée vers le haut, personne ne veut sortir de la cabine à l'endroit où elle se trouve, et personne ne demande une cabine dans cette direction à partir de l'endroit où elle se trouve.

  Le comportement de `attracted_dn` et `can_continue_dn` est similaire. Cette partie du code est commentée par : `/* boolean tools */`.

- **Codage des mouvements des cabines :** Pour une cabine donnée, il s'agit de coder les cas où ...
  - "elle continue son chemin vers le haut" : dans le cas où la cabine était en train de se déplacer vers le haut, et qu'elle peut continuer, elle monte d'un étage ;
  - "elle s'arrête alors qu'elle se déplaçait vers le haut" : dans le cas où la cabine était en train de se déplacer vers le haut, et qu'elle ne peut pas continuer, elle s'arrête, et les requêtes $(cabine, floors[cabine])$ et $(floors[cabine][dirs[cabine]])$ sont effacées ;
  - "elle (re-)démarre en s'élançant vers le haut" : si la cabine était arrêtée dans la direction "haut", et qu'elle est attirée vers le haut, elle se remet en marche et monte d'un étage ;
  - "elle change de direction du haut vers le bas" : si la cabine était arrêtée dans la direction "haut", et qu'elle n'est pas attirée vers le haut, mais attirée vers le bas, elle prend la direction "bas"et la requête $(floors[cabine], dn)$ est effacée ;

  ... et aussi les mouvements symétriques concernant la direction "haut" ! Cette partie du code est commentée par : `/* lift movements */`.

### 3.7.2 Les propriétés à vérifier et l'Abstraction

Nous décidons de vérifier les propriétés suivantes :
- Toute requête provenant d'un étage finira par être satisfaite.
- Toute requête provenant d'une cabine finira par être satisfaite.
- Si il n'y a plus de requêtes, toutes les cabines s'arrêtent.

L'abstraction choisie provient d'une constatation toute simple ; il n'y a pas besoin de prendre en compte les requêtes provenant d'étages lorsque l'on veut vérifier des requêtes provenant de cabines. Dans l'abstraction `req_int` nous autorisons donc tous les comportements pour la variable `tab_out`. Chaque vérification de `tab_out[i][j]//req_int` requiert $4,5$s, mais cette perte de temps est vite occultée parle gain en temps et en espace induit par cette abstraction : en effet, la vérification de la propriété `reqf_ok` prend avec abstraction 28s et 1148381 BDD-nœuds contre 107s et 3679341 BBD-nœuds.

Remarque 3.9 L'idée initiale était de remplacer le code correspondant au changement de direction en autorisant uniquement les transitions réalisant la décroissance d'une certaine distance, trouvée dans [Abr96]. Nous avons abandonné cette possibilité d'abstraction qui au demeurant est très intéressante pour des questions de difficulté de codage. En effet les propriétés de croissance font intervenir `x` et `next(x)` en même temps, et la syntaxe de SMV semble refuser nombre de formulations différentes de ces propriétés.

Ainsi on voit bien que loin de n'être qu'une pure idée théorique, la notion d'abstraction de systèmes est effectivement réalisable en SMV et permet de repousser les fontières du *state explosion problem*. Plus qu'un gain de temps, elle permet de gagner un ordre de grandeur dans l'évaluation des possibilités de vérification.

# 4 Sur la méthodologie utilisée

Dans cette section nous décrivons la méthodologie utilisée pour arriver à une connaissance assez large du logiciel. Nous verrons que nous avons adopté une méthode “expérimentale”, c'est-à-dire une démarche utilisant un jeu de tests assez étendus pour essayer de dégager les caractéristiques de SMV.

## 4.1 Prise en main de SMV

SMV est un outil universitaire, donc peu documenté et sans support technique. Ne sont disponibles que deux manuels de référence (qui sont une liste d'exemple, sans guère d'explications générales), et aussi une mailing-list, mais qui n'est pas trop utilisée. Les manuels de référence nous ont été utiles pour l'apprentissage du langage, naturellement, mais en ce qui concerne les possibilités les plus subtiles de SMV, les nombreux exemples, sans aucune formalisation, nous ont posé de nombreuses difficultés de compréhension. Il nous a donc fallu élaborer une batterie de tests pour comprendre le comportement de SMV, comme lorsque l'on dispose d'une boîte-noire dont on doit comprendre le comportement sans connaître ce qu'il y a à l'intérieur. Signalons aussi que le code source de SMV n'est pas disponible.

Cette difficulté de prise en main, notamment en ce qui concerne l'utilisation de l'abstraction, explique certainement le fait que les plus grosses utilisations de SMV aient été réalisées par l'équipe de développement de ce logiciel. Par exemple, on ne trouve dans aucune littérature des exemples d'utilisation de l'abstraction en SMV pour l'étude de gros systèmes qui auraient été réalisés par des équipes étrangères au développement du logiciel.

Signalons aussi les points suivants qui nous ont plus ou moins dérangés dans cette étude, soit parce qu'ils n'apparaissaient pas dans la documentation, soit parce qu'ils nous ont posé des problèmes de compréhension :

- l'absence d'un emacs-mode performant rend la production de code plus longue et fastidieuse,
- la syntaxe `init(x):=0` et surtout `next(x):=!y` est trompeuse car, nous l'avons vu, il ne s'agit aucunement d'affectations, mais d'équivalences logiques,
- il semble ne pas avoir de type `int`. Il faut mettre une borne pour déclarer des variables dans un intervalle de $\mathbb{N}$ (`nb := 0..100`) ; ceci peut être justifié par le fait que l'on ne code que des comportements à branchements finis,
- SMV alloue au démarrage toute la place nécessaire au codage des variables en binaire : par exemple, si une variable $x$ peut prendre 3 valeurs différentes, il alloue 2 cases-mémoires pour $x$, même si une valeur peut ne jamais être atteinte ; cela pose des problèmes lorsque l'on veut tester des comportements sur des grandes plages de valeurs (VW se bloque alors au démarrage),
- `z:=x&y` est un raccourci pour `init(z):=init(x)&init(y)` et `next(z):=next(x)&next(y)` ;
- la définition des automates est assez fastidieuse, et dans certains cas la définition utilisée dans la version précédente de SMV, CMU SMV, est préférable. On pouvait par exemple définir les états initiaux par des formules booléennes sur les variables, en faisant : `INIT (x = !y)`. Cette syntaxe évide des définitions par cas, donc des phrases syntaxiquement plus difficiles à comprendre. Les définitions des transitions pouvaient être définies de manière similaire. Des définitions de ce type sont toujours tolérées en Cadence SMV, mais elles sont obsolètes et donc déconseillées.
- Lorsque l'on lance VW, il calcule de quelle(s) variable(s) dépendent les définitions des variables de l'implémentation et de l'abstraction, et aussi de quelles variables dépendent les formules à vérifier, afin de pouvoir restreindre au maximum le champ des vérifications.

## 4.2 Codage de l'abstraction en SMV

Malgré l'imposante part de la partie “Abstraction” de la documentation de SMV, la découverte de la syntaxe associée a été là-encore très fastidieuse. En effet, cette partie n'offre qu'un nombre impressionnant d'exemples, en pointant du doigt certains gains en terme de temps de calcul, mais elle ne formalise pas la notion d'abstraction utilisée, ni n'indique clairement quelle est la sémantique des différentes parties du code. Nous avons dû les découvrir à l'aide de tests.

Il faut tout de même signaler que malgré l'absence de documentation sur VW, cet outil graphique offre tout de même quelques renseignements sur le comportement de SMV. En particulier, la commande `Abstraction - Explain layer` qui permet de savoir quelle définition a été utilisée pour une variable donnée nous a aussi servi à déterminer quel est le comportement de SMV dans le cas des abstractions multiples.

## 4.3 Définition de la SMV-abstraction

Les définitions de la section 3.3 ont été obtenues en testant le comportement de SMV sur des exemples, et en confrontant les résultats avec la théorie de [CGL94] :

- Nous nous sommes très vite aperçu de l'identité des ensembles de variables de l'implémentation et de l'abstraction. VW refusait en effet la définition de nouvelles variables à l'intérieur de l'abstraction.

- La restriction de l'abstraction aux états atteignables nous a en revanche posé quelques problèmes. Il semblait en effet plus naturel de ne pas calculer les états atteignables et de vérifier l'abstraction sur tous les états possibles (comme dans la définition de [CGL94]). De plus un tel calcul est difficile et peut être très long. Pour vérifier notre hypothèse, nous avons codé l'exemple `mod3.smv` qui réalise l'incrémentation d'une variable `nb` de 3 unités, jusqu'à atteindre une valeur `BORNE` paramétrable, puis qui redescend jusqu'à 0 en décrémentant de 3, *etc.* Il s'agit du code suivant :

```
---------------------------------------------------------------------------------
#define BORNE 10

module main()
{
       nb : 0..BORNE ;
       up : boolean ;

       init(nb) := 0 ;
       init(up) := 1;

       case{
         (up & nb<=BORNE-3) : next(nb) := nb+3 ;
         (up & nb>BORNE-3) : {next(nb) := nb-3 ; next(up):=0;}
         (!up & nb>=3) : next(nb) := nb-3 ;
         (!up & nb<3) : {next(nb) := nb+3 ; next(up) := 1;}
       }

       layer M2 : {
          init(nb):= 0..BORNE ;

           case{
              (nb = 7) : next(nb):= 11 ;
              1 : next(nb):= 0..BORNE ;
           }
        }
}
----------------------------------------------------------------------------------
```

L'abstraction permet quant à elle tous les comportements de la variable `nb`, sauf les transitions $7 \to i$ avec $i \neq 11$. Cet exemple va permettre de confirmer ou d'infirmer notre hypothèse selon laquelle seules les transitions atteignables sont vérifiées. En effet, comme la transition $7 \to 11$ n'est pas du type $n \to n \pm 3$, l'abstraction devrait être refusée. Mais dans le cas où l'on ne regarde que les états et les transitions atteignables, comme l'état $nb = 7$ n'est jamais atteint lors de l'exécution, l'abstraction doit être validée.

Regardons donc le comportement de SMV sur ce fichier :

```
Model checking results
======================
nb//M2.............................................................true
...
number of states reached..............................................7
```

Cela montre bien que SMV parcourt tous les états atteignables de l'implémentation pour vérifier l'abstraction. Les 7 états parcourus sont en effet conforme à l'exécution :

$$(0, up) \to (3, up) \to (6, up) \to (9, up) \to (6, \neg up) \to (3, \neg up) \to (0, \neg up) \to (3, up),$$

REMARQUE 4.1 Cette exécution suffit à parcourir tous les états atteignables car elle est cyclique.

- Enfin, nous nous sommes convaincus de la possibilité de valider une abstraction variable par variable en démontrant nous-mêmes la proposition 3.1 (paragraphe 3.4.2).

## 4.4 Limites de SMV

De par le choix des BDD pour la conception du logiciel, il apparaît une limite à son utilisation pour la vérification de programmes. En effet, si le choix des BDDs (et surtout le codage des variables en binaire) s'avère judicieux lorsqu'il s'agit de vérifier des automates, des systèmes de transition, la vérification de programmes de type arithmétique est plus complexe. Le coût des multiplications et des additions est en effet très grand en SMV (on sait par ailleurs qu 'il n'existe pas de "petits"BDD pour réaliser la multiplication par exemple). Il faudrait donc utiliser une technique d'abstraction à chaque apparition d'une multiplication dans un programme, ce qui paraît difficilement réalisable. De plus, le fait qu'un programme utilise de nombreuses boucles se révèle très coûteux lors de sa vérification en SMV, et aussi le codage de structures plus complexes, telles matrices ou graphes.

# 5 Sur le stage

Ce stage s'est très bien déroulé. Les personnes du laboratoire (LSV) ont été très attentives à nos besoins, tant matériels que bibliographiques. Un bureau et des machines performantes ont été mis à notre disposition, et de nombreuses discussions au "coin café" ont permi d'accumuler des connaissances sur la vie du chercheur.

Je remercie Philippe Schnoebelen, mon directeur de stage, qui m'a encadrée dans toutes les phases du stage : bibliographie, recherche, rédaction. De nombreuses mises au point (confrontation de points de vue, rapports de tests, élaboration de nouvelles hypothèses) m'ont permis d'avancer de manière régulière dans mes travaux.

Enfin, la conférence suivie lors de ma dernière semaine de stage, CAV'01 (Computer Aided Verification), dont le programme peut être trouvé sur la page : `http://www.lsv.ens-cachan.fr/cav01/`, m'a permis d'élargir mes connaissances et aussi de rencontrer K. McMillan et E.M. Clarke.

# Conclusion et perspectives

Le Model Checking est un domaine de recherche très vivant, notamment parce que ses domaines d'applications sont très variés et qu'ils intéressent l'industrie. Les algorithmes, les programmes, les sytèmes étudiés sont de plus en plus des exemples de la vie réelle, aux comportements complexes et aux nombre d'états/de comportements élevé.

Si les machines actuelles sont assez performantes pour la résolution de problèmes simples, "de type universitaire", le Model Checking Symbolique, et tout particulièrement les méthodes d'abstractions, offre de nouveaux outils pour repousser le plus possible les limites du *state explosion problem*. Ces méthodes sont de plus en plus employées pour la vérification de systèmes et de protocoles industriels.

Parmi les nombreux points de vue non cités dans cette étude, mentionnons :
- des méthodes de vérification probabilistes et aussi une notion d'abstraction probabiliste (voir [LLM+01]),
- des méthodes de raffinement qui permettent de vérifier les formules CTL autres que les formules de sûreté (voir [CIY95]),
- et plein d'autres choses encore ...

# Références

[Abr96] J.-R. Abrial. *The B-Book : assigning programs to meanings.* Cambridge University Press, 1996.

[CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5) :1512–1542, September 1994.

[CIY95] Rance Cleaveland, Purush Iyer, and Daniel Yankelevich. Optimality in abstractions of model checking. In *Static Analysis Symposium, LNCS 983*, pages 51–63, 1995.

[LLM+01] S. Laplante, R. Lassaigne, F. Magniez, S. Peyronnet, and M. de Rougemont. *Probabilistic abstraction for Model Checking, an approach based on property testing.* preprint, 2001.

[McM99a] K. L. McMillan. *Getting started with SMV.* Cadence Berkerley labs, 1999.

[McM99b] K. L. McMillan. *The SMV Language.* Cadence Berkerley labs, 1999.

[SBL+99] P. Schnoebelen, B. Bérard, F. Laroussinie, M. Bidoit, and A. Petit. *Vérification de logiciels, techniques et outils du model-checking.* Vuibert, 1999.