Automates finis pour la fiabilité logicielle et l'analyse d'accessibilité

Pierre-Cyrille Heam

▶ **To cite this version:**

Pierre-Cyrille Heam. Automates finis pour la fiabilité logicielle et l'analyse d'accessibilité. Informatique [cs]. Université de Franche-Comté, 2009. tel-00432301

HAL Id: tel-00432301
https://theses.hal.science/tel-00432301

Submitted on 16 Nov 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Automates finis pour la fiabilité logicielle et l'analyse d'accessibilité

Pierre-Cyrille Héam

Mémoire d'habilitation à diriger des recherches

Laboratoire d'Informatique de l'Université de Franche-Comté
Equipe-projet INIRIA-CASSIS

Université de Franche-Comté

1er septembre 2009

Table des matières

1 Introduction **5**
- 1.1 Validation et vérification logicielle . . . 6
- 1.2 Propriétés et spécifications . . . 6
- 1.3 Exemple jouet de modélisation par un système états-transitions 7
- 1.4 Systèmes infinis, explosion combinatoire . . . 9
- 1.5 Analyse régulière d'accessibilité . . . 9
- 1.6 Model-checking Régulier . . . 11
- 1.7 Plan . . . 14
- 1.8 Remarques sur les notations . . . 16

2 Approximations régulières de langages d'arbre **19**
- 2.1 Analyse d'accessibilité régulière pour les langages d'arbres . . . 20
- 2.2 Automatiser la complétion . . . 31
- 2.3 Gérer les systèmes non linéaires à gauche . . . 35
- 2.4 Raffinement d'approximations . . . 40
- 2.5 Vérifier des propriétés temporelles . . . 44
- 2.6 Conclusion et perspectives . . . 50

3 Accélération de relations de semi-commutation **53**
- 3.1 Les relations de semi-commutation . . . 54
- 3.2 Calcul du R-mélange . . . 57
- 3.3 PolCom et autres classes . . . 61
- 3.4 Relations de semi-commutation sur les arbres . . . 65
- 3.5 Conclusion et perspectives . . . 67

4 Vérification de systèmes composés **69**
- 4.1 Substitutivité et composition . . . 70
- 4.2 Substitutivité qualitative et quantitative . . . 71
- 4.3 Composition avec contraintes . . . 75
- 4.4 Idéaux de mélange . . . 79
- 4.5 Conclusion et perspectives . . . 79

5 Test aléatoire **81**
5.1 Tester les systèmes 82
5.2 Utilisation d'une génération aléatoire d'automates pour le test . 84
5.3 Génération aléatoire d'automates d'arbre 90
5.4 Génération aléatoire équiprobable de structures récursives : outil Seed 93
5.5 Conclusion et Perspectives 96

6 Autres travaux sur l'analyse d'accessibilité **99**
6.1 Calculs de noyaux abéliens 100
6.2 Comparaison d'automates max-plus 102

7 Bilan personnel **109**

Références personnelles **112**

Bibliographie **115**

Chapitre 1

Introduction

Sommaire

1.1	**Validation et vérification logicielle**	**6**
1.2	**Propriétés et spécifications**	**6**
1.3	**Exemple jouet de modélisation par un système états-transitions**	**7**
1.4	**Systèmes infinis, explosion combinatoire**	**9**
1.5	**Analyse régulière d'accessibilité**	**9**
1.6	**Model-checking Régulier**	**11**
1.7	**Plan**	**14**
1.8	**Remarques sur les notations**	**16**

1.1 Validation et vérification logicielle

L'informatisation de la société est actuellement indéniable, quelque soit le domaine socio-économique. Dans ce cadre, la fiabilité des logiciels devient un facteur critique : toute panne, tout bogue, tout dysfonctionnement, perturbe la bonne marche de nos activités avec parfois des conséquences humaines, financières ou morales importantes.

Contrairement à d'autres domaines industriels plus anciens, la démarche qualité en informatique n'en est encore qu'à ses débuts. Il est couramment convenu comme normal qu'un ordinateur perde un fichier, que l'installation d'un nouveau logiciel en rende un autre inutilisable ou que certaines options d'impression refusent de fonctionner. Si cela peut s'expliquer par la multitude d'équipements existants ainsi que leur évolution très rapide, nous ne tolérerions pas autant de dysfonctionnements sur d'autres appareils.

Cependant, dans des domaines précis (banque, systèmes embarqués, médecine, etc.), dits critiques, on ne peut se permettre d'avoir un système défaillant : une phase importante du développement logiciel est celle de la validation durant laquelle le système est analysé afin de garantir qu'il respecte certaines exigences de fiabilité. Cette phase de validation fait appel à deux approches complémentaires, le test et la vérification :

- Le **test** consiste à faire passer au système informatique un ensemble d'épreuves afin de voir comment il se comporte. Cette phase de validation, antérieurement effectuée à la main selon l'expérience des testeurs, a fait l'objet de très nombreux travaux récents afin d'en améliorer l'efficacité, la qualité et l'automatisation.
- La phase de **vérification** consiste à prouver qu'un système vérifie bien certaines spécifications. Cette phase fournit une preuve mathématique du bon fonctionnement d'un modèle du système. Cependant, de nombreux problèmes de vérification sont indécidables : la vérification se fait donc soit avec l'aide d'un expert (on parle en général de preuve de programme), ou de façon automatique sur des modèles abstraits (on parle alors de model-checking). La preuve est très coûteuse en temps et en ressources humaines et le model-checking est limité dans ses applications.

Les méthodes de test et de vérification sont complémentaires et doivent toutes deux être effectuées pour les systèmes critiques.

1.2 Propriétés et spécifications

Les travaux présentés dans ce document se placent très majoritairement dans le cadre de la vérification par model-checking. Les systèmes que l'on doit analyser sont modélisés mathématiquement par des objets abstraites (automates, mots, termes, formules logiques, etc.), et l'on cherche à prouver au-

tomatiquement des propriétés de ces modèles. Ces propriétés se classent en différents types (selon les auteurs, la classification peut varier ; celle présentée ici est issue de [SBB+99]) :

- Les propriétés **d'atteignabilité** modélisent qu'une situation donnée peut être atteinte par le système. Pour une machine à café, un exemple de propriété d'atteignabilité est : "*Il existe un moyen pour que la machine serve du café*".
- Les propriétés de **vivacité** modélisent que, sous certaines conditions, une situation arrivera. Par exemple, une propriété de vivacité pour une machine à café est : "*s'il reste du café et que l'utilisateur a mis le montant adéquat, la machine finira par lui servir un café*".
- Les propriétés de **sûreté** modélisent que, sous certaines conditions, une situation n'arrivera jamais. Dans le cadre d'une machine à café, "*si le montant adéquat n'a pas été mis, la machine ne servira jamais de café*" est une propriété de sûreté.
- Il en existe d'autres, comme l'**équité**, l'**absence de blocage**, etc.

On peut définir des propriétés beaucoup plus complexes ; cependant la vérification des propriétés ci-dessus est déjà difficile théoriquement et algorithmiquement.

1.3 Exemple jouet de modélisation par un système états-transitions

Nous allons illustrer l'approche par model-checking sur l'exemple du jeu de morpion. On dispose d'une grille de trois cases sur trois. Il y a deux joueurs, A et B. Les règles sont les suivantes :

- Chaque case de la grille soit est vide, soit contient un jeton A, soit contient un jeton B.
- Initialement la grille est vide.
- À chaque partie, le joueur n'ayant pas commencé la partie précédente commence. À la première partie, le joueur A commence.
- Les joueurs jouent à tour de rôle, A en posant un jeton A et B un jeton B.
- Lors de son tour, plutôt que jouer, un joueur peut décider d'abandonner. Il a alors perdu la partie et on en recommence une nouvelle.
- Si trois jetons A sont alignés, A est déclaré vainqueur. La partie s'arrête et on en recommence une nouvelle.
- Si trois jetons B sont alignés, B est déclaré vainqueur. La partie s'arrête et on en recommence une nouvelle.
- Si toutes les cases sont pleines et qu'il n'y a aucun vainqueur, la partie est déclarée nulle et on en recommence une nouvelle.

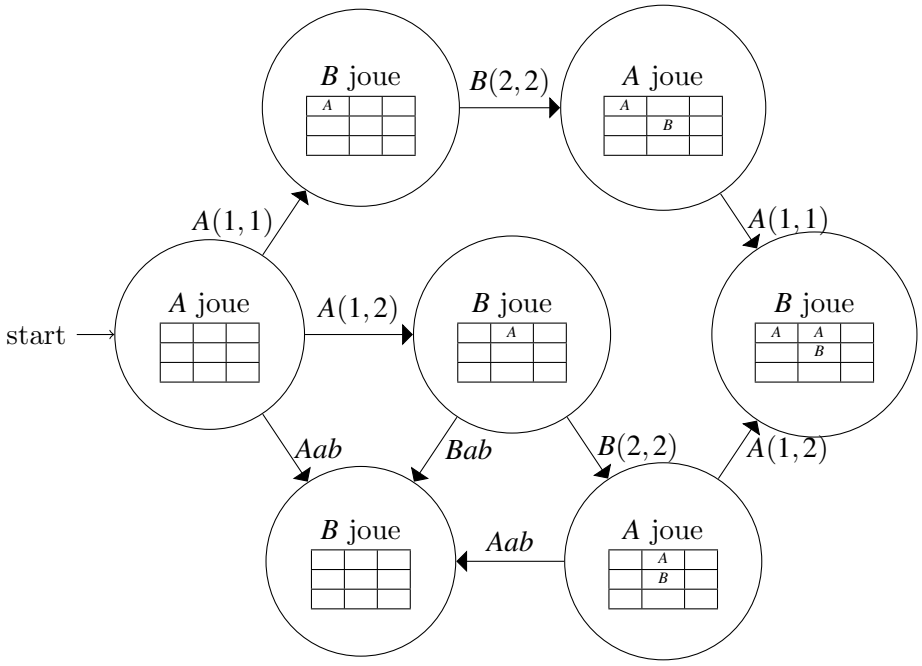


FIG. 1.1 – Jeu de morpion (vue partielle)

L'état du système est défini par la configuration courante de la grille et par le nom du joueur qui doit jouer. On peut définir les actions suivantes :

- $A(i,j)$ pour "*le joueur A joue en* i,j".
- $B(i,j)$ pour "*le joueur B joue en* i,j".
- `A`ab pour *"le joueur A abandonne"*.
- `B`ab pour *"le joueur B abandonne"*.
- `reset` pour recommencer la partie.

Les états du jeu de morpion pourraient s'écrire sous la forme d'un graphe que nous ne représentons que partiellement sur la figure 1.1.

La propriété "*A peut gagner au morpion*" est une propriété d'atteignabilité : on vérifie dans le graphe qu'à partir de la configuration initiale, on peut atteindre un état où *A* a gagné. Le fait que "*B ne jouera jamais deux fois de suite dans une même partie*" est une propriété de sûreté. La propriété "*La différence entre le nombre de pions de A et de B n'est jamais supérieure à 1*" est aussi une propriété de sûreté. La propriété "*Si les joueurs jouent, alors la partie finira*" est une propriété de vivacité.

On pourrait aussi imaginer des propriétés plus complexes, du type "*Modulo la parité des parties, A n'a pas d'avantage sur B*".

1.4 Systèmes infinis, explosion combinatoire

Après l'exemple du morpion, on peut penser que le model-checking ne présente que peu de difficultés : on code tout dans des variables, on construit un graphe et on analyse exhaustivement le système en parcourant ce graphe.

En pratique cela s'avère impossible car les systèmes contiennent trop d'états (et même parfois en contiennent un nombre infini). On ne peut donc pas tout explorer. L'exemple du morpion est assez significatif. Il s'agit d'un système très simple, mais le graphe contient potentiellement $2.3^9 = 39366$ états différents. Bien sûr, certains ne sont pas valides. Si l'on élimine les grilles non accessibles, c'est-à-dire ne pouvant pas apparaître au cours d'une partie (les grilles contenant 3 jetons A et aucun jeton B par exemple), ce nombre d'états diminue sensiblement. On peut cependant montrer qu'il y en a plusieurs centaines. Il est donc possible, sur le morpion, de faire une recherche exhaustive. Cependant, cela est impossible pour des jeux comme les échecs ou le go. Face à cette explosion combinatoire, les techniques vont viser à limiter l'espace de recherche soit en quotientant le graphe à l'aide de propriétés du système (par exemple, sur le morpion, on pourrait raisonner sur les configurations modulo les symétries de la grille, ce qui réduirait le nombre d'états), soit par des preuves mathématiques (et automatisables).

Afin d'apprécier les ordres de grandeurs, quelques *grands* nombres sont répertoriés dans le tableau 1.1. La première partie du tableau montre une indication du nombre d'instructions processeurs exécutables dans un temps donné. Ces nombres sont à mettre en correspondance avec ceux de la deuxième partie du tableau qui donne des ordres de grandeur des graphes (en nombre d'états), de différents systèmes. A titre indicatifs, quelques nombres astronomiques sont indiqués dans la table[1]. En bref il est important de noter que des vérifications naïves s'appuyant sur une construction exhaustive des modèles est hors de portée des calculateurs.

1.5 Analyse régulière d'accessibilité

Étant donné une description formelle d'un système, l'analyse d'accessibilité a pour objectif de vérifier des propriétés de sûreté ou d'atteignabilité. On peut informellement décrire l'analyse d'accessibilité par la résolution du problème suivant (illustré dans la figure 1.2).

Accessibilité
Entrée : Un ensemble de configurations initiales I, une relation d'évolution du système R, un ensemble de configurations B.
Question : Le système décrit par I et R peut-il atteindre un état de B ?

[1] Ces chiffres sont tirés de `http://pagesperso-orange.fr/yoda.guillaume`.

Nombre de cycles de 10000 processeurs à 5GHz qui tournent pendant une semaine.	10^{20}
Nombre de cycles d'un processeur à 5GHz qui tourne pendant un an.	10^{16}
Nombre de cycles d'un processeur à 5GHz, qui tourne depuis le bigbang.	10^{27}
Nombre de cycles d'un milliard de processeurs à 5GHz, qui tournent depuis le bigbang.	10^{36}
Nombre de configurations d'un programme de 10 lignes (points de programme) manipulant trois variables entières (entre 0 et 32000).	10^{16}
Nombre de configurations d'un programme de 100 lignes (points de programme) manipulant 10 variables entières (entre 0 et 32000).	10^{57}
Nombre de configurations de la mémoire d'un ordinateur avec 1MO de Ram et aucun disque dur.	10^{301030}
Nombre d'atomes dans l'univers.	10^{78}
Taille de l'univers.	10^{28} mètres
Taille de la galaxie.	10^{21} mètres
Masse de la terre.	10^{24} tonnes
Nombre de positions possibles aux dames.	10^{32}
Nombre de positions possibles à Othello.	10^{60}
Nombre de positions possibles aux échecs.	10^{93}

TAB. 1.1 – Quelques ordres de grandeur

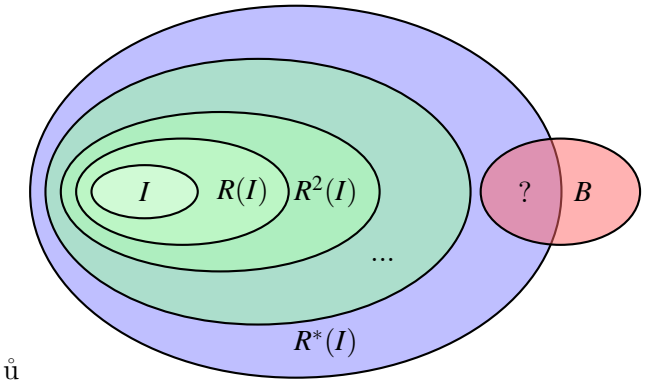


FIG. 1.2 – Analyse d'accessibilité

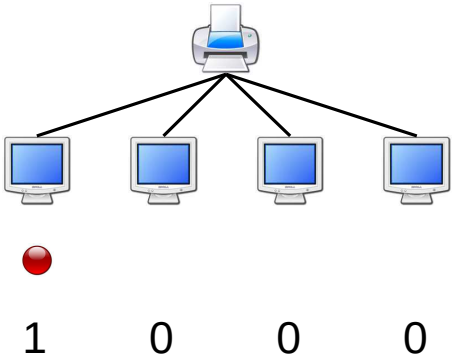
Il s'agit du problème classique d'accessibilité dans un graphe (éventuellement infini), donné implicitement. Selon la façon dont I et R sont codés, il existe de très nombreuses techniques, théories, heuristiques, etc, pour le résoudre (ou essayer de le résoudre).

Nous nous restreignons dans ce document au cadre particulier de l'analyse d'accessibilité régulière, appelée *model-checking régulier* dans le cadre de la vérification algorithmique. La particularité ici est de considérer que les états du système sont des mots (ou des termes) et que les ensembles considérés, comme I et B, sont réguliers. L'objectif est d'utiliser à la fois les propriétés de codage intéressantes des langages réguliers (automates finis, formules logiques, expressions régulières), les résultats de décidabilité (vide, inclusion, etc.), ainsi que l'efficacité des algorithmes associés.

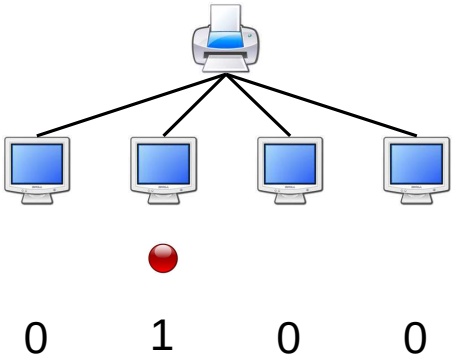
1.6 Model-checking Régulier

Le *model-checking régulier* [BG96, WB98, BJNT00] est un formalisme utilisé pour raisonner sur des systèmes ayant une infinité de configurations accessibles tels que, par exemple, les réseaux paramétrés de processus aux comportements identiques ou les systèmes communicants au travers de canaux FIFO.

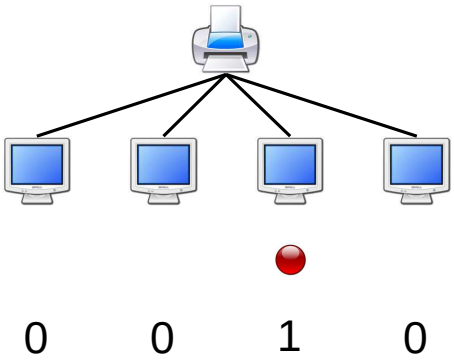
Prenons un exemple jouet de quatre machines reliées à une même imprimante. Dans ce système, la première machine a un jeton lui donnant droit d'accéder à l'impression, les autres machines n'ont pas de jeton. On code alors le système par le mot 1000. Le 1 indique la présence d'un jeton et le 0 son absence.



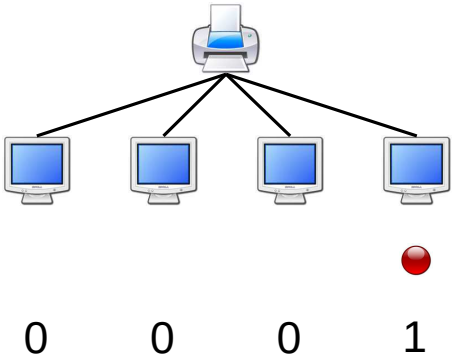
On considère que chaque imprimante peut, si elle a le jeton, le donner à sa voisine de droite. Le système peut alors évoluer comme suit :



Le système est alors codé par le mot 0100. Puis, après un nouveau passage de jeton :



Et enfin,



On peut généraliser ce protocole simple de passage d'un jeton de la façon suivante :

- Initialement, le jeton est sur la machine la plus à gauche. Les états initiaux du système sont donc tous les mots de : $I = 10^*$. Dans ce cas, la taille du mot initial code le nombre de machines.
- La relation de transitions R induite par la règle de réécriture $01 \rightarrow 10$ exprime que le jeton peut passer d'une machine à sa voisine de droite.
- Les états indésirables sont les états dans lesquels deux machines tenteraient d'accéder en même temps à l'imprimante, c'est-à-dire un mot contenant deux 1 : $B = (0+1)^*1(0+1)^*1(0+1)^*$.
- Pour la vérification, la question qu'on se pose est donc est-ce que $R^*(I) \cap B = \emptyset$?

Revenons au problème du model-checking régulier plus généralement qui peut s'écrire comme suit.

Model-checking régulier
Entrée : Un langage régulier I, une relation R, un langage régulier B.
Question : A-t-on $R^*(I) \cap B = \emptyset$?

Le problème du model-checking régulier est indécidable, même avec de nombreuses restrictions. Plusieurs approches sont donc possibles :

- On peut décomposer la relation R en k sous-relations $R_1, \ldots, R_k$ telles que $R = \cup_{1 \leq i \leq k} R_i$. Puis on essaye, à l'aide d'algorithmes ad hoc, de calculer des *accélérations* ou *métatransitions* T_i réalisant (si possible) une infinité d'éléments de $R_i^*(L)$, pour un langage régulier L. Autrement dit, il faut que $R_i(L) \subseteq T_i(L) \subseteq R_i^*(L)$ et, si possible, que $T_i(L) = R_i^*(L)$. Le calcul du point fixe est alors entrepris en utilisant le plus souvent possible ces accélérations. L'approche est semi-algorithmique : elle peut ne pas terminer.

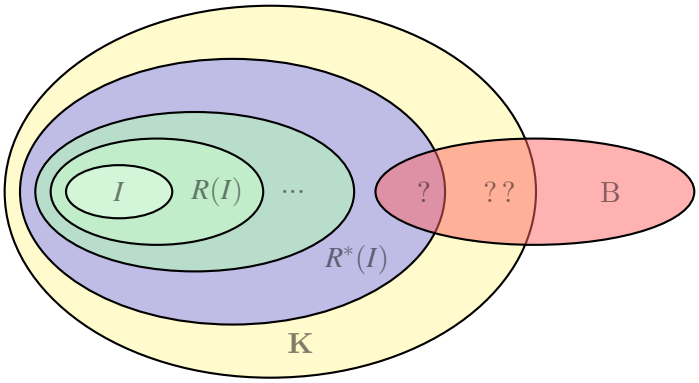


FIG. 1.3 – Approche par sur-approximation

– L'approche par approximations régulières consiste à ne pas chercher à calculer $R^*(I)$ mais à calculer un langage régulier K contenant $R^*(I)$ et dont l'intersection avec B est vide. Dans ce cas, par un simple argument d'inclusion, on sait que $R^*(I) \cap B$ est vide. A contrario, si $R^*(I) \cap K$ est non vide, on ne peut rien déduire sur $R^*(I) \cap B$. La difficulté de ce genre d'approche est d'élaborer des heuristiques pertinentes en pratique pour calculer K.
On peut facilement utiliser la même idée pour montrer qu'un terme est accessible en calculant un sous-language régulier J de $R^*(I)$ tel que $J \cap B \neq \emptyset$. Dans ce cas, on sait que $R^*(I) \cap B \neq \emptyset$.

1.7 Plan

Ce document est découpé en 7 chapitres, dont le premier est cette introduction et le dernier est un bilan personnel.

- Au **chapitre 2**, nous nous intéresserons à l'analyse d'accessibilité régulière sur les arbres par approximation en nous appuyant sur la technique (non automatique) de complétion introduite dans [GK00]. Plus particulièrement,
  – Nous montrons comment automatiser cette procédure dans le cadre de l'analyse de protocole cryptographiques [1, 12, 10, 7][2].
  – Nous exposons comment modifier la procédure de [GK00] afin de gérer des systèmes de réécriture non linéaires à gauche pour le calcul de sur-approximations [8], de sous-approximations [9]. Nous montrons aussi comment ces nouvelles procédures s'appliquent à l'analyse de protocoles cryptographiques. Nous proposons enfin une autre approche, par un algorithme original, pour les mêmes problèmes [4, 5].

[2]Les références numériques du type [1] sont des références personnelles.

  - Dans le cas où la procédure par sur-approximations ne conclurait pas, nous développons une procédure de raffinement d'approximations [3] s'appuyant sur le paradigme de *Counter-Example Guided Abstraction Refinment* [CGJ+00] dans le cas où le système de réécriture est linéaire. Nous étudions aussi les limites de ce type d'approches [6].
  - Nous explorons aussi comment les procédures pour le calcul automatique de sur-approximations peuvent s'utiliser dans le cadre de la vérification de propriétés de trace utiles en analyse statique [16].
- Le **chapitre 3** est dédié au calcul d'accélérations de relations de semi-commutation, qui sont des relations sur les mots très utiles pour la modélisation des systèmes.
  - Nous montrons comment calculer le R-mélange de deux langages réguliers et comment ce résultat permet d'obtenir une preuve simple du résultat principal de [BMT07] et dont découle un algorithme efficace de calcul d'accélrations [15].
  - Nous exposerons ensuite le résultat théorique le plus marquant en exhibant une classe de langages, PolCom, close par semi-commutation [15].
  - Nous revenons vers des problématiques de vérification en montrant comment ce résultat théorique peut s'adapter à l'analyse d'accessibilité [13, 14].
  - Enfin, nous nous attardons sur l'extension des resultats de [BMT07] aux langages d'arbres [20].
- Nous quittons la problématique d'analyse d'accessibilité au **chapitre 4** en étudiant des problèmes de vérification pour des systèmes composés.
  - Nous commençons par présenter des résultats de décision pour des notions de substitutivité à la fois qualitatives et quantitatives [23, 21, 24, 22].
  - Nous étudions ensuite le problème de composition pour des classes d'automates gardés par des formules booléennes [2].
  - Nous terminons en donnonant quelques propriétés sur les langages clos par le haut pour la relation sous-mot [19].
- Nous changeons d'orientation dans le **chapitre 5** où nous nous intéressons au test aléatoire. Après avoir montré comment une génération aléatoire d'automates finis déterministes [BN07] pouvait s'utiliser dans le cadre du test à partir de modèles [17], nous montrons comment générer aléatoirement et uniformément des automates d'arbres [26]. Nous présentons aussi l'outil Seed développé pour générer aléatoirement et uniformément des structures récursives vérifiant des motifs définis par des grammaires [25].
- Nous terminons l'exposé de nos contributions dans le **chapitre 6** en revenant à deux problèmes d'analyse d'accessibilité : nous montrons que le calcul du noyau abélien d'un monoïde fini peut se faire en temps poly-

nomial [18] (alors que le seul algorithme connu était exponentiel). Nous terminons par une approche semi-algorithmique [11] afin de s'attaquer au problème de la positivité des automates max-plus, problème indécidable [Kro94] mais dont l'utilité en terme de vérification est exposée au chapitre 4. prometteuses.

1.8 Remarques sur les notations

Les notations utilisées dans ce manuscrit sont classiques. Nous les redéfinissons dans un soucis de simplicité. En revanche, les objets (eux aussi classiques), ne seront pas redéfinis. Le lecteur intéressé pourra se référer à [CDG+02] pour les automates d'arbres et à [Sak03] pour les automates de mots.

Ensembles, applications, etc.
Si X est un ensemble fini, $|X|$ désigne le cardinal de X. Si X et Y sont deux ensembles, X^Y désigne l'ensemble des applications de Y dans X et 2^X l'ensemble des parties de X.

Si $\mathcal{R}$ est une relation sur $X \times Y$ et $Z \subseteq X$, on note $\mathcal{R}(Z)$ l'ensemble des y de Y tels qu'il existe $z \in Z$ vérifiant $(z,y) \in \mathcal{R}$. On note $\mathcal{R}^*$ la clôture réflexive-transitive de $\mathcal{R}$. On écrit parfois $x\mathcal{R}y$ pour $(x,y) \in \mathcal{R}$.

Mots et automates de mots.
Si u est un mot fini sur l'alphabet A, on note $u(i)$ la i-ème lettre de u si elle existe. La longueur de u est noté $|u|$. On note $\alpha(u)$ l'ensemble des lettres apparaissant dans u, c'est-à-dire l'image de u. On note A^* le monoïde libre engendré par A, c'est-à-dire l'ensemble des mots sur A. Notamment, $\mathbb{N}^*$ désigne l'ensemble des mots sur l'alphabet $\mathbb{N}$ et non $\mathbb{N} \setminus \{0\}$ comme c'est l'usage en mathématiques.

Un automate fini est noté (Q,A,E,I,F), où Q est l'ensemble des états, A est l'alphabet, E l'ensemble des transitions, I l'ensemble des états initiaux et F l'ensemble des états finaux. Le langage reconnu par un automate A est noté $L(\mathcal{A})$. Graphiquement, les états initiaux d'un automate sont représentés par des ronds où entrent une petite flèche et les états finaux par un double cercle.

Termes, substitutions et automates d'arbres.
Si $\mathcal{F}$ désigne un alphabet muni d'une fonction d'arité, on note $\mathcal{T}(\mathcal{F})$ l'ensemble des termes sur F. L'ensemble des positions d'un terme t de $\mathcal{T}(\mathcal{F})$ est noté $\mathcal{P}\mathrm{os}(t)$. Si p est une position de t, $t(p)$ désigne le symbole en position p de t, $t_{|p}$ le sous-terme de p enraciné en position p. De plus, si s est un terme, $t[s]_p$ est le terme obtenu à partir de t en substituant le sous-terme en position p par s. Si $\mathcal{X}$ est un ensemble de variables (disjoint de $\mathcal{F}$), on note par $\mathcal{T}(\mathcal{F},\mathcal{X})$ l'ensemble $\mathcal{T}(\mathcal{F} \cup \mathcal{X})$, où les éléments de $\mathcal{X}$ sont d'arité 0.

Si $\mathcal{X}$ est un ensemble de variables et Y un ensemble, un élément de $A^{\mathcal{X}}$ est appelé une substitution. Si t est un terme de $\mathcal{T}(\mathcal{F},\mathcal{X})$, et σ une substitution

de A^X, on note $t\sigma$ le terme obtenu à partir de t en remplaçant chaque $x \in \mathcal{X}$ par $\sigma(x)$. De plus, $\mathcal{V}\mathrm{ar}(t)$ désigne l'ensemble des variables apparaissant dans t, c'est-à-dire $\{t(p) \mid p \in \mathcal{P}\mathrm{os}(t),\ t(p) \in X\}$.

Un automate d'arbre est, sauf mention contraire, un automate d'arbre de bas en haut (*bottom-up*). Un automate d'arbre est un quadruplet $(\mathcal{F}, Q, \Delta, Q_f)$ où Q est l'ensemble des états, Δ l'ensemble des transitions et Q_f l'ensemble des états finaux.

Si $\mathcal{R}$ est un système de réécriture, on note par $\rightarrow_{\mathcal{R}}$ la relation sur $\mathcal{T}(\mathcal{F}) \times \mathcal{T}(\mathcal{F})$ induite par $\mathcal{R}$. Si $\mathcal{A} = (\mathcal{F}, Q, \Delta, Q_f)$ est un automate d'arbre, on note aussi $\rightarrow_{\mathcal{A}}$ la relation $\rightarrow_{\Delta}$ sur $\mathcal{T}(\mathcal{F}, Q) \times \mathcal{T}(\mathcal{F}, Q)$.

Chapitre 2

Approximations régulières de langages d'arbre

Sommaire

2.1 Analyse d'accessibilité régulière pour les langages d'arbres 20
2.1.1 Analyse d'accessibilité régulière sur les arbres 20
2.1.2 Vérification symbolique de protocoles cryptographiques . . 23
2.1.3 Complétion d'automates d'arbre 27
2.2 Automatiser la complétion 31
2.2.1 Problème de la $\mathcal{R}$-cohérence et automatisation 31
2.2.2 Expérimentations : outil TA4SP 33
2.3 Gérer les systèmes non linéaires à gauche 35
2.3.1 Problématique 35
2.3.2 Modification de la complétion 35
2.3.3 Le cas des règles quadratiques 37
2.3.4 Résultats expérimentaux 38
2.3.5 Non-linéarité et déterminisme 38
2.4 Raffinement d'approximations 40
2.4.1 Raffinement d'approximations pour les systèmes linéaires . 41
2.4.2 Limite théorique 44
2.5 Vérifier des propriétés temporelles 44
2.5.1 Contexte scientifique 44
2.5.2 Trois motifs de propriétés 46
2.5.3 Une approche par automates avec contraintes globales . . 46
2.5.4 Algorithmes 48
2.6 Conclusion et perspectives 50

Les publications liées à ce chapitre sont [1, 3, 4, 5, 6, 8, 9, 10, 12, 16].

Nous nous intéressons dans ce chapitre à l'analyse d'accessibilité régulière sur les arbres avec, comme application principale, l'analyse de protocoles de sécurité.

Section 2.1. Cette section est dédiée à la présentation générale du problème. Après un bref état de l'art sur l'analyse régulière d'accessibilité sur les arbres en section 2.1.1, puis sur l'analyse de protocoles en section 2.1.2, la technique de complétion [Gen98, FGT04] sera présentée dans la section 2.1.3.

Section 2.2. Cette section présente les travaux que nous avons effectués afin d'automatiser la procédure de complétion (section 2.2.1), ainsi que les résultats obtenus expérimentalement pour l'analyse de protocoles (section 2.2.2). Les résultats de cette section ont donné lieu aux publications [1, 10, 12].

Section 2.3. La procédure de complétion définie dans [Gen98, FGT04] est fortement contrainte, pour les applications, par une condition de linéarité à gauche sur le système de réécriture. Nous montrons dans cette section comment résoudre ce problème, en proposant une nouvelle procédure de complétion, plus coûteuse algorithmiquement, mais moins restrictive. Les résultats de cette section ont été publiés dans [8, 9]. Nous exposerons aussi les résultats de [4, 5] qui explorent une autre piste pour le même problème.

Section 2.4. Nous définissons dans cette section une procédure de raffinement d'approximations pour les systèmes de réécritures linéaires (section 2.4.1). Une limite générale de l'approche est présentée dans la section 2.4.2. Ces travaux ont donné lieu aux publications [3, 6].

Section 2.5. Nous montrons dans cette section comment les résultats de calculs de sur-approximations peuvent être utilisés pour semi-décider des propriétés plus complexes de trace définis par des motifs. Les résultats de cette section ont été publiés dans [16].

2.1 Analyse d'accessibilité régulière pour les langages d'arbres

2.1.1 Analyse d'accessibilité régulière sur les arbres

L'analyse d'accessibilité sur les langages d'arbre a fait l'objet de nombreux travaux. Il s'agit d'une problématique générale de la réécriture, dont le cadre dépasse celui de la vérification. La notion d'automate d'arbre a d'ailleurs été largement développée dans le contexte théorique de la réécriture.

Indécidabilité

Accessibilité Régulière par réécriture
Entrée : Deux langages réguliers L_1 et L_2, un système de réécriture $\mathcal{R}$
Question : A-t-on $\mathcal{R}^*(L_1) \cap L_2 = \emptyset$?

Le problème d'accessibilité régulière est indécidable. On peut, en effet, très facilement coder dans ce formalisme des problèmes classiques comme le problème de correspondance de Post ou le problème du test du vide d'une machine de Turing. Il est intéressant de savoir qu'il est indécidable de savoir si $\mathcal{R}^*(L_1)$ est régulier, même pour des systèmes linéaires et confluents [GT95]. Le problème est même indécidable pour des systèmes de réécriture linéaires sur des mots [Sak92] et même lorsque L_1 est sans-étoile [MP96].

Analyse exacte

Plusieurs travaux ont pour objectif de trouver des classes de systèmes de réécriture pour lesquels $\mathcal{R}^*(L)$ est régulier pour tout L régulier. Les principaux travaux dans ce sens sont :

[Sal88] pour les systèmes linéaires droits et monadiques (les parties droites des règles sont de profondeur 1 et ne contiennent que des variables sur les feuilles).

[DT90] pour les systèmes clos.

[Jac96] pour les systèmes décroissants (chaque partie droite est soit une variable, soit un terme clos, soit un terme dans lequel les variables qui apparaissent à gauche sont de profondeur 1).

[TKS00] pour les *Right Linear Path Overlapping TRS*, classe qui contient toutes les précédentes.

[STFK02] pour les *Layered Transducing TRS*, classe incomparable avec celle de [TKS00].

[OT05] construit l'ensemble $\mathcal{R}^*(L)$ dans un cadre particulier d'automates utilisant des symboles associatifs et commutatifs.

On peut aussi citer [Rét99] qui restreint la classe des langages réguliers pour relâcher les contraintes sur le système de réécriture.

Model-checking régulier sur les arbres

Un des premiers résultats d'accélération pour les langages d'arbre est [BT02], où il est montré comment une classe particulière de transducteurs d'arbre peut être accélérée. Dans [ALdR06] une approche par itération des transducteurs et réduction par simulation est proposée. Dans ces deux travaux, l'objectif est de calculer, si possible, un transducteur codant $\mathcal{R}^*$ (plus exactement la clôture réflexive-transitive de la relation induite par $\mathcal{R}$). Dans

[BHRV06], les auteurs étendent l'approche développée dans [BHV04] au model-checking régulier s'appuyant sur des abstractions et des raffinements.

Approches par complétion

La technique de complétion, dans sa version générale et pour obtenir des sur-approximations de $\mathcal{R}^*(L)$, a été introduite dans [Gen98], puis généralisée (notamment pour les sous-approximations et les cas exacts) dans [FGT04]. La technique de complétion a été implémentée par Th. Genet dans l'outil TIMBUK[1] dont la version 3 propose d'utiliser des approximations définies par des systèmes d'équations [GR09]. La complétion a aussi été redéfinie récemment dans [GR08] dans le cadre plus général des systèmes définis par des clauses de Horn. De plus, il est proposé dans [GR08] d'intéressantes pistes algorithmiques pour appliquer la complétion en utilisant des BDD. On peut noter aussi qu'en se basant sur l'outil de réécriture TOM [Mor00], une implantation particulièrement efficace de la complétion est exposée dans [BBGM08]. Enfin, [BG06] explore des pistes algorithmiques pour reconstruire des traces de contre-exemples à partir d'une sur-approximation obtenue par complétion.

Applications des approximations régulières

En combinant la technique de complétion avec des techniques d'interprétation abstraite, [Tak04] montre comment utiliser des approximations régulières de langages d'arbres pour la vérification.

Dans le cadre de l'analyse de protocoles, dont nous reparlerons tout au long de ce chapitre, le premier travail a été fait dans [GK00] pour une vérification du protocole `NSPK` (avec des approximations spécifiques au problème). Une première approche automatique, pour `NSPK`, a été proposée dans [OCKS02]. En utilisant des contraintes particulières et des approximations définies *à la main* (par opposition à *calculées par un algorithme*), [GTTT03] propose une vérification d'un protocole utilisant le ou-exclusif.

Dans le cadre de l'analyse statique de programmes une première approche de l'utilisation de la complétion pour vérifier des propriétés d'appartenance de classe est proposée dans [BGJR07]. Notons que, dans ce travail, les approximations ont été obtenues à la main, mais en utilisant des équations, de façon préliminaire à [GR09]. Toujours avec un objectif d'analyse statique, la procédure de complétion à été certifiée dans [BGJ08] de la façon suivante : on dispose d'un programme certifié en Coq qui vérifie qu'un langage régulier L_1 donné par un automate obtenu par complétion à partir de $\mathcal{R}$ et L (lui aussi donné par un automate d'arbre) satisfait bien $L \subseteq L_1$ et $\mathcal{R}(L_1) \subseteq L_1$, ce qui implique que $\mathcal{R}^*(L) \subseteq L_1$.

[1] `http://www.irisa.fr/lande/genet/timbuk`.

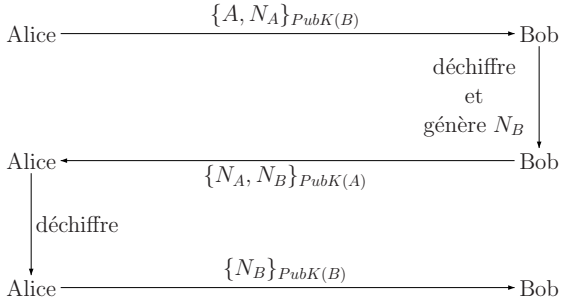


FIG. 2.1 – Protocole `NSPK`

2.1.2 Vérification symbolique de protocoles cryptographiques

Protocole NSPK

Nous allons illustrer les problèmes pouvant survenir sur le célèbre exemple du protocole de Needham-Shröder, considéré comme étant sûr pendant de nombreuses années, avant que Lowe [Low95] montre une attaque.

Ce protocole, illustré par la figure 2.1, utilise un système à clé publique. Alice souhaite entamer une communication sécurisée avec Bob. Elle connaît la clé publique de Bob et réciproquement. L'objectif de ce protocole est qu'Alice soit persuadée de communiquer avec Bob et Bob avec Alice. Au cours de ce protocole ils mettront au point une information secrète partagée qui leur servira par exemple de clé secrète de chiffrement.

1. **Étape préliminaire :** Au début du protocole Alice génère, sur sa machine, un nombre aléatoire - appelé *Nonce* dans le vocabulaire des protocoles - que l'on note N_A. Ce nombre aléatoire va servir de défi pour Bob.
2. **Premier message :** Alice envoie à Bob un message contenant son identité (notée A sur la figure) et le nombre N_A. Ce message est chiffré avec la clé publique de Bob, ce que l'on note $\{A,N_A\}_{PubK(B)}$.
3. **Bob reçoit le message :** Pour le moment, Bob reçoit juste un message. Il ne sait ni d'où il vient, ni qui l'a composé. Il le déchiffre avec sa clé privée et en découvre le contenu : l'identité d'Alice et le nombre N_A.
4. **Second message :** Bob souhaite convaincre Alice qu'il est bien Bob. Pour cela il va renvoyer à Alice le nombre N_A que lui seul peut connaître dans la mesure où lui seul connaît sa clé privée. Mais comme il veut lui aussi être sûr de communiquer avec Alice, il va a son tour générer un

nombre aléatoire N_B. Il envoie alors un message contenant N_A et N_B et chiffré avec la clé publique d'Alice.

5. **Alice reçoit le message :** Alice reçoit un message, qu'elle déchiffre avec sa clé privée. Dans ce message, elle reconnaît N_A, que seul Bob pouvait déduire du premier message $\{A, N_A\}_{PubK(B)}$. Une personne écoutant ce qui se passe sur le réseau ne pouvant accéder à cette information, elle peut être persuadée de bien communiquer avec Bob. Il lui reste à convaincre Bob qu'elle est bien Alice (n'importe qui aurait pu envoyer à Bob le premier message).
6. **Dernier message :** Alice renvoie à Bob le nombre N_B, chiffré avec la clé publique de Bob.
7. **Bob reçoit le message :** Bob reçoit un message qu'il ouvre avec sa clé privé. Il reconnaît N_B que seule Alice a pu déduire du message $\{N_A, N_B\}_{PubK(A)}$ qu'il a envoyé précédemment. Il peut donc être persuadé de bien communiquer avec Alice.

Par la suite, les nombres N_B ou N_A peuvent être utilisés pour garantir à Alice et Bob qu'ils communiquent bien ensemble[2].

Si l'on regarde ce protocole, on voit qu'un individu malveillant espionnant le réseau ne récupère que des informations chiffrées avec les clés d'Alice ou Bob. Il ne peut donc rien en tirer (on suppose que la cryptographie est parfaite).

Cependant, si Bob est malhonnête, Lowe a montré la faille décrite ci dessous et illustrée dans la figure 2.2.

1. **Étape préliminaire :** Alice génère N_A.
2. **Premier message :** Comme prévu Alice envoie le message

$$\{A, N_A\}_{PubK(B)}$$

3. **Bob est malhonnête :** Bob reçoit le message d'Alice. Il l'ouvre et se dit qu'il va jouer un mauvais tour à Charlie. Au lieu de poursuivre le protocole, il chiffre le message qu'il vient de déchiffrer avec la clé publique de Charlie et lui envoie. Bob se fait passer pour Alice.
4. **Charlie reçoit le message de Bob :** Charlie reçoit un message chiffré avec sa clé publique. Il l'ouvre et découvre l'identité d'Alice ainsi qu'un nombre aléatoire. Il pense donc que ce message vient d'Alice. Il génère donc un nombre aléatoire N_C.
5. **Message de Charlie pour Alice :** Poursuivant le protocole, Charlie envoie à Alice un message contenant N_A et N_C, le tout chiffré avec la clé publique d'Alice.
6. **Alice reçoit le message de Charlie :** Alice reçoit un message contenant deux nombres aléatoires, l'un étant le défi N_A qu'elle a envoyé à

[2]Il y a cependant des précautions à prendre sur la manière dont ils sont utilisés : des failles de sécurité peuvent apparaître lorsque l'on compose, même séquentiellement, des protocoles.

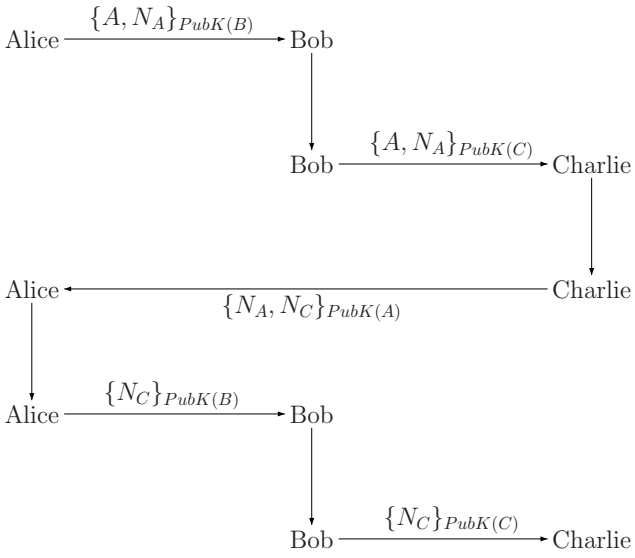


FIG. 2.2 – Faille du protocole NSPK

Bob. Elle pense donc que ce message provient de Bob. Elle poursuit donc le protocole.

7. **Alice répond à Bob :** Pour cela, elle envoie à Bob un message, chiffré avec la clé publique de Bob et contenant le nonce N_C.
8. **Bob répond à Charlie :** Bob peut maintenant terminer sa session avec Charlie. Il lui envoie le nonce N_C chiffré avec la clé publique de Charlie. Charlie pense avoir reçu une réponse d'Alice.

Comme on peut le remarquer, Bob arrive à se faire passer pour Alice auprès de Charlie, sans même avoir besoin de savoir intercepter des messages. Il n'a pas non plus "cassé" de message chiffré. Il lui suffit de jouer les deux sessions en parallèle. Ce type de faille est connue sous le nom de *Man in the middle attack*. Si l'on considère que Bob peut de plus intercepter des messages, les possibilités d'attaques de ce type sont plus nombreuses encore.

Vérification de protocoles

La recherche sur la vérification de protocoles est extrêmement active. Nous ne citons que quelques références pouvant servir de pointeurs.

Étant donné un protocole cryptographique utilisant les primitives usuelles (chiffrement, déchiffrement, composition, nonce), savoir s'il existe une attaque[3] de type *man-in-the-middle* est indécidable [DLMS99]. En revanche, si l'on fixe le nombre maximal de sessions pouvant s'exécuter en parallèle, le problème est NP-complet [RT03]. De nombreux résultats de complexité ont été prouvés pour différentes variantes. Les principaux sont résumés dans [CDL06].

Au début des années 2000, la recherche sur la vérification de protocoles s'est orientée vers des résultats plus précis : de nombreux protocoles utilisent des fonctions ayant des propriétés algébriques (par exemple le ou-exclusif est nilpotent, certains chiffrements sont commutatifs, etc.), qui sont indispensables au bon fonctionnement du protocole mais qui peuvent aussi être source d'attaques. A nouveau [CDL06] expose de nombreuses attaques de ce type et présente les résultats principaux de décidabilité du domaine.

Le modèle présenté ici pour les protocoles est appelé *modèle symbolique* : les messages sont des termes. Parallèlement, l'approche cryptographique, où les messages sont des suites de bits, s'appuie sur des probabilités et la théorie de la complexité. Dans ce cadre, la cryptologie n'est plus considérée comme parfaite. C'est ce qu'on appelle le modèle *computationel*. Actuellement, les travaux sur la vérification de protocoles, comme par exemple [CW05], s'orientent vers la convergence des deux modèles.

Parallèlement, des travaux comme [DKR09] explorent la vérification de propriétés plus complexes, par exemple pour la certification de protocoles de

[3]Le modèle utilisé est celui de Dolev-Yao : la cryptographie est supposée parfaite, en revanche l'intrus a tout pouvoir sur le réseau (interception, envoi, etc.).

vote.

De nombreux outils ont été développés pour la vérification symbolique de protocoles : AVISPA [1], ProVerif [Bla01], Hermes [BLP03], etc.

2.1.3 Complétion d'automates d'arbre

Dans cette section, $\mathcal{S}$ est un ensemble infini et $\mathcal{F}$ un alphabet fini muni d'une fonction d'arité.

Soit $\mathcal{A} = (\mathcal{F}, Q, \Delta, Q_f)$ un automate d'arbre tel que $Q \subseteq \mathcal{S}$ et $\mathcal{R}$ un système de réécriture sur $\mathcal{F}$. Une *fonction d'approximation* γ pour $\mathcal{A}$ et $\mathcal{R}$ est une fonction de $Q^{\mathcal{X}} \times Q \times \mathcal{R}$ dans $\mathcal{S}^{\mathbb{N}^*}$ (des contraintes sur le domaine de γ seront imposées plus loin).

La *normalisation* (modulo une fonction d'approximation γ) est une fonction qui a un élément de $Q^{\mathcal{X}} \times Q \times \mathcal{R}$ associe l'ensemble :

$$\begin{aligned}\mathrm{Norm}_\gamma(\sigma, q, l \to r) = \{ & r(p)(q_1^p, \ldots, q_n^p) \to q^p \mid p \in \mathcal{P}\mathrm{os}(r) \text{ et} \\ & q^\varepsilon = q \text{ et} \\ & r(p) \text{ est d'arité } n \text{ et} \\ & q_i^p = \gamma(\sigma, q, l \to r)(p \cdot i) \text{ si } r(p) \notin \mathcal{X} \text{ et} \\ & q_i^p = \sigma(r(p)) \text{ si } r(p) \in \mathcal{X} \text{ et} \\ & q^p = \gamma(\sigma, q, l \to r)(p) \text{ pour } p \neq \varepsilon\}\end{aligned}$$

Les fonctions d'approximation doivent être définies afin que toutes les normalisation soient possibles.

La fonction de normalisation a donc pour image un ensemble de transitions sur $\mathcal{S}$. Une *paire critique* est un couple $(r\sigma, q)$ où r est une partie droite d'une règle $l \to r$ de $\mathcal{R}$, q un état de $\mathcal{A}$ et σ un élément de $Q^{\mathcal{X}}$ telle que $r\sigma \not\to_{\mathcal{A}}^* q$ et $l\sigma \to_{\mathcal{A}}^* q$.

Le γ-*complété* de $\mathcal{A} = (\mathcal{F}, Q, \Delta, Q_f)$ est l'automate $(\mathcal{F}, Q_1, \Delta_1, Q_f)$, noté $\mathcal{C}_\gamma^{\mathcal{R}}(\mathcal{A})$, et défini par

- Δ_1 est l'union de Δ et de tous les $\mathrm{Norm}_\gamma(\sigma, q, l \to r)$, pour lesquels $(r\sigma, q)$ est une paire critique (elles sont en nombre fini).
- Q_1 est l'union de Q et de tous les états (dans $\mathcal{S}$) apparaissant dans les transitions de Δ_1.

Considérons par exemple l'automate[4] et le système de réécriture décrits dans la figure 2.3 ; l'automate reconnaît les termes de la forme

$$b(a(b(a(\ldots b(a(\bot))\ldots)))).$$

[4] Les automates illustrant les notions sont des automates de mots pour lesquels on dispose d'une représentation graphique. Afin de rester cohérent avec les notations sur les termes, on ajoute un symbole de constante $\bot$ et les mots ne sont pas écrits comme usuellement de gauche à droite, mais comme des fonctions. Par exemple le mot *abcb* est représenté par le terme $b(c(b(a(\bot))))$.

Dans ce cas, il n'y a qu'une paire critique $(b(a(q_2)), q_2)$. Soit γ_1 une fonction d'approximation telle que

$$\gamma_1(\{x \mapsto q_2\}, q_2, a(b(x)) \to b(a(x))) = \{1 \mapsto q_3\}.$$

FIG. 2.3 – Exemple de complétion

Dans ce cas, le γ_1-complété de l'automate de la figure 2.3 est l'automate de la figure 2.4. Sur ce nouvel automate, il n'y a qu'une paire critique $(b(a(q_3)), q_3)$. Soit γ_2 une fonction d'approximation telle que

$$\gamma_2(\{x \mapsto q_3\}, q_3, a(b(x)) \to b(a(x))) = \{1 \mapsto q_4\}.$$

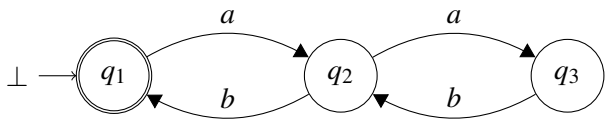


FIG. 2.4 – Exemple après une étape de complétion

Avec la fonction γ_2 appliquée à l'automate de la figure 2.4, on obtient l'automate de la figure 2.5.

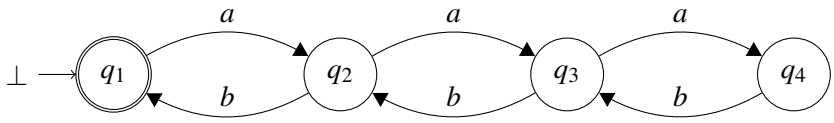


FIG. 2.5 – Exemple après deux étapes de complétion

En utilisant sur l'automate de la figure 2.4 une fonction γ_3 une telle que

$$\gamma_3(\{x \mapsto q_3\}, q_3, a(b(x)) \to b(a(x))) = \{1 \mapsto q_2\},$$

on obtient l'automate de la figure 2.6, qui n'a aucune paire critique.
De manière analogue, sur l'automate de la figure 2.4, utiliser une fonction γ_4 telle que

$$\gamma_4(\{x \mapsto q_3\}, q_3, a(b(x)) \to b(a(x))) = \{1 \mapsto q_1\}$$

produit l'automate de la figure 2.7, qui n'a aucune paire critique.

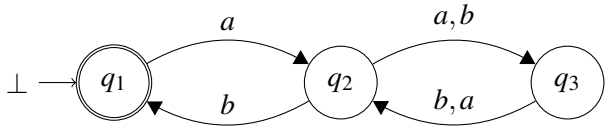


FIG. 2.6 – Exemple après deux (autres) étapes de complétion

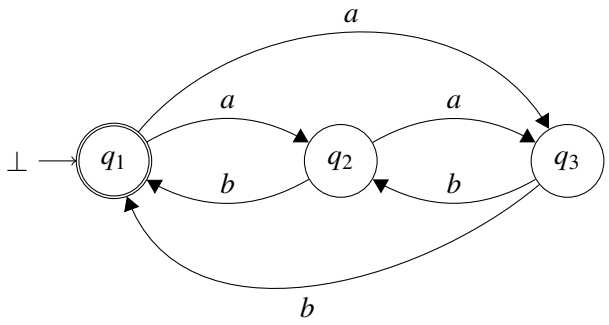


FIG. 2.7 – Autre exemple après deux (autres) étapes de complétion

Nous continuons maintenant à exposer les résultats sur la complétion. Une fonction d'approximation est $(\mathcal{A},\mathcal{R})$-*exacte* si toutes les fonctions $\gamma(\sigma,q,l\to r)$ sont injectives, si leurs ensembles images sont dans $\mathcal{S}\setminus Q$ et si ces ensembles images sont deux à deux disjoints[5]. Notons que pour tout couple $\mathcal{A},\mathcal{R}$ il existe une fonction d'approximation $(\mathcal{A},\mathcal{R})$-exacte. De plus, pour tout couple $\mathcal{A},\mathcal{R}$, et pour tout couple de fonctions d'approximation $(\mathcal{A},\mathcal{R})$-exactes, les complétés de $\mathcal{A}$ par ces fonctions sont isomorphes. Lorsque le contexte le permet, c'est-à-dire que le nom des états n'est pas important, on note $\mathcal{C}^{\mathcal{R}}_{\text{exact}}(\mathcal{A})$ un représentant de cette classe d'équivalence.

Un automate $\mathcal{A}=(\mathcal{F},Q,\Delta,Q_f)$ est $\mathcal{R}$-*cohérent* si pour tout $\mu\in\mathcal{T}(\mathcal{F})^{\mathcal{X}}$, tout $l\to r\in\mathcal{R}$, tout $q\in Q$, si $l\mu\to^*_{\mathcal{A}} q$ alors il existe $\sigma\in Q^{\mathcal{X}}$ tel que $l\mu\to^*_{\mathcal{A}} l\sigma\to^*_{\mathcal{A}} q$. Notons que tout automate est $\mathcal{R}$-cohérent avec tout système linéaire à gauche (il suffit de prendre pour σ la substitution que l'on obtient lors d'un calcul de $\mathcal{A}$ sur $l\mu$). Les principaux résultats relatifs à la procédure de complétion sont donnés dans le théorème ci-dessous [FGT04].

Théorème 2.1 *Soit $\mathcal{A}$ un automate d'arbre, $\mathcal{R}$ un système de réécriture et γ une fonction d'approximation. Si $\mathcal{A}$ est $\mathcal{R}$-cohérent, alors $L(\mathcal{A})\cup\mathcal{R}(L(\mathcal{A}))\subseteq L(\mathcal{C}^{\mathcal{R}}_{\gamma}(\mathcal{A}))$. Si $\mathcal{A}$ est déterministe, alors $L(\mathcal{A})\cup\mathcal{R}(L(\mathcal{A}))\subseteq L(\mathcal{C}^{\mathcal{R}}_{\gamma}(\mathcal{A}))$. Si $\mathcal{R}$ est linéaire droit et si γ est $(\mathcal{A},\mathcal{R})$-exacte, alors $L(\mathcal{C}^{\mathcal{R}}_{\gamma}(\mathcal{A}))\subseteq\mathcal{R}^*(L(\mathcal{A}))$.*

[5]Intuitivement, cela signifie que lors des normalisations, les états utilisées seront à chaque fois différents.

Dans le cadre de la vérification, la technique de complétion est plutôt utilisée pour calculer des sur-approximations, la première partie du théorème est plus importante et s'utilise comme suit : on se donne tout d'abord un automate $\mathcal{A}_0$, un système de réécriture $\mathcal{R}$ et une famille (γ_n) de fonctions d'approximation. La suite d'automates $(\mathcal{A}_i)_{i\geq 0}$ est définie récursivement par $\mathcal{A}_{i+1} = \mathcal{C}^{\mathcal{R}}_{\gamma_i}(\mathcal{A}_i)$. Cette suite est une suite croissante (à la fois pour les ensembles d'états et les ensembles de transitions). Par ailleurs, si $\mathcal{A}_i = \mathcal{A}_{i+1}$ alors cette suite est ultimement constante à partir de $\mathcal{A}_i$. On utilise donc le semi-algorithme d'approximation suivant.

Semi-algorithme 2.2

Nom : PointFixeParCompletion
Entrées : $\mathcal{A}_0$, (γ_n), $\mathcal{R}$
Variables : i, $\mathcal{A}$, $\mathcal{B}$
Début
  $i := 1$
  $\mathcal{A} := \mathcal{A}_0$
  $\mathcal{B} := \mathcal{C}_{\gamma_0}(\mathcal{A}_0)$
  TantQue $(\mathcal{A} \neq \mathcal{B})$
    $\mathcal{A} := \mathcal{B}$
    $\mathcal{B} := \mathcal{C}_{\gamma_i}(\mathcal{B})$
    $i := i + 1$
  FinTanQue
  Retourner $\mathcal{B}$.
Fin

Plusieurs remarques sont à faire sur ce semi-algorithme.

- Tout d'abord, la terminaison n'est pas garantie et dépend des fonctions d'approximation choisies. En général, on se donne un sous-ensemble fini de $\mathcal{S}$ dans lequel les images des fonctions d'approximation peuvent prendre leurs valeurs ; ce qui garantit par un argument de monotonie la terminaison.
- En cas de convergence, le théorème 2.1 garantit que $\mathcal{R}^*(L(\mathcal{A}_0)) \subseteq L(\mathcal{B})$ à condition que chaque $\mathcal{A}_i$ calculé soit $\mathcal{R}$-cohérent. La définition de la normalisation ne le garantit aucunement : la correction du semi-algorithme $(\mathcal{R}^*(L(\mathcal{A}_0)) \subseteq L(\mathcal{B}))$ nécessite de tester à chaque étape que $\mathcal{A}_i$ est $\mathcal{R}$-cohérent.
- Pour contourner la condition de cohérence, on pourrait être tenté de déterminiser l'automate $\mathcal{B}$ à chaque étape. Cependant, il s'agit d'une opération exponentielle : le nombre d'états va très vite rendre impossible la recherche des paires critiques.

– Le test $\mathcal{A} \neq \mathcal{B}$ est facile à faire. Il ne s'agit nullement d'un test de non-isomorphisme, mais bien d'un test d'inégalité, qui se résout en temps constant si on implémente le test lors de la complétion (ajoute-t-on des transitions ?). Avoir un test $L(\mathcal{A}) \neq L(\mathcal{B})$ terminerait plus souvent et en moins d'étapes mais est EXPTIME-complet.
– Sur le plan algorithmique, l'étape lourde est $\mathcal{B} := \mathcal{C}_{\gamma_i}(\mathcal{B})$ qui nécessite de trouver les paires critiques. L'existence d'une paire critique est un problème NP-complet. Les énumérer peut demander un temps exponentiel.

2.2 Automatiser la complétion

Dans un contexte de vérification, on souhaite avoir des techniques automatiques, les plus simples possibles à utiliser. La procédure de complétion n'est pas automatique dans la mesure où elle exige des fonctions d'approximation que l'on souhaite rapides à calculer et qui préservent la $\mathcal{R}$-cohérence. Si les systèmes de réécriture étudiés sont linéaires à gauche, la $\mathcal{R}$-cohérence est garantie. Cependant, en pratique, le système $\mathcal{R}$ provient de la modélisation et il n'est pas linéaire à gauche. Nous allons présenter une famille de fonctions d'approximation qui :

– Sont calculées automatiquement (et dynamiquement),
– Préservent la $\mathcal{R}$-cohérence dans le cadre applicatif de l'analyse de protocole,
– Permettent des normalisations en temps polynomial (mais pas la recherche des paires critiques),
– Concluent dans presque tous les cas pratiques.

2.2.1 Problème de la $\mathcal{R}$-cohérence et automatisation

Lorsque l'on analyse les protocoles cryptographiques, on doit gérer des règles du type

$$\mathsf{And}(\mathsf{Crypt}(\mathsf{Message}(x), \mathsf{Key}(y)), \mathsf{Key}(y)) \to \mathsf{Message}(x)$$

qui codent le fait qu'à partir d'un message chiffré avec une clé et de cette clé de chiffrement (on suppose que le chiffrement est symétrique), on peut déduire le contenu du message. Cette règle est non linéaire à gauche (la variable y apparaît deux fois), et la linéariser serait une sur-approximation trop grossière.

On ne peut donc pas se limiter à des systèmes linéaires à gauche pour l'analyse de protocoles. Pour garantir la $\mathcal{R}$-cohérence, on va s'appuyer sur une abstraction (il y a un nombre fixé de clés) et s'en servir pour définir des fonctions d'approximation qui préservent ce point. La procédure est en pratique assez technique mais l'idée intuitive peut être donnée facilement. La règle

$$\mathrm{And}(\mathrm{Crypt}(\mathrm{Message}(x),\mathrm{Key}(y)),\mathrm{Key}(y)) \to \mathrm{Message}(x)$$

va être copiée en plusieurs règles du type

$$\mathrm{And}(\mathrm{Crypt}(\mathrm{Message}(x),\mathrm{Key}(k)),\mathrm{Key}(k)) \to \mathrm{Message}(x)$$

où k n'est plus une variable mais décrit l'ensemble (fini) de clés possibles. Ensuite, des restrictions sont imposées aux fonctions d'approximation afin de garantir que de nouvelles clés ne sont pas créées, ce qui se fait par des contraintes syntaxiques. En pratique, les nouvelles règles ne sont pas explicitement crées. Grâce à ces contraintes (pour les précisions techniques voir [7]), on peut garantir la $\mathcal{R}$-cohérence à chaque étape.

Définir précisément les fonctions d'approximation utilisées serait assez lourd et peu lisible. Le lecteur intéressé pourra se référer à [7]. Cependant, il est possible d'expliquer intuitivement comment elles sont définies :

- Lorsque l'on doit normaliser une paire critique $(r\sigma, q)$, provenant d'un tuple $(\sigma, q, l \to r)$, on définit d'abord γ_i en réduisant au maximum (il faut pour cela introduire un ordre) $r\sigma$ dans $\mathcal{A}_i$. Cette réduction donne les valeurs de $\gamma_i(\sigma, q, l \to r)(p)$ pour les positions où la réduction est possible.
- Pour les règles qui ne codent pas un message du protocole, lorsqu'il n'y a plus de réduction possible, $\gamma_i(\sigma, q, l \to r)(p)$ dépend uniquement de $l \to r$ et de p : on a au plus un nouvel état par position de r.
- Pour les règles qui codent un message du protocole, lorsqu'il n'y a plus de réduction possible, $\gamma_i(\sigma, q, l \to r)(p)$ ne dépend que de $l \to r$ et de p et des agents qui sont impliqués dans la règle : on a au plus un nouvel état par position de r et par couple d'agents. Comme on utilise une abstraction avec deux agents [CLC04], le nombre d'états que l'on ajoute est borné.

Grâce à ces fonctions, qui utilisent un nombre polynomial de nouveaux états, la convergence de l'algorithme 2.2 est garantie. Illustrons sur un exemple jouet comment fonctionnent ces approximations. On considère l'automate de la figure 2.8.

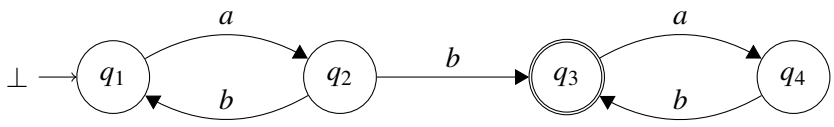


FIG. 2.8 – Exemple d'approximation automatique

En considérant le système de réécriture $\mathcal{R} = \{a(b(x)) \to b(a(x))\}$, il y a trois paires critiques : $(a(b(q_2)), q_2)$, $(a(b(q_2)), q_4)$ et $(a(b(q_4)), q_4)$. Toutes les nor-

malisations sont faites en utilisant le même état q_5. On obtient alors l'automate de la figure 2.9.

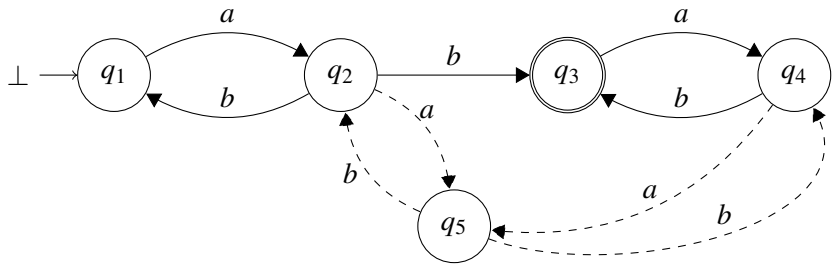


FIG. 2.9 – Exemple d'approximation automatique après une étape

L'automate de la figure 2.9 n'a qu'une paire critique : $(a(b(q_5)), q_5)$. Lors de la complétion, cette paire critique donne lieu à une normalisation qui utilise encore l'état q_5. On obtient alors l'automate de la figure 2.10 qui n'a aucune paire critique.

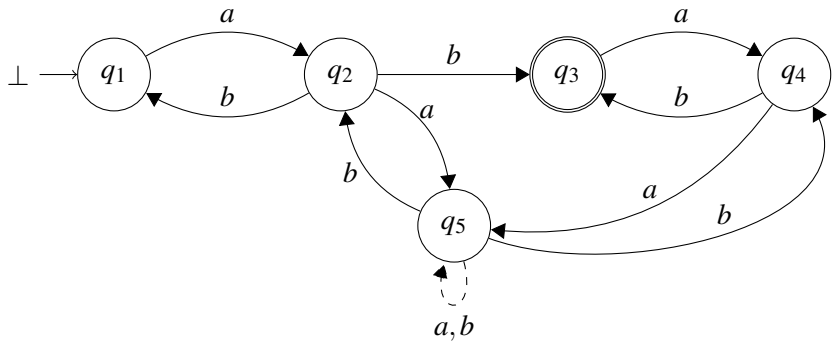


FIG. 2.10 – Exemple d'approximation automatique après deux étapes

2.2.2 Expérimentations : outil TA4SP

Les fonctions proposées ci-dessus ont été implémentées dans l'outil TA4SP, développé dans le cadre de la thèse de Yohan Boichut. Cet outil fait partie de la plate-forme AVISPA et il est disponible en ligne[6].

TA4SP utilise le format intermédiaire commun d'AVISPA qui est automatiquement calculé à partir d'une spécification haut niveau des protocoles. Ce format intermédiaire est alors traduit par TA4SP en un système de réécriture et deux automates codant la propriété de secret à vérifier ainsi que

[6] `http://www.avispa-project.org`

Protocole	Temps de calcul (s)	Diagnostique
NSPKL	4.12	SAFE
NSPK	10.26	RMU
NSSK	266.88	SAFE
Denning-Sacco shared key	24.98	SAFE
Yahalom	874.35	SAFE
Andrew Secure RPC	212.01	SAFE
Wide Mouthed Frog	30.45	SAFE
Kaochow v1	227.30	SAFE
Kaochow v2	153.00	SAFE
AAA Mobile IP	1115.00	SAFE
UMTS-AKA	2.55	SAFE
CHAPv2	18.69	SAFE
CRAM-MD5	1.14	SAFE
DHCP-Delayed-Auth	1.05	SAFE
EKE	11.76	SAFE
EKE2	1541.43	SAFE
LPD-IMSR	12.24	SAFE
h.530-fix	54687.67	??
TSIG	1140.38	SAFE
SHARE	50.41	SAFE
TMN	109.08	RMU
LPD-MSR	6.52	RMU
NSPK-XOR	1803.97	RMU
View-only-untyped	18444.57	SAFE

TAB. 2.1 – Résultats expérimentaux avec TA4SP

la connaissance initiale de l'intrus. Une sur-approximation des connaissances accessibles par l'intrus est alors calculée par complétion, en utilisant TIMBUK et les fonctions d'approximation décrites dans les sections précédentes.

Le tableau 2.1 montre les différents résultats de vérification que l'on a obtenus à l'aide de nos techniques en utilisant TA4SP. Les quatre dernières lignes font référence à des protocoles utilisant le ou-exclusif et seront commentées plus tard.

L'indication `SAFE` indique que le protocole est sûr, alors que l'indication `RMU` montre qu'il y a une attaque (en utilisant une sous-approximation). Seul le protocole `h.530-fix` a abouti à une approximation trop grossière. Les temps de calcul, en secondes, montrent l'efficacité de la méthode.

2.3 Gérer les systèmes non linéaires à gauche

2.3.1 Problématique

On a vu précédemment comment, dans le cas des protocoles, gérer la non-linéarité à gauche des systèmes de réécriture. Malheureusement, il s'agit d'une solution ad hoc qui repose sur l'hypothèse d'un nombre fini de clés (et de nonces). D'une part cela implique que les protocoles sont typés, et d'autre part cela ne permet pas de gérer des propriétés algébriques. Par exemple, si le protocole utilise l'opérateur ou-exclusif, il est nécessaire d'avoir une règle du type

$$x \text{ XOR } x \to 0,$$

et, dans ce cas, on ne peut pas limiter x à un nombre fini de valeurs.

Les exemples ci-dessous illustrent ce qui peut se passer avec des systèmes non-linéaires à gauche. Supposons que

$$\mathcal{A}_1 = (\{f,h,A,B\},\{q_1,q_2,q_f\},\{A \to q_1, A \to q_2, f(q_1,q_2) \to q_f\},\{q_f\}),$$

et que $\mathcal{R} = \{f(x,x) \to h(x)\}$. Il n'y a aucune paire critique. La procédure de complétion se réduit à l'identité, alors que $h(A) \in \mathcal{R}(L(\mathcal{A}_1)) \setminus L(\mathcal{A}_1)$. Supposons maintenant que

$$\mathcal{A}_2 = (\{f,A,B\},\{q_1,q_f\},\{A \to q_1, f(q_1,q_1) \to q_f\},\{q_f\}),$$

et que $\mathcal{R} = \{f(x,x) \to h(x)\}$. Dans ce cas il y a une paire critique $(h(q_1),q_f)$. Après complétion, l'automate obtenu sera $\mathcal{A}_2$ auquel on aura ajouté la transition $h(q_1) \to q_f$; le terme $h(A)$ est reconnu. Il faut noter que $\mathcal{A}_2$ est déterministe (voir le théorème 2.1).

Nous allons montrer dans les sections suivantes comment modifier la procédure de complétion afin de calculer des sur-approximations avec des automates non-déterministes et des systèmes non-linéaires à gauche.

2.3.2 Modification de la complétion

Quelques définitions sont nécessaires avant de montrer comment modifier la procédure de complétion.

Soit $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$. Le *linéarisé* de t est le terme de $\mathcal{T}(\mathcal{F},\mathcal{X} \times \mathbb{N}^*)$ noté $\bar{t}$ et défini par : pour toute position p de t, si $t(p) \in \mathcal{F}$ alors, $\bar{t}(p) = t(p)$ et si $t(p) \in \mathcal{X}$, alors $\bar{t}(p) = (t(p),p)$. Si l'on considère par exemple le terme $f(x,f(x,h(y))$, alors $\bar{t} = f((x,1),f((x,2\cdot 1),h((y,2\cdot 2\cdot 1))))$. On note naturellement $\overline{\mathcal{R}}$ le système $\{\bar{l} \to \bar{r} \mid l \to r \in \mathcal{R}\}$.

Si $\bar{l} \to \bar{r}$, q et σ satisfont $\bar{l}\sigma \to^*_{\mathcal{A}} q$, $\bar{r}\sigma \not\to^*_{\mathcal{A}} q$, et pour tout x de $\mathcal{V}\text{ar}(l)$,

$$\bigcap_{(x,p)\in\mathcal{V}\text{ar}(\bar{l})} L(\mathcal{A},\sigma((x,p)) \neq \emptyset,$$

alors le couple $(\bar{r}, q)$ est appelé une *NL-paire critique*.

La *NL-normalisation* (modulo une fonction d'approximation γ) est une fonction qui à un élément de $Q^{\mathcal{X}\times\mathbb{N}^*} \times Q \times \overline{\mathcal{R}}$, associe l'ensemble défini par :

$$\begin{aligned}\mathrm{NLNorm}_\gamma(\sigma, q, \bar{l} \to \bar{r}) = \{r(p)(q_1^p, \ldots, q_n^p) \to q^p \mid\ & p \in \mathcal{P}\mathrm{os}(r) \text{ et} \\ & q^\varepsilon = q \text{ et} \\ & \bar{r}(p) \text{ est d'arité } n \text{ et} \\ & q_i^p = \gamma(\sigma, q, \bar{l} \to \bar{r})(p \cdot i) \text{ si } \bar{r}(p) \notin \mathcal{X} \text{ et} \\ & q_i^p = \sigma(\bar{r}(p)) \text{ si } \bar{r}(p) \in \mathcal{X} \text{ et} \\ & q^p = \gamma(\sigma, q, \bar{l} \to \bar{r})(p) \text{ pour } p \neq \varepsilon\}\end{aligned}$$

La fonction de NL-normalisation a donc pour image un ensemble de transitions sur $\mathcal{S}$.

Le *NL-γ-complété* de $\mathcal{A} = (\mathcal{F}, Q, \Delta, Q_f)$ est l'automate, $(\mathcal{F}, Q_1, \Delta_1, Q_f)$, noté $NL - \mathcal{C}_\gamma(\mathcal{A})$ et défini par

- Δ_1 est l'union de Δ et de tous les ensembles $\mathrm{NLNorm}_\gamma(\sigma, q, \bar{l} \to \bar{r})$, pour lesquels $(\bar{r}\sigma, q)$ est une NL-paire critique (elle sont en nombre fini).
- Q_1 est l'union de Q et de tous les états de $\mathcal{S}$ apparaissant dans les transitions de Δ_1.

Si l'on reprend l'exemple de l'automate

$$\mathcal{A}_1 = (\{f, h, A, B\}, \{q_1, q_2, q_f\}, \{A \to q_1, A \to q_2, f(q_1, q_2) \to q_f\}, \{q_f\}),$$

et du système de réécriture $\mathcal{R} = \{f(x,x) \to h(x)\}$. Comme A peut se réduire dans $\mathcal{A}$ à la fois sur q_1 et sur q_2, il y a une NL-paire critique $(f(q_1, q_2), q_f)$. Le calcul du NL-complété de $\mathcal{A}_1$ ajoutera donc la transition $h(q_1) \to q_f$.

La correction de l'approche réside dans le théorème suivant.

Théorème 2.3 *Soit $\mathcal{A}$ un automate d'arbre, $\mathcal{R}$ un système de réécriture et γ une fonction d'approximation. Alors $L(\mathcal{A}) \cup \mathcal{R}(L(\mathcal{A})) \subseteq L(\mathrm{NL} - \mathcal{C}_\gamma(\mathcal{A}))$.*

Ce résultat, à comparer avec le théorème 2.1, permet de s'affranchir de l'hypothèse de cohérence. Cependant, cette nouvelle procédure de complétion est algorithmiquement plus lourde car demande des calculs d'intersections en plus (même si cela ne change pas la complexité théorique dans le cas le pire, le surcoût est sensible en pratique). Nous montrerons dans la section 2.3.3 comment atténuer ce surcoût en pratique pour les règles quadratiques (une même variable n'apparaît qu'au plus deux fois à gauche).

De plus, l'approche utilisée dans le théorème 2.3 est adaptable, dans des cas très particuliers, aux calculs de sous-approximations.

Proposition 2.4 *Soit $\mathcal{A}$ un automate d'arbre, $\mathcal{R}$ un système de réécriture et γ une fonction d'approximation satisfaisant :*

– *Pour toute règle $l \to r$ de $\mathcal{R}$, si x est une variable apparaissant au moins deux fois dans l, alors x n'apparaît pas dans r et,*
– *γ est $(\mathcal{A}, \overline{\mathcal{R}})$-exacte,*
alors $L(\mathrm{NL} - \mathcal{C}_\gamma(\mathcal{A})) \subseteq L(\mathcal{A}) \cup \mathcal{R}(L(\mathcal{A}))$.

La contrainte sur les variables est forte mais permet de gérer les règles suivantes, utiles pour l'analyse de protocoles manipulant des opérateurs algébriques, comme le ou-exclusif ou l'exponentiel :

$$x \text{ XOR } x \to 0 \quad \text{ou} \quad \mathrm{Exp}(x) * \mathrm{Exp}(\mathrm{Inv}(x)) \to 1.$$

2.3.3 Le cas des règles quadratiques

Une règle est dite quadratique à gauche si chaque variable apparaît au plus deux fois à gauche. Nous nous intéressons à ce type de règles car, en pratique, les règles non-linéaires sont souvent quadratiques. Dans le cadre des protocoles cryptographiques, elles sont toutes quadratiques à gauche.

Rappelons que si $\mathcal{A} = (Q, \Delta, F)$ est un automate d'arbre, alors le carré de $\mathcal{A}$ est l'automate $(Q \times Q, \Delta', F \times F)$ où :

$$\Delta' = \{f((q_1,q'_1),\ldots,(q_n,q'_n)) \to (q,q') \mid f(q_1,\ldots,q_n) \to q \in \Delta \text{ et } f(q'_1,\ldots,q'_n) \to q' \in \Delta\}.$$

Nous montrons ici comment effectuer efficacement les calculs pour la NL-complétion lorsque les règles sont quadratiques à gauche. Rappelons que l'on doit calculer toutes les substitutions satisfaisant des contraintes du type

$$\mathcal{L}(\mathcal{A}, \sigma(p)) \cap \mathcal{L}(\mathcal{A}, \sigma(p')) \neq \emptyset,$$

où p et p' sont des positions de règles où apparaissent une même variable. On se retrouve donc avec le problème suivant.

Règles Quadratiques
Entrée : $\mathcal{A}$ un automate fini
Question : Trouver l'ensemble des paires q, q' telles que $L(\mathcal{A}, q) \cap L(\mathcal{A}, q') \neq \emptyset$.

Une solution naïve consisterait à tester, pour tous les couples d'états, si $\mathcal{L}(\mathcal{A} \times \mathcal{A}, (\sigma(p), \sigma(p')))$ est non vide. Cependant on peut remarquer que pour tester si une substitution est compatible, on calcule toujours le carré du même automate (modulo les états finaux). Nous allons donc effectuer qu'une seule fois le calcul du carré. Par ailleurs, à chaque étape de complétion, le nouveau carré peut se recalculer à partir du précèdent, comme l'illustre la figure 2.11.

Plus formellement, si $\mathcal{A} = (\mathcal{F}, Q, \Delta, Q_F)$, si $\mathrm{NL} - \mathcal{C}_\gamma(\mathcal{A}) = (\mathcal{F}, Q', \Delta', Q_F)$ et si $\mathcal{A}^2 = (\mathcal{F}, Q_{\mathcal{A}^2}, \Delta_{\mathcal{A}^2}, Q_{F_{\mathcal{A}^2}})$, alors le carré de $\mathrm{NL} - \mathcal{C}_\gamma(\mathcal{A})$ a pour ensemble d'états

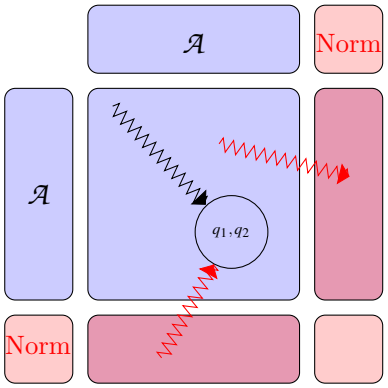


FIG. 2.11 – Calculs à la volée pour les règles quadratiques

$Q_{\mathcal{A}^2} \cup Q \times (Q' \setminus Q) \cup (Q' \setminus Q) \times Q \cup (Q' \setminus Q) \times (Q' \setminus Q)$. Non seulement le calcul du carré de $\mathrm{NL} - \mathcal{C}_\gamma(\mathcal{A})$ peut se faire à partir de $\mathcal{A}^2$ en ajoutant les transitions introduites lors de la complétion, mais en utilisant de bonnes structures de données, de nombreux calculs sur les tests du vide peuvent se factoriser.

Notons que cette optimisation pourrait s'adapter pour des règles cubiques (ou supérieures) en calculant les puissances équivalentes des automates.

2.3.4 Résultats expérimentaux

En utilisant le théorème 2.3 et les fonctions d'approximation décrites en section 2.2.1, on a pu montrer automatiquement la sûreté du protocole `ViewOnly` développé par Thomson pour la diffusion chiffrée de films. De plus, la Proposition 2.4 a permis de montrer des failles[7] (déjà connues) dans `NSPK-XOR`, `LPD-MSR` et `TMN` utilisant des opérateurs algébriques (voir bas du tableau 2.1).

Les temps de calculs indiqués sont obtenus en utilisant l'algorithme décrit dans la section 2.3.3.

2.3.5 Non-linéarité et déterminisme

Nous avons exploré une autre approche pour la non-linéarité à gauche s'appuyant sur le déterminisme : il n'est pas nécessaire d'effectuer une déterminisation complète afin d'obtenir une complétion correcte. Comme dans le cas précédent, la procédure présentée ici sera plus efficace si le système est quadratique à gauche.

Nous allons pour cela utiliser une déterminisation partielle d'un automate. Intuitivement, cette déterminisation partielle est comme une déterminisation

[7] Il faut préciser ici que notre modèle est une abstraction de ce qui se passe dans les protocoles dans la mesure où les messages ne sont pas ordonnés. Il peut donc s'agir de *fausses attaques*.

classique par sous-ensembles, sauf que l'on *éclate* chaque sous-ensemble en ensembles de taille au plus h. Formellement, cela est défini comme suit.

Définition 2.5 *Soit $\mathcal{A} = (\mathcal{F}, Q, \Delta, Q_f)$ un automate d'arbre. L'automate $\mathcal{A}^{(h)} = (\mathcal{F}, Q^{(h)}, \Delta^{(h)}, Q_f^{(h)})$, pour $h \geq 1$, est défini par :*

- $Q^{(h)} = \{H \subseteq Q \mid 1 \leq |H| \leq h\}$ *(les éléments de $Q^{(h)}$ sont notés avec un exposant (h)),*
- $Q_f^{(h)} = \{\{q\} \mid q \in Q_f\}$,
- $\Delta^{(h)} = \{f(q_1^{(h)}, \ldots, q_n^{(h)}) \to q^{(h)} \mid \forall q \in q^{(h)},\ \exists q_1, \ldots, q_n \in Q,\ \forall 1 \leq i \leq n, q_i \in q_i^{(h)}$ *et* $f(q_1, \ldots, q_n) \to q \in \Delta\}$.

Notons que $\mathcal{A}^{(h)}$ a au plus $|Q|^h$ états, et peut donc se construire en temps polynomial (si h est considéré comme une constante). Un exemple est donné dans la figure 2.12.

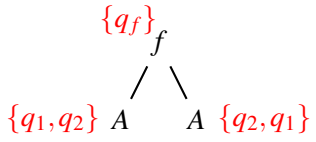


FIG. 2.12 – Un *run* de $\mathcal{A}^{(2)}$ sur $f(A,A)$

Illustrons la définition 2.5 par l'automate $\mathcal{A}$ dont l'état final est q_f et dont les transitions sont $A \to q_1$, $A \to q_2$ et $f(q_1, q_2) \to q_f$. Les états de $\mathcal{A}^{(2)}$ sont toutes les paires et les singletons sur $\{q_1, q_2, q_f\}$, et les transitions sont $A \to \{q_1\}, A \to \{q_2\}$, $A \to \{q_1, q_2\}$, $f(\{q_1\}, \{q_2\}) \to \{q_f\}$, $f(\{q_1, q_i\}, \{q_2, q_j\}) \to \{q_f\}$ pour tout i, j dans $\{1, 2, f\}$ (voir figure 2.12) $f(\{q_1, q_i\}, \{q_2, q_j\}) \to \{q_f\}$ pour tout i, j dans $\{1, 2\}$.

Tout d'abord les automates ainsi construits reconnaissent le même langage que $\mathcal{A}$.

Proposition 2.6 *Pour tout $h \geq 1$, $L(\mathcal{A}) = L(\mathcal{A}^{(h)})$.*

Le résultat principal, en lien avec la complétion est le suivant.

Théorème 2.7 *Si pour chaque règle de $\mathcal{R}$ chaque variable apparaît au plus h fois dans la partie gauche de la règle, alors $\mathcal{R}(L(\mathcal{A})) \cup L(\mathcal{A}) \subseteq L(\mathcal{C}_\gamma^{\mathcal{R}}(\mathcal{A}^{(h)}))$. Si, de plus, $\mathcal{R}$ est linéaire droit et si γ est $(\mathcal{A}, \mathcal{R})$-exacte, alors $L(\mathcal{C}_\gamma(\mathcal{A}^{(h)})) \subseteq \mathcal{R}^*(L(\mathcal{A}))$.*

Cela peut conduire à modifier l'approche par complétion par le semi-algorithme suivant.

Semi-algorithme 2.8

Nom : PointFixeParCompletionModifié
Entrées : $\mathcal{A}_0$, (γ_n), $\mathcal{R}$
Variables : i, $\mathcal{A}$, $\mathcal{B}$, h
Début
$i := 1$
$h :=$ nombre maximal de fois qu'une variable apparaît à gauche dans une règle de $\mathcal{R}$
$\mathcal{A} := \mathcal{A}_0^{(h)}$
$\mathcal{B} := \mathcal{C}_{\gamma_0}(\mathcal{A})$
TantQue $(L(\mathcal{A}) \neq L(\mathcal{B}))$
$\mathcal{A} := \mathcal{B}$
$\mathcal{B} := \mathcal{C}_{\gamma_i}(\mathcal{B}^{(h)})$
$i := i + 1$
FinTanQue
Retourner $\mathcal{B}$.
Fin

Plusieurs remarques sont à faire sur cette approche :
- Le test de point fixe doit se faire sur les langages, non sur les automates. En effet, ajouter le calcul de déterminisation partielle empêche tout espoir de convergence syntaxique.
- Un avantage de cette approche, par rapport à la NL-complétion, est de pouvoir faire du calcul exact (sous-approximation) dans le cadre général.
- Tester le point fixe sur les langages ajoute un surcoût important algorithmique.
- Même pour les règles quadratiques, la taille des automates grossit exponentiellement avec le nombre d'étapes de complétion. La procédure ne peut fonctionner que sur des exemples avec peu d'étapes.
- Pour le moment, l'approche n'a pas été implémentée de façon satisfaisante, permettant d'apprécier le gain sur des exemples concrets.

2.4 Raffinement d'approximations

Rappelons que l'on s'intéresse au problème suivant.

Accessibilité Régulière par réécriture
Entrée : Deux automates d'arbres $\mathcal{A}_0$ et $\mathcal{A}_{\text{prop}}$, un système de réécriture $\mathcal{R}$
Question : A-t-on $\mathcal{R}^*(L(\mathcal{A}_0)) \cap L(\mathcal{A}_{\text{prop}}) = \emptyset$?

Les méthodes proposées ci-avant pour automatiser la complétion sont certes automatiques mais non adaptatives : que faire si l'approximation est

trop grossière ? Comment en calculer une plus fine ?

Pour répondre à ces questions, nous nous sommes appuyés sur le concept de *Counter-Example Guided Abstraction Reffinment* [CGJ+00] : si l'approximation de l'ensemble des termes accessibles a une intersection non vide avec le langage de la propriété, on prend (un ou plusieurs) éléments de cette intersection et l'on modifie l'approximation pour essayer de décider si ces éléments sont ou non accessibles. Cela peut permettre de répondre qu'il y a effectivement un terme de $L(\mathcal{A}_{\text{prop}})$ accessible ou qu'il n'y en a pas. Cela peut aussi rester non-concluant.

2.4.1 Raffinement d'approximations pour les systèmes linéaires

Nous présentons ici une méthode de raffinement d'approximations guidé par un contre-exemple adaptée à la méthode de complétion, pour les systèmes de réécriture linéaires. Pour cela, on va modifier le semi-algorithme 2.2 :

(1) En ajoutant, à chaque étape, un test du vide de l'intersection entre le langage reconnu par l'automate calculé à chaque itération et le langage d'un automate $\mathcal{A}_{\text{prop}}$ codant *les mauvais termes*,

(2) En stockant dans une liste tous les automates calculés.

Rappelons que l'on cherche à savoir si $L(\mathcal{A}_{\text{prop}}) \cap \mathcal{R}^*(L(\mathcal{A}_0)) = \emptyset$. Le semi-algorithme *Raffiner* sera décrit plus loin.

Semi-algorithme 2.9

Nom : PointFixeRaffine
Entrées : $\mathcal{A}_0$, (γ_n), $\mathcal{R}$, $\mathcal{A}_{\text{prop}}$
Variables : i, $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_k, \ldots$
Début
  $i := 1$
  $\mathcal{A}_1 := \mathcal{C}_{\gamma_0}(\mathcal{A}_0)$
  TantQue ($\mathcal{A}_i \neq \mathcal{A}_{i-1}$ **et** $L(\mathcal{A}_{\text{prop}}) \cap L(\mathcal{A}_i) = \emptyset$)
    $\mathcal{A}_{i+1} := \mathcal{C}_{\gamma_i}(\mathcal{A}_i)$
    $i := i + 1$
  FinTanQue
  Si $\mathcal{A}_i \neq \mathcal{A}_{i-1}$
  Alors Raffiner ($\mathcal{A}_0, \ldots, \mathcal{A}_i$, (γ_n), $\mathcal{A}_{\text{prop}}$)
  Sinon Retourner vrai
  FinSi
Fin

Dans le semi-algorithme 2.9, on arrête la procédure de complétion soit parce qu'on a atteint un point fixe, soit parce que l'intersection entre $L(\mathcal{A}_i)$ et $L(\mathcal{A}_{\text{prop}})$

est non vide, ce qui répond à la question. Si $\mathcal{A}_i \neq \mathcal{A}_{i-1}$ à la sortie de la boucle **TantQue**, alors $\mathcal{R}^*(L(\mathcal{A}_0) \cap L(\mathcal{A}_{\text{prop}})$ est vide. Sinon, on ne peut pas conclure à ce stade et l'on va raffiner l'approximation.

Supposons que l'on soit dans ce dernier cas : $\mathcal{A}_i \neq \mathcal{A}_{i-1}$. On peut alors distinguer deux situations (rappelons que la notation $C^{\mathcal{R}}_{\text{exact}}$ est définie page 29) :

(a) Soit $L(\mathcal{C}^{\mathcal{R}}_{\text{exact}}(\mathcal{A}_{i-1})) \cap L(\mathcal{A}_{\text{prop}}) \neq \emptyset$,

(b) Soit $L(\mathcal{C}^{\mathcal{R}}_{\text{exact}}(\mathcal{A}_{i-1})) \cap L(\mathcal{A}_{\text{prop}}) = \emptyset$,

Si l'on est dans le cas (a), quelque soit la fonction d'approximation utilisée sur $\mathcal{A}_{i-1}$, le résultat sera un automate reconnaissant des termes dans $L(\mathcal{A}_{\text{prop}})$. Raffiner γ_{i-1} ne sera donc jamais concluant. On va alors, par des calculs *en arrière*, chercher la dernière étape de complétion qui est potentiellement raffinable. On utilise pour cela l'algorithme suivant, illustré sur la figure 2.13, où $\mathcal{R}^{-1} = \{r \to l \mid l \to r \in \mathcal{R}\}$.

Algorithme 2.10

Nom : RechercheArrière
Entrées : $\mathcal{A}_0, \mathcal{A}_1, \ldots, \mathcal{A}_k$, $\mathcal{A}_{\text{prop}}$, $\mathcal{R}$, $\mathcal{B}$, $i \leq k$
Variables : j
Début
  $j := i - 1$
  $\mathcal{B} := \mathcal{A}_{\text{prop}}$
  TantQue ($j \geq 0$ **et** $L(\mathcal{B}) \cap L(\mathcal{C}^{\mathcal{R}}_{\text{exact}}(\mathcal{A}_j)) \neq \emptyset$)
    $\mathcal{B} := \mathcal{C}^{\mathcal{R}^{-1}}_{\text{exact}}(\mathcal{B})$
    $j := j - 1$
  FinTanQue
  Retourner j
Fin

Deux cas d'arrêt de la boucle **TantQue** se présentent. Soit la valeur de j est négative, soit on a $L(\mathcal{B}) \cap L(\mathcal{C}^{\mathcal{R}}_{\text{exact}}(\mathcal{A}_j)) = \emptyset$. Si la valeur de j est négative, cela signifie en particulier que $L(\mathcal{B}) \cap L(\mathcal{C}^{\mathcal{R}}_{\text{exact}}(\mathcal{A}_0)) \neq \emptyset$. Or, comme $L(\mathcal{B}) \subseteq \mathcal{R}^{-1^*}(L(\mathcal{A}_{\text{prop}}))$ (théorème 2.1), on sait alors que $\mathcal{R}^*(L(\mathcal{A}_0)) \cap L(\mathcal{A}_{\text{prop}}) \neq \emptyset$, ce qui répond à la question générale.

Dans le cas où l'algorithme retourne un j positif, son déroulement peut s'illustrer par la figure 2.13. Dans ce cas, on va raffiner l'approximation en procédant comme suit : on cherche quelles transitions de $\mathcal{A}_j$ du produit de $\mathcal{A}_j$ par $\mathcal{A}^{i-j}_{\text{prop}}$ permettent d'atteindre un état final et en inspectant quelles paires critiques sont associées à ces transitions : pour ces paires, la nouvelle fonction d'approximation[8] γ'_i utilisera des nouveaux états ; pour les autres elle

[8] En pratique, la procédure est un peu plus complexe.

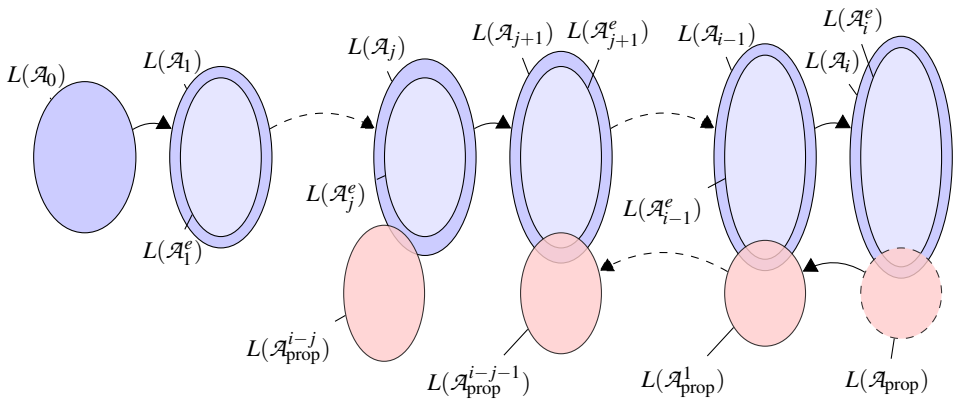


FIG. 2.13 – Raffinement d'approximations. Les automates $\mathcal{A}_k$ sont les automates obtenus par complétion. Les automates $\mathcal{A}_k^e$ sont les automates obtenus par une complétion exacte à partir de $\mathcal{A}_{k-1}$. Les automates $\mathcal{A}_{\text{prop}}^k$ sont obtenus à partir de $\mathcal{A}_{\text{prop}}$ en effectuant k étapes de complétion exactes en utilisant $\mathcal{R}^{-1}$.

coïncidera avec γ_i.
Cela se traduit dans le semi-algorithme suivant.

Semi-algorithme 2.11

Nom : Raffiner
Entrées : $\mathcal{A}_0, \mathcal{A}_1, \ldots, \mathcal{A}_k$, $\mathcal{A}_{\text{prop}}$, $\mathcal{R}$, $\mathcal{B}$, $i \leq k$
Variables : j
Début
  j :=RechercheArrière($\mathcal{A}_0, \mathcal{A}_1, \ldots, \mathcal{A}_k$, $\mathcal{A}_{\text{prop}}$, $\mathcal{R}$, $\mathcal{B}$, i)
  Si $j == -1$
  Alors Retourner faux
  Sinon
    Calculer (γ'_n) nouvelles fonctions d'approximation
    Retourner PointFixeRaffine($\mathcal{A}_j$, (γ'_n), $\mathcal{R}$, $\mathcal{A}_{\text{prop}}^k$)
  FinSi

On obtient ainsi un semi-algorithme possédant les propriétés suivantes.

Proposition 2.12 *Le semi-algorithme 2.9 est correct, c'est-à-dire que s'il répond vrai, alors* $\mathcal{R}^*(L(\mathcal{A}_0)) \cap L(\mathcal{A}_{\text{prop}})$ *est vide et s'il répond faux, alors* $\mathcal{R}^*(L(\mathcal{A}_0)) \cap L(\mathcal{A}_{\text{prop}})$ *est non vide. De plus, si* $\mathcal{R}^*(L(\mathcal{A}_0)) \cap L(\mathcal{A}_{\text{prop}})$ *est non vide, alors il termine.*

Le semi-algorithme 2.11 peut ne pas terminer, comme sur l'exemple de la section 2.4.2. Nous avons testé ce semi-algorithme sur un exemple simple de protocole entre deux processus. Tout d'abord avec une mauvaise spécification du protocole, une erreur a été trouvée. Ensuite, sur une bonne spécification du protocole et en utilisant une fonction d'approximation très simple, la correction a été prouvée avec une étape de raffinement.

2.4.2 Limite théorique

Soit $I = \{g(h^k(A)) \mid k \geq 0\}$, $\mathcal{R} = \{g(x) \to g(f(x,x)), f(h(x),h(y)) \to f(x,y), f(h(x),A) \to C, f(A,h(x)) \to C\}$, et l'on se pose la question suivante : $\mathcal{R}^*(I) \cap \{C\} = \emptyset$?

On peut facilement montrer que $\mathcal{R}^*(I) = I \cup \{g(f(h^k(A),h^k(A))) \mid k \geq 0\}$, et donc que $C \notin \mathcal{R}^*(I)$.

Cependant, pour tout langage régulier K tel que $\mathcal{R}^*(I) \subseteq K$ et $K \subseteq R(K)$ on peut montrer que $C \in K$. Il en résulte que quelque soit la technique utilisée, quelque soit la sur-approximation obtenue, on ne pourra jamais conclure par cette approche que $C \notin \mathcal{R}^*(I)$.

Cet exemple montre une limite de l'approche par approximation régulière et suggère que des modèles plus précis doivent être envisagés pour certaines applications.

2.5 Vérifier des propriétés temporelles

Nous étudions dans cette section comment utiliser l'analyse d'accessibilité par approximation afin de (semi)-vérifier des propriétés un peu plus complexes. Il s'agit ici de propriétés temporelles simples utiles dans un cadre d'une analyse statique de programmes Java.

2.5.1 Contexte scientifique

Réécriture et logique temporelle

La vérification de propriétés temporelles sur des modèles de réécriture a été abordée dans le cadre des logiques de réécriture [Mes92, EM07, Mes08, Mes07]. Ces résultats proposent des procédures effectives en imposant des hypothèses restrictives.

Le modèle que nous utilisons est plus simple que le modèle général des logiques de réécriture car nous ne nous intéressons qu'à des propriétés sur des séquences de transitions, et jamais sur des propriétés d'état. Le modèle, assez naturel, est celui d'un système de transition infini généré par un système de réécriture.

Soit $\mathcal{R}$ un système de réécriture et L_0 un ensemble de termes. On note $G(L_0,\mathcal{R})$ le graphe étiqueté par $\mathcal{R}$: $G(L_0,\mathcal{R}) = (\mathcal{T}(\mathcal{F}),L_0,\Delta)$ où $\Delta = \{t_i \stackrel{l\to r}{\to} t_j \mid l \to r \in \mathcal{R}$ et $t_j \in \{l \to r\}(t_i)\}$. Un chemin π dans ce graphe est *complet* s'il est infini ou si son dernier état ne possède aucune transition sortante.

Les formules LTL (Linear Temporal Logic) [Pnu77] sur $\mathcal{R}$ sont inductivement définies par : $\mathcal{R}_0 \subseteq \mathcal{R}$ est une formule LTL, et si φ et ψ sont des formules LTL sur $\mathcal{R}$, alors $\top$, $\neg\varphi$, $(\varphi \vee \psi)$, $\circ\varphi$ et $\varphi\ U\ \psi$ sont aussi des formules LTL sur $\mathcal{R}$. Classiquement, on définit les opérateurs suivants : $\Box\varphi = \neg(\top U\ \neg\varphi)$, $(\varphi \wedge \psi) = \neg(\neg\varphi \vee \neg\psi)$ et $\varphi \Rightarrow \psi = (\neg\varphi \vee \psi)$.

Soit w un mot (fini ou infini) sur $\mathcal{R}$ (considéré comme un alphabet). La satisfaction d'une formule LTL par un mot w à la position i, notée $(w,i) \models \varphi$, est définie par :

$(w,i) \models \top$	ssi $w(i)$ existe,
$(w,i) \models \mathcal{R}_0$, avec $\mathcal{R}_0 \subseteq \mathcal{R}$	ssi $w(i)$ existe et $w(i) \in \mathcal{R}_0$,
$(w,i) \models \neg\varphi$	ssi $(w,i) \not\models \varphi$,
$(w,i) \models (\varphi_1 \vee \varphi_2)$	ssi $(w,i) \models \varphi_1$ ou $(w,i) \models \varphi_2$,
$(w,i) \models \circ\varphi$	ssi $(w,i+1) \models \varphi$,
$(w,i) \models (\varphi_1 U\ \varphi_2)$	ssi il existe $j \geq i$ tel que $(w,i) \models \varphi_2$ et pour tout $i \leq k < j$, $(w,k) \models \varphi_1$.

On dit que w est un modèle de φ si $(w,1) \models \varphi$. Un graphe $G(L_0,\mathcal{R})$ satisfait une formule LTL φ, ce que l'on note $G \models \varphi$, si et seulement si l'étiquette de chaque chemin complet de $G(L_0,\mathcal{R})$ satisfait φ.

Automates d'arbre avec contraintes globales

Les TAGED, *Tree Automata with Gobal Equality and Disequality constraints*, ont été introduits dans [FTT08] afin d'enrichir le modèle d'automate d'arbre[9] pour introduire des comparaisons de branches dans les termes reconnus.

Dans le présent document, seuls les TAGED avec contraintes d'égalité, appelés *TAGED positifs*, seront utilisés et donc définis. Un TAGED positif est un tuple $\mathcal{A} = (\mathcal{F},Q,\Delta,E,Q_f)$, où $(\mathcal{F},Q,\Delta,Q_f)$ est un automate d'arbre sur $\mathcal{F}$, et $E \subseteq Q \times Q$ est une relation réflexive et symétrique sur un sous-ensemble de Q. Un *calcul* réussi d'un TAGED positif sur un terme t est un *calcul* ρ réussi de l'automate d'arbre sous-jacent sur t qui satisfait : pour toutes positions p_1, $p_2 \in \mathcal{P}os(t)$, si $(\rho(p_1),\rho(p_2)) \in E$ alors $t_{|_{p_1}} = t_{|_{p_2}}$.

Pour les TAGED positifs, le problème du vide est dans EXPTIME [FTT08, Theorem 1], les problèmes de l'universalité et de l'inclusion sont tous deux indécidables [FTT08, Proposition 5].

[9] De nombreux travaux ont été fait dans ce sens, car l'utilisation d'automate d'arbres pour la modélisation demande souvent de comparer des branches entres elle. Pour plus de renseignements voir le chapitre 4 de [CDG+02].

2.5.2 Trois motifs de propriétés

Les trois motifs de propriétés que nous allons étudier ont été introduits pour de l'analyse de code Java de téléphonie mobile[10].

– La formule $\Box(\mathcal{R}_1 \Rightarrow \circ\mathcal{R}_2)$ code que si un terme est accessible en utilisant une règle de $\mathcal{R}_1$ alors on peut le réécrire en utilisant une règle de $\mathcal{R}_2$ et uniquement une règle de $\mathcal{R}_2$ (voir figure 2.14). Dans le cadre de l'analyse statique, on souhaite spécifier que si une méthode m_1 est appelée, alors la méthode m_2 est appelée juste après. Par exemple, si une procédure d'authentification est lancée, elle doit être suivie soit d'une authentification réussie, soit de l'annulation de cette authentification.

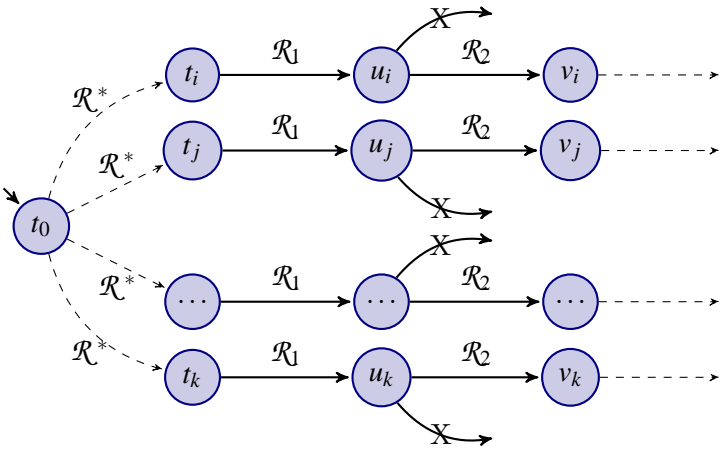


FIG. 2.14 – Modèle de $\Box(\mathcal{R}_1 \Rightarrow \circ\mathcal{R}_2)$

– La formule $\neg\mathcal{R}_2 \wedge \Box(\circ\mathcal{R}_2 \Rightarrow \mathcal{R}_1)$ est la duale de la précédente : si on utilise une règle de $\mathcal{R}_2$ alors on a nécessairement utilisé une règle de $\mathcal{R}_1$ juste avant (voir figure 2.15). On souhaite pouvoir spécifier une propriété du type : si un SMS est envoyé, alors l'accord pour cet envoi vient juste d'être donné.

– La formule $\Box(\mathcal{R}_1 \Rightarrow \Box\neg\mathcal{R}_2)$ code que si une règle de $\mathcal{R}_1$ est utilisée, alors aucune règle de $\mathcal{R}_2$ ne pourra être utilisée ensuite (voir figure 2.16). On souhaite spécifier ainsi des propriétés comme : si l'application accède à des données sensibles (carnet d'adresse par exemple), alors l'application ne fera pas d'accès Web ensuite.

2.5.3 Une approche par automates avec contraintes globales

Afin de savoir si un graphe généré par un système de réécriture et un ensemble d'états initiaux satisfont une des propriétés temporelles définies précé-

[10] Dans le cadre du projet RAVAJ, `http://www.irisa.fr/lande/genet/RAVAJ`

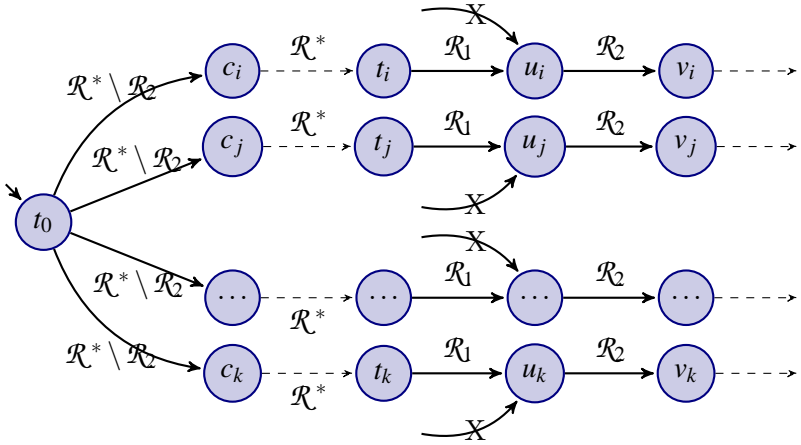


FIG. 2.15 – Modèle de $\neg\mathcal{R}_2 \wedge \Box(\circ\mathcal{R}_2 \Rightarrow \mathcal{R}_1)$

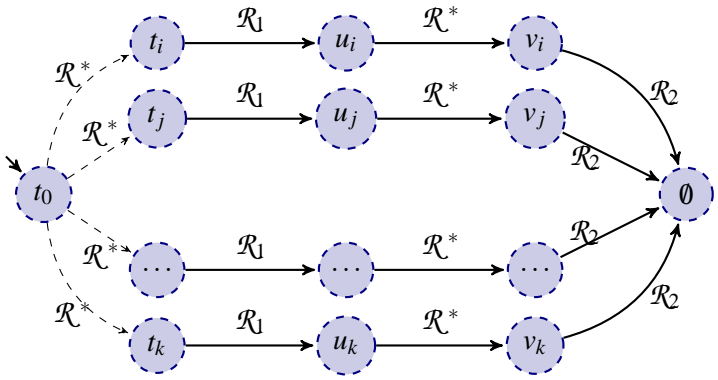


FIG. 2.16 – Modèle de $\Box(\mathcal{R}_1 \Rightarrow \Box\neg\mathcal{R}_2)$

demment, la première étape consiste à transformer ce problème de satisfaction en équation sur des langages. Ce qui est fait grâce à la proposition suivante.

Proposition 2.13 *Soit $\mathcal{R}$ un système de réécriture, $\mathcal{R}_1, \mathcal{R}_2 \subseteq \mathcal{R}$ et L_0 un langage d'arbre. On a*

- $G(L_0, \mathcal{R}) \models \Box(\mathcal{R}_1 \Rightarrow \circ\mathcal{R}_2)$ *si et seulement si* $(\mathcal{R} \setminus \mathcal{R}_2)(\mathcal{R}_1(\mathcal{R}^*(L_0))) = \emptyset$ *et* $\mathcal{R}_1(\mathcal{R}^*(L_0)) \cap \mathcal{R}_2^{-1}(\mathcal{T}(\mathcal{F})) = \mathcal{R}_1(\mathcal{R}^*(L_0))$.
- $G(L_0, \mathcal{R}) \models \neg\mathcal{R}_2 \wedge \Box(\circ\mathcal{R}_2 \Rightarrow \mathcal{R}_1)$ *si et seulement si* $\mathcal{R}_2((\mathcal{R} \setminus \mathcal{R}_1)(\mathcal{R}^*(L_0))) = \emptyset$ *et* $\mathcal{R}_2(L_0) = \emptyset$.
- $G(L_0, \mathcal{R}) \models \Box(\mathcal{R}_1 \Rightarrow \Box\neg\mathcal{R}_2)$ *si et seulement si* $\mathcal{R}_2(\mathcal{R}^*(\mathcal{R}_1(\mathcal{R}^*(L_0)))) = \emptyset$.

Ces différentes équations seront abordées en utilisant les résultats suivants.

Proposition 2.14 *Soit $\mathcal{R}$ un système de réécriture. On peut calculer en temps polynomial un TAGED positif reconnaissant $\mathcal{R}^{-1}(\mathcal{T}(\mathcal{F}))$.*

Considérons par exemple $\mathcal{F} = \{\bot, h, f\}$ où $\bot$ est une constante, h est d'arité 1 et f d'arité 2. Le langage $\{f(x,x) \to h(x)\}^{-1}(\mathcal{T}(\mathcal{F}))$ est accepté par le TAGED positif $\mathcal{A}_l = (Q_l, E_l, F_l, \Delta_l)$ où

- $Q_l = \{q_\varepsilon, q_1, q_2\} \cup \{q_x, q_x^a\} \cup \{q^a\}$,
- $E_l = \{(q_x, q_x)\}$,
- $\Delta_l = \Delta_1 \cup \Delta_2$ avec $\Delta_1 = \{f(q_x, q_x) \to q_\varepsilon\} \cup \{f(q_x^a, q_x^a) \to q_x^a, \bot \to q_x^a, h(q_x^a) \to q_x^a\} \cup \{f(q_x^a, q_x^a) \to q_x, \bot \to q_x, h(q_x) \to q_x\}$ et $\Delta_2 = \{f(q^a, q^a) \to q^a, \bot \to q_a, h(q_a) \to q_a\} \cup \{f(q^a, q_\varepsilon) \to q_\varepsilon, f(q_\varepsilon, q^a) \to q_\varepsilon, h(q_\varepsilon) \to q_\varepsilon\}$
- $F_l = \{q_\varepsilon\}$.

Proposition 2.15 *Soit $\mathcal{A}$ un automate d'arbre et $\mathcal{R}$ un système de réécriture. Le langage $\mathcal{R}(L(\mathcal{A}))$ est reconnu par un TAGED positif.*

Enfin,

Proposition 2.16 *Soit $\mathcal{A}$ un TAGED positif et $\mathcal{R}$ un système de réécriture. Décider si $\mathcal{R}(L(\mathcal{A}))$ est vide est dans EXPTIME.*

2.5.4 Algorithmes

Pour semi-décider les propriétés temporelles décrites, nous avons besoin des procédures suivantes.

- Approx($\mathcal{A}$,$\mathcal{R}$), où $\mathcal{A}$ est un automate d'arbre et $\mathcal{R}$ un système de réécriture, retourne un automate d'arbre $\mathcal{B}$ tel que $\mathcal{R}^*(L(\mathcal{A})) \subseteq L(\mathcal{B})$. Cela est fait, par exemple, en utilisant les procédures de complétion décrites précédemment dans ce chapitre.
- ta($\mathcal{A}$), où $\mathcal{A}$ est un TAGED positif, retourne l'automate d'arbre sous-jacent à $\mathcal{A}$.
- OneStep($\mathcal{A}$,$\mathcal{R}$), où $\mathcal{A}$ est un automate d'arbre et $\mathcal{R}$ est un système de réécriture, retourne un TAGED positif $\mathcal{B}$ reconnaissant $\mathcal{R}(L(\mathcal{A})$ (proposition 2.15).
- Backward($\mathcal{R}$), où $\mathcal{R}$ est un système de réécriture, retourne le TAGED positif $\mathcal{B}$ reconnaissant $\mathcal{R}^{-1}(\mathcal{T}(\mathcal{F}))$ (proposition 2.14).
- IsEmpty($\mathcal{A}$,$\mathcal{R}$), où $\mathcal{A}$ est un TAGED positif et $\mathcal{R}$ un système de réécriture, retourne vraie si $\mathcal{R}(L(\mathcal{A}))$ est vide et faux, sinon.

En utilisant la proposition 2.13 et les procédures ci-dessus, on peut construire des semi-algorithmes pour le problème de la satisfaction des motifs.

Si l'algorithme 2.17 retourne vrai, alors $G(L(\mathcal{A}_0), \mathcal{R}) \models \Box(\mathcal{R}_1 \Rightarrow \circ\mathcal{R}_2)$. S'il retourne faux, on ne peut pas conclure. Il faut noter que l'on a besoin dans cet algorithme que $\mathcal{R}_2$ soit linéaire droit afin que l'inclusion $L(\mathcal{B}) \subseteq$ Backward($\mathcal{R}_2$)

soit décidable (en effet dans ce cas le TAGED positif produit par le calcul $\mathsf{Backward}(\mathcal{R}_2)$ est un automate d'arbre classique).

Algorithme 2.17

Nom : Formule1
Entrées : $\mathcal{A}_0$, $\mathcal{R}$, $\mathcal{R}_1$, $\mathcal{R}_2$ ($\mathcal{R}_2$ doit être linéaire à gauche)
Variables : $\mathcal{B}$
Début
  $\mathcal{B} := \mathsf{Approx}(\mathcal{A}_0,\mathcal{R})$
  $\mathcal{B} := \mathsf{OneStep}(\mathcal{B},\mathcal{R}_1)$
  Si $\mathsf{IsEmpty}(\mathcal{B},\mathcal{R}_1)==$ faux
  Alors Retourner faux
  Sinon
    Si $L(\mathcal{B}) \subseteq \mathsf{Backward}(\mathcal{R}_2)$
    Alors Retourner vrai
    Sinon Retourner faux
    FinSi
  FinSi
Fin

Si l'algorithme 2.18 retourne vrai, alors $G(L(\mathcal{A}_0),\mathcal{R}) \models \Box(\circ\mathcal{R}_2 \Rightarrow \mathcal{R}_1)$; s'il retourne faux, on ne peut pas conclure.

Algorithme 2.18

Nom : Formule2
Entrées : $\mathcal{A}_0$, $\mathcal{R}$, $\mathcal{R}_1$, $\mathcal{R}_2$
Variables : $\mathcal{B}$
Début
  Si $\mathsf{IsEmpty}(\mathcal{A},\mathcal{R}_2)==$ faux
  Alors Retourner faux
  Sinon
    $\mathcal{B} := \mathsf{Approx}(\mathcal{A}_0,\mathcal{R})$
    $\mathcal{B} := \mathsf{OneStep}(\mathcal{B},\mathcal{R} \setminus \mathcal{R}_1)$
    Si $\mathsf{IsEmpty}(\mathcal{B},\mathcal{R}_2)==$ vrai
    Alors Retourner vrai
    Sinon Retourner faux
    FinSi
  FinSi
Fin

Enfin, l'algorithme 2.19 retourne vrai, alors $G(L(\mathcal{A}_0), \mathcal{R}) \models \Box(\mathcal{R}_1 \Rightarrow \Box\neg\mathcal{R}_2)$, sinon on ne peut pas conclure.

Algorithme 2.19

Nom : Formule3
Entrées : $\mathcal{A}_0$, $\mathcal{R}$, $\mathcal{R}_1$, $\mathcal{R}_2$
Variables : $\mathcal{B}$
Début
$\mathcal{B} :=$ Approx($\mathcal{A}_0$,$\mathcal{R}$)
$\mathcal{B} :=$ OneStep($\mathcal{B}$,$\mathcal{R}_1$)
$\mathcal{B} :=$ ta($\mathcal{B}$)
$\mathcal{B} :=$ Approx($\mathcal{B}$,$\mathcal{R}$)
Si IsEmpty($\mathcal{A}$,$\mathcal{R}_2$)== faux
Alors Retourner faux
Sinon Retourner vrai
FinSi
Fin

Pour le moment, ces algorithmes n'ont pas été implémentés et n'ont donc pas été expérimentés sur des cas pratiques. Il faut souligner sur ce point qu'il n'existe pas, à notre connaissance, de bibliothèque gérant les TAGED positifs. De plus, la taille des automates mis en jeu impose une programmation soignée.

2.6 Conclusion et perspectives

En partant d'une procédure non-automatique de complétion, nous avons montré comment obtenir une procédure efficace de vérification, notamment dans le cadre de la vérification de protocoles cryptographiques. Nous avons pour cela développé des fonctions d'approximation générées automatiquement, nous avons montré comment gérer les spécifications utilisant des systèmes non-linéaire à gauche et aussi exploré comment utiliser les procédures d'approximation pour vérifier des propriétés plus complexes. Enfin, nous avons proposé pour les systèmes linéaire une procédure de raffinement d'approximations.

Dans le cadre des protocoles de sécurité, l'analyse utilisant des approximations régulières s'avère très efficace pour la vérification de propriété de secret. Les premiers travaux faits dans le cadre de l'analyse statique exhibent de nouvelles difficultés : les spécifications (automates et TRS) ont des tailles qui les rendent difficiles à manipuler.

Sur un plan théorique, de nombreuses questions restent à résoudre :

- De nombreux systèmes manipulent des variables, notamment entières, et leur modélisation par un système de réécriture conduit à des systèmes de grande taille : les entiers sont codés dans l'arithmétique de Peano, c'est-à-dire en unaire. Ce type de codage fait exploser la taille des termes mis en jeu par rapport à des variables codées en binaire (par exemple). Il serait intéressant de coupler la technique de complétion avec les techniques d'accélération d'automates à compteurs, qu'elles soient exactes comme dans [BFLP08] ou obtenues par approximation comme par exemple dans [HMG06].
- De façon duale, la technique de complétion pourrait être utile pour surapproximer l'ensemble des états accessibles d'un système sur les mots ou manipulant des compteurs.
- Des travaux récents [dT06, GMSZ08, LS07, JR08], motivés par des applications en théorie des bases de données, s'intéressent à la réécriture de *hedge*, qui sont des termes d'arité non borné. Dans ce cadre, les techniques de complétion sont à redéfinir et à évaluer.
- Un problème général de l'analyse par approximation est l'évaluation des techniques mises en œuvre qui ne peut se faire que par expérimentation : on ne dispose pas de mesure pour comparer deux techniques d'approximation théoriquement (à part l'inclusion qui est trop grossière). Mettre en place des indicateurs théoriques aiderait au développement de ces techniques.

D'un point de vue plus *pratique*, de nombreuses pistent restent à explorer.

- Si la complétion donne de bons résultats sur l'analyse de protocole, son utilisation dans le cadre de l'analyse statique de programme, comme dans [BGJR07], est beaucoup plus délicat : les automates et les systèmes de réécriture ont des tailles importantes (plusieurs centaines de règles), et la recherche de paires critiques en devient complexe. Développer des implantations efficaces, des algorithmes performants ainsi que des structures de données dédiées est probablement un passage obligé pour obtenir des résultats sur des systèmes plus grands.
- Comme mentionné ci-dessus, on ne dispose pas de moyen théorique pour comparer des techniques d'approximation. Une approche non théorique serait de proposer une base de données d'exemples variés de problèmes afin de tester les différentes approches.
- La procédure de complétion a la particularité d'être facilement parallélisable : on peut découper un système de réécriture imposant en morceaux et chercher des points fixes en distribuant les calculs. La façon de procéder influera sur le point fixe obtenu, mais les gains d'efficacité (en temps de calcul) pourraient être significatifs.

Chapitre 3

Accélération de relations de semi-commutation

Sommaire

3.1 Les relations de semi-commutation 54
3.1.1 Définitions utiles 54
3.1.2 Motivations et résultats connus 56
3.2 Calcul du R-mélange 57
3.2.1 Calcul sur les automates 57
3.2.2 Calcul sur APC - Résultats expérimentaux 58
3.3 PolCom et autres classes 61
3.3.1 PolCom 61
3.3.2 Pour aller plus loin 62
3.3.3 Exemple : un contrôleur d'ascenseurs 62
3.4 Relations de semi-commutation sur les arbres . . 65
3.4.1 Automates d'arbre partiellement ordonnés et formules du premier ordre. 65
3.4.2 Automates d'arbre partiellement ordonnés et semi-commutation 67
3.5 Conclusion et perspectives 67

Les publications liées à ce chapitre sont [13, 14, 15, 20].

Nous nous intéressons ici à l'analyse d'accessibilité régulière sur les mots, pour une classe particulière de relations appelées *relations de semi-commutation*.

Section 3.1. Dans cette section, nous posons les définitions utiles (section 3.1.1) et présentons les principaux résultats connus (section 3.1.2).

Section 3.2. Nous montrons dans cette section le résultat technique sur lequel s'appuiera nos contributions : le produit de R-mélange de deux langages réguliers est régulier et se calcule en temps polynomial si les deux langages sont donnés par des automates finis. Ce résultat a été publié dans [15].

Section 3.2.2. Nous exposons dans cette section le premier résultat découlant de la construction sur le produit de R-mélange en obtenant, comme conséquence directe, une nouvelle preuve d'un résultat prouvé dans [GRS04, BMT07]. Des résultats expérimentaux montrent l'efficacité de notre approche par rapport à celle de [BMT07].

Section 3.3. Le résultat théorique principal est donné ici, où nous étendons les résultats de [GRS04, BMT07] à une classe plus grande : nous montrons que la classe PolCom (clôture polynomiale des langages réguliers commutatifs), est close par semi-commutation. Nous montrons aussi comment le résultat sur le calcul du R-mélange peut s'utiliser dans un cadre plus large de model-checking régulier. Ces résultats ont été publiés dans [13, 14, 15].

Section 3.4. Nous explorons dans cette section la généralisation des résultats de [GRS04, BMT07] aux langages d'arbre. Les résultats sont négatifs : des langages d'arbre réguliers très simples admettent une clôture par semi-commutation non régulière. Ces résultats ont fait l'objet de la publication [20].

3.1 Les relations de semi-commutation

3.1.1 Définitions utiles

Dans tout ce chapitre, A désigne un alphabet fini. On rappelle que si u est un mot de A^*, $\alpha(u)$ désigne l'ensemble des lettres y apparaissant, c'est-à-dire $\{u(i) \mid 1 \leq i \leq |u|\}$.

Relations de semi-commutation

Une relation de *semi-commutation* R, est un sous-ensemble de $A \times A$. Une semi-commutation symétrique est appelée une *commutation partielle*.

Si u est un mot de A^*, on note $R(u)$ l'ensemble des mots v tels qu'il existe $x, y \in A^*$, $(a,b) \in R$ satisfaisant $u = xaby$ et $v = xbay$. La définition s'étend na-

turellement aux langages par :

$$R(L) = \bigcup_{u \in L} R(u).$$

Par exemple, si $R = \{(a,b)\}$, alors $R(abab) = \{baab, abba\}$.

Pour tout langage L, on définit inductivement $R^k(L)$ pour $k \geq 0$ par : $R^0(L) = L$ et $R^{k+1}(L) = R(R^k(u))$. Par exemple

$$R^*(abab) = \{baab, baba, bbaa, abba\}.$$

R-mélange

Le R-mélange de deux mots u et v, noté $u \sqcup\!\sqcup_R v$ est l'ensemble des mots de la forme $u_1v_1 \dots u_nv_n$ où les u_i et les v_i sont des mots satisfaisant $u = u_1 \dots u_k$, $v = v_1 \dots v_k$ et pour tout $1 \leq i < j \leq n$ $(\alpha(u_j), \alpha(v_i)) \subseteq R$. Notons que si $R = A \times A$, alors le R-mélange est le produit de mélange classique (appelé dans la littérature *shuffle* ou *interleaving*). Par exemple, si $R = \{(a,b)\}$, alors

$$aba \sqcup\!\sqcup_R bba = \{ababba, abbaba, abbbaa\}.$$

Naturellement, le R-mélange s'étend aux langages :

$$L_1 \sqcup\!\sqcup_R L_2 = \bigcup_{u_1 \in L_1, u_2 \in L_2} u_1 \sqcup\!\sqcup_R u_2.$$

Le principal résultat qui nous sera utile est le suivant.

Proposition 3.1 ([DM97]) *Pour tous langages $L_1, L_2, \dots L_n$ sur A et toute relation de semi-commutation R,*

$$R^*(L_1L_2 \dots L_k) = L_1 \sqcup\!\sqcup_R (L_2 \sqcup\!\sqcup_R (\dots (L_{n-1} \sqcup\!\sqcup_R L_n) \dots)).$$

Automates partiellement ordonnés et APC

La définition d'automate partiellement ordonné provient de [STV01], même si la classe sous-jacente a été utilisée antérieurement, par exemple dans [Arf87, Arf91, PW97].

Définition 3.2 *Un automate de mot $\mathcal{A} = (Q, A, E, I, F)$ est dit partiellement ordonné s'il existe un ordre partiel $\leq$ sur Q tel que pour chaque transition $(p,a,q) \in E$, $p \leq q$.*

Intuitivement, un automate partiellement ordonné est un automate dans lequel les boucles simples sont toutes de longueur 1. Par exemple, l'automate de la figure 3.1 est partiellement ordonné avec comme ordre sur les états l'ordre sur les entiers.

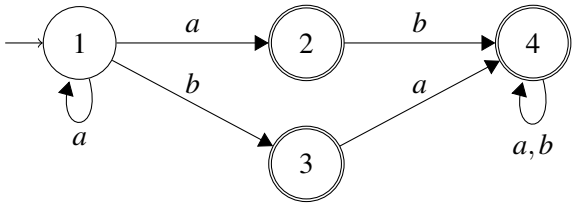


FIG. 3.1 – Automate partiellement ordonné.

Tous les langages réguliers ne sont pas reconnaissables par un automate partiellement ordonné, comme par exemple $(ab)^*$. Il existe différentes caractérisations dont certaines sont données dans la proposition 3.3. La caractérisation *iii*) sera commentée et précisée dans la section 3.4. D'autres caractérisations existent dans [PW97, STV01, TT02].

Proposition 3.3 *Les assertions suivantes sont équivalentes :*

i) L est accepté par un automate partiellement ordonné.

ii) L est une union finie de langage du type $A_0^* a_1 A_1^* \dots a_k A_k^*$*, où les* A_i *sont inclus dans A et les* a_i *sont des lettres.*

iii) L est reconnaissable par une formule Σ_2*.*

L'équivalence entre *i*) et *ii*) est triviale. L'équivalence entre *i*) et *iii*) découle d'un résultat plus général de W. Thomas [Tho82]. Une expression du type *ii*) est appelée APC pour *Alphabetic Pattern Constraints* [BMT07]. Il faut noter que l'on peut décider [Arf87] en temps polynomial [PW97] si un langage, donné par un automate déterministe, est reconnaissable par un automate partiellement ordonné. Le problème est même dans NLOGSPACE [BMT07]. En revanche, on ne sait pas calculer en temps raisonnable une relation APC codant le langage [Héa03]. Savoir si l'on peut, à partir d'un automate déterministe, calculer un automate partiellement ordonné reconnaissant le même langage en temps polynomial est un problème ouvert. Enfin, le problème de l'universalité de langages donnés par APC est PSPACE-complet [BMT07], comme dans le cadre général.

Une restriction de la classe des langages partiellement ordonnés est étudiée pour des problématiques de vérification dans [ABJ98, AAB99].

3.1.2 Motivations et résultats connus

Les relations de semi-commutation apparaissent fréquemment lorsque l'on modélise des systèmes, que cela soit pour modéliser des passages de jetons ou du parallélisme : plusieurs cas d'applications sont détaillés dans [BMT07]. Cependant, si $R = \{(a,b),(b,a)\}$ et $L = (ab)^*$, alors $R^*(L)$ est l'ensemble des

mots contenant autant de a que de b n'est pas régulier. Le premier problème intéressant est donc le suivant.

SCEstRegulier
Entrée : L régulier, R relation de semi-commutation
Question : Est-ce que $R^*(L)$ est régulier ?

Le problème **SCEstRegulier** est indécidable, même pour les commutations partielles [Sak92] et même si l'on restreint L à être sans-étoile [MP96]. En revanche, pour les commutations partielles transitives, le problème est décidable [Sak92].

Plutôt que d'opérer par sur-approximations comme dans le chapitre précédant, on va chercher à restreindre le problème. Plus précisément on s'intéresse au second problème : on cherche des classes de langages $\mathcal{C}$ et de relations de semi-commutation $\mathcal{R}$ tels que pour tout $L \in \mathcal{C}$ et tout $R \in \mathcal{R}$, $R^*(L)$ est dans $\mathcal{C}$. Dans ce cadre, les résultats suivants sont connus [GRS04, BMT07, GGP08b].

Proposition 3.4
- *La classe des langages qui sont une union finie de langages du type $A^*a_1A^*\dots a_kA^*$, où les a_i sont des lettres, est close par commutation,*
- *La classe $\mathcal{J}$, clôture booléenne de la classe ci-dessus, est close par commutation,*
- *La classe des langages reconnaissables par des automates partiellement ordonnés est close par semi-commutation,*
- *La classe PolG des langages qui sont une union finie de langages du type $L_0a_1L_1\dots a_kL_k$, où les a_i sont des lettres et les L_i des langages à groupe (reconnaissable par un automate complet dont toutes les transitions induisent une bijection sur l'ensemble des états) est close par commutation. Elle est aussi close par les commutations partielles R telles que $A \times A \setminus R$ soit transitive. Avec d'autres conditions sur R, plus techniques, si L appartient à PolG, alors $R^*(L)$ est régulier.*

3.2 Calcul du R-mélange

Le point clé des résultats est le suivant : si L_1 et L_2 sont des langages réguliers, alors $L_1 \sqcup\!\sqcup_R L_2$ est régulier aussi. De plus, on peut faire ce calcul en temps polynomial.

3.2.1 Calcul sur les automates

L'approche que nous avons développée s'appuie aussi sur le calcul du R-mélange et la proposition 3.1, mais en utilisant des automates.

Proposition 3.5 *Soient* $\mathcal{A}_1 = (Q_1, A, E_1, I_1, F_1)$ *et* $\mathcal{A}_2 = (Q_2, A, E_2, I_2, F_2)$ *deux automates finis et* R *une relation de semi-commutation sur* A. *Si* $B \subseteq \alpha(L(\mathcal{A}_2))$, *on note* B_R *l'ensemble* $\{a \in \alpha(L(\mathcal{A}_1)) \mid \{a\} \times B \subseteq R\}$ *et* $\overline{B}$ *l'ensemble* $\{a \in \alpha(L(\mathcal{A}_2)) \mid B_R \times \{a\} \subseteq R\}$.

L'automate fini $\mathcal{A} = (Q, A, E, I, F)$ *définit par :*

- $Q = Q_1 \times Q_2 \times \mathcal{P}(A)$,
- $I = \{(p_1, p_2, \overline{\emptyset}) \mid p_1 \in I_1, p_2 \in I_2\}$,
- $F = \{(p_1, p_2, B) \mid p_1 \in F_1, p_2 \in F_2, B \subseteq A\}$,
- $E = G_1 \cup G_2$, *avec*
  $G_1 = \{((p_1, p_2, B), a, (q_1, p_2, B)) \mid p_1 \in Q_1, p_2 \in Q_2, q_1 \in p_1 \cdot a, B \subseteq A$ *et* $\{a\} \in B_R\}$ *et*
  $G_2 = \{((p_1, p_2, B), a, (p_1, q_2, \overline{B \cup \{a\}})) \mid p_1 \in Q_1, p_2 \in Q_2, q_2 \in p_2 \cdot a, B \subseteq A\}$.

est noté $\mathcal{A}_1 \sqcup\!\sqcup_R \mathcal{A}_2$ *et reconnaît* $L(\mathcal{A}_1) \sqcup\!\sqcup_R L(\mathcal{A}_2)$.

La construction des transitions est illustrée par la figure 3.2. L'intuition de la construction est de modifier la construction classique pour le mélange, mais en marquant les lettres lues du second mot dans un ensemble[1] B.

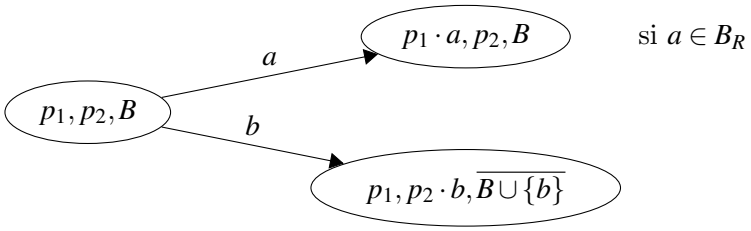


FIG. 3.2 – Transitions pour le R-mélange

Considérons par exemple les automates $\mathcal{A}_1$ et $\mathcal{A}_2$ de la figure 3.3 ainsi que la relation de semi-commutation suivante :

$$R = \{(b,c), (b,d), (a,c)\}.$$

Dans ce cas, $\mathcal{A}_1 \sqcup\!\sqcup_R \mathcal{A}_2$ est l'automate de la figure 3.4 (en ne représentant que les états accessibles).

3.2.2 Calcul sur APC - Résultats expérimentaux

Il est connu (et trivial), que les automates partiellement ordonnés ont la même expressivité que les expressions APC. On peut facilement démontrer que si $\mathcal{A}_1$ et $\mathcal{A}_2$ sont des automates partiellement ordonnés, alors $\mathcal{A}_1 \sqcup\!\sqcup_R \mathcal{A}_2$ l'est

[1] L'opération notée par une barre est une optimisation à la volée pour limiter le nombre d'états. La proposition resterait vraie si on supprimait les barres.

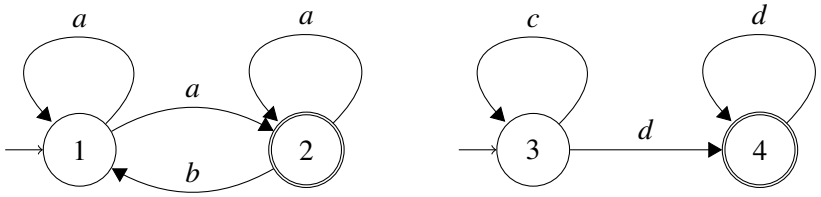


FIG. 3.3 – Automates $\mathcal{A}_1$ et $\mathcal{A}_2$

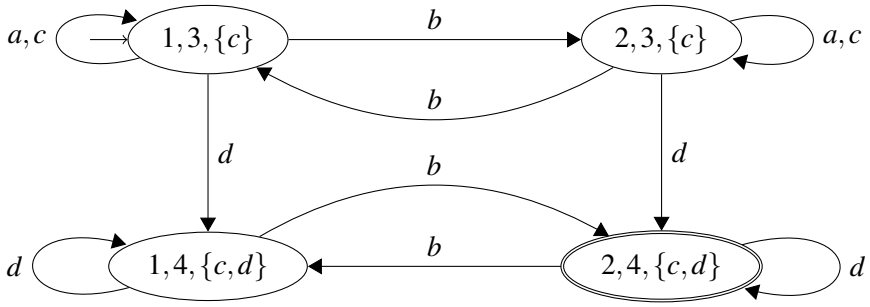


FIG. 3.4 – R-mélange des automates $\mathcal{A}_1$ et $\mathcal{A}_2$.

aussi. Par une récurrence directe, la clôture des APC par semi-commutation est une conséquence immédiate de la proposition 3.1. On pourrait légitimement se demander si notre approche ne consiste pas en un simple codage par automates de l'algorithme de [BMT07] dans lequel on fusionnerait des états. Comme nous espérons l'illustrer, la réponse est négative car non seulement notre approche permettra d'obtenir un résultat plus large (voir section 3.3), mais, de plus, fusionner des états équivalents dans un automate partiellement ordonné est un problème PSPACE-complet [BMT07].

Considérons l'exemple suivant pour illustrer la *factorisation* des états : Soient $B = \{a,b,c\}$, $C = \{d,e,f\}$ et $R = \{(a,d),(c,f),(b,d),(b,e)\}$. En utilisant la proposition 3.5, on obtient :

B	R_B	$\overline{B}$
$\emptyset$	$\{a,b,c\}$	$\emptyset$
$\{d\}$	$\{a,b\}$	$\{d\}$
$\{e\}$	$\{b\}$	$\{d,e\}$
$\{f\}$	$\{c\}$	$\{f\}$
$\{d,e\}$	$\{b\}$	$\{d,e\}$
$\{e,f\}$	$\emptyset$	$\{d,e,f\}$
$\{d,f\}$	$\emptyset$	$\{d,e,f\}$
$\{d,e,f\}$	$\emptyset$	$\{d,e,f\}$

Donc, le langage $B^* \sqcup\!\sqcup_R C^*$, qui est en fait $R^*(B^*C^*)$ est reconnu par l'automate de la figure 3.5.

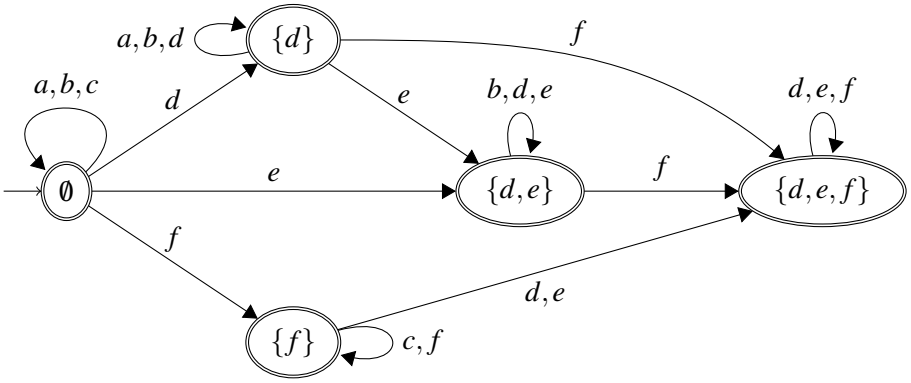


FIG. 3.5 – Automates $\mathcal{A}_1$ et $\mathcal{A}_2$

En utilisant [BMT07, Example 2], on obtient que $R^*(B^*C^*) = \{a,b,c\}^*\{c,f\}^*\{d,e,f\}^* \cup \{a,b,c\}^*\{a,b,d\}^*\{b,d,e\}^*\{d,e,f\}^*$ qui est précisément le langage reconnu par l'automate ci-dessus. La *compacité* des automates se voit déjà sur cet exemple où les expressions $\{a,b,c\}^*$ et $\{d,e,f\}^*$ sont chacune *factorisées* dans un état.

Passons maintenant à une comparaison expérimentale des deux méthodes. Pour ces tests, nous avons choisi aléatoirement des alphabets, des relations de semi-commutation et des produits. Comme critère de comparaison, nous nous sommes intéressés à la taille des expressions produites (en nombre de symboles), et la taille des automates générés (nombre de transitions plus nombre d'états). Le développement a été fait en OCAML.

Comme leur effet sur les algorithmes est très différent, nous avons utilisé deux formes d'entrées.

type 1 : $A_0^* a_1 A_2^* \dots a_{n-1} A_n^*$

type 2 : $A_0^* A_1^* \dots A_n^*$

Plus précisément, notre procédure est la suivante : pour chaque test, on fixe la longueur n du produit, la taille $|A|$ de l'alphabet ainsi que la taille $|R|$ de la relation de semi-commutation. Avec ces limites, on génère aléatoirement un produit, et nous exécutons les deux algorithmes sur ce produit. Chaque résultat présenté dans les tableaux 3.1 et 3.2 est en fait une moyenne obtenue sur 15 tests à chaque fois. Les chiffres indiqués sont les tailles des sorties.

Tous ces tests ont été effectués en moins de deux minutes sur un Athlon de 1.3 GHz avec 1 GB de mémoire. Notre approche utilise autour de 3 à 4 MB de mémoire alors que celle de [BMT07] croit beaucoup plus rapidement avec la taille des entrées (plus de 800 MB pour certains tests).

Taille du produit		2	3	5	7
APC	type 1	10	418	48361	897004
automates	type 1	28	82	333	836
APC	type 2	-	15	252	6402
automates	type 2	-	50	206	591

TAB. 3.1 – Comparaison des techniques en fonction de n avec $|A| = 10$ et $|R| = 5$

Taille de la relation		3	5	7	9
APC	type 1	785597	1162952	286499	4213859
automate	type 1	578	828	1031	1522
APC	type 2	7540	15153	16965	29730
automate	type 2	502	622	830	936

TAB. 3.2 – Comparaison des techniques en fonction de n avec $|A| = 10$ et $|R| = 7$

On a aussi appliqué notre technique sur un langage de type 1 avec $n = 40$, $|A| = 10$ et $|R| = 10$. L'automate obtenu avait une taille d'environ 450000 et le calcul a pris 42 heures et 128 M de mémoire. Ce type de test ne peut pas être effectué avec l'approche [BMT07], à cause d'un rapide dépassement mémoire.

Pour conclure cette section, il faut aussi remarquer que la plupart des algorithmes de model-checking réguliers fonctionnent sur des automates et les propriétés importantes comme l'intersection ou le test du vide se font en utilisant des automates finis. Le choix de l'encodage du problème par des automates plutôt que par des expressions régulières n'est pas gênant dans le cadre de la vérification.

3.3 PolCom et autres classes

3.3.1 PolCom

Nous présentons ici notre résultat principal sur les relations de semi-commutation, même s'il s'énonce brièvement.

Un langage est dit *commutatif* s'il est clos par commutation. Un langage est dans PolCom s'il est une union finie de langages du type $L_0 a_1 L_1 \dots a_k L_k$ où les L_i sont des langages réguliers commutatifs et les a_i des lettres. Il est clair que PolCom contient strictement les langages reconnaissables par un automate partiellement ordonné (car les langages du type B^*, avec $B \subseteq A$, sont commutatifs).

Le résultat sur le produit de mélange permet de montrer le théorème suivant.

Théorème 3.6 *La classe PolCom est close par semi-commutation.*

Par ailleurs, par application directe de résultats de [PW97], on sait que PolCom est close par quotient, union, produit et intersection. Il faut noter que la preuve du théorème 3.6 est constructive. Par exemple, soient $L = (a^*ba^*ba^*)^*$ l'ensemble des mots sur $\{a,b\}$ avec un nombre pair de b et $R = \{(b,c),(b,d),(a,c)\}$ une relation de semi-commutation sur $\{a,b,c,d\}$. Alors on peut montrer que $R^*(Lc^*dd^*)$ est donné par l'automate de la figure 3.4.

3.3.2 Pour aller plus loin

Dans le cadre du model-checking régulier, le calcul du R-mélange, associé à la proposition 3.1, permet de calculer $R^*(L_1 \dots L_k)$ dès que l'on sait calculer les $R^*(L_i)$, notamment si $R^*(L_i) = L_i$. La recherche de la stabilité par R est suffisante, la stabilité par toute relation de semi-commutation n'est pas nécessaire.

Nous avons mis en place des heuristiques[2] pour, étant donné un langage L, le décomposer en produits de langages R-clos, et calculer alors sa R-clôture en évitant des redondances de calcul.

L'idée est illustrée sur la figure 3.6 : on suppose que le langage L est donné par un automate fini $\mathcal{A}$ et que l'on souhaite calculer $R^*(L)$, où R est une relation de semi-commutation. On commence par faire un tri topologique sur $\mathcal{A}$: on décompose $\mathcal{A}$ en composantes fortement connexes (sur l'exemple, il y en a quatre, représentées par les $\mathcal{A}_i$). On teste ensuite si tous les automates obtenus ainsi (sur l'exemple, on aura $\mathcal{A}_1$ avec comme état initial 0 et comme état final 1, $\mathcal{A}_1$ avec comme état initial 0 et comme état final 2, $\mathcal{A}_2$ avec comme état initial 3 et comme état final 7, etc.), sont R-clos. Alors on pourra calculer $R^*(L)$ en décomposant tous les chemins possibles dans le graphe engendré par le tri topologique. Comme dans le calcul du R-mélange les états initiaux et finaux interviennent peu, de nombreux calculs peuvent être factorisés.

3.3.3 Exemple : un contrôleur d'ascenseurs

Nous illustrons les résultats précédents sur l'exemple d'un système de deux ascenseurs.

Configurations du système

Considérons un contrôleur gérant deux ascenseurs qui desservent respectivement les étages pairs et impairs d'un immeuble. Pour simplifier la présentation, nous supposons que cet immeuble présente un nombre pair d'étages (en

[2] Ces heuristiques ne sont pas des algorithmes car elles ne trouvent par forcement une décomposition s'il en existe une.

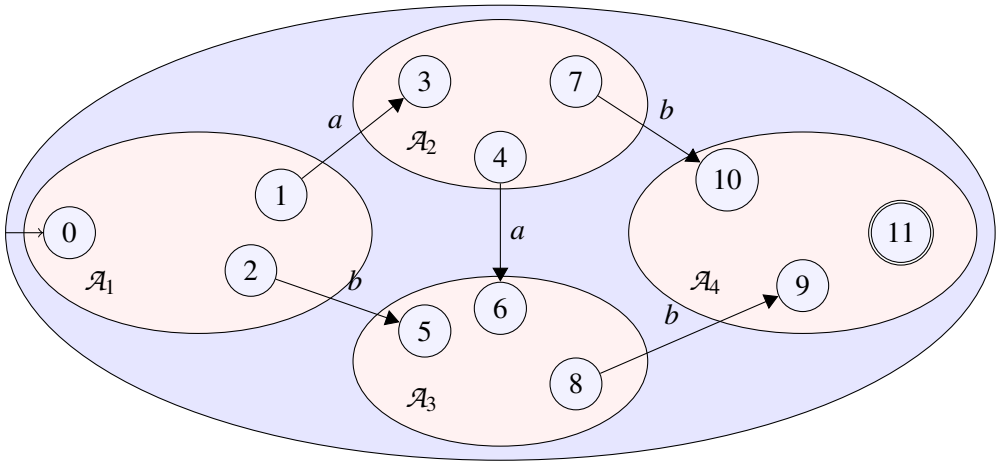


FIG. 3.6 – Heuristique de décomposition

comptant le rez-de-chaussée) et nous comptons les étages à partir du rez-de-chaussée (qui porte donc le numéro d'étage 1). Des usagers peuvent arriver à n'importe quel moment et n'importe quel étage, et déclarer qu'ils veulent monter ou qu'ils veulent descendre. Les ascenseurs sont initialement au rez-de-chaussée et naviguent ensuite, de manière indépendante, de bas en haut, jusqu'au niveau le plus élevé, puis de haut en bas, jusqu'au rez-de-chaussée et ainsi de suite. Dans sa phase ascendante (resp. descendante), un ascenseur prend tous les usagers voulant monter (resp. descendre), et ne tient pas compte des autres.

Pour chaque n pair, représentant le nombre d'étages, une configuration du système peut être représentée par un mot de la forme :

$$\#x_1x_2\ldots x_jl_rx_{j+1}\ldots x_kl_sx_{k+1}\ldots x_n\#$$

où pour $q \in \{1,2\}$, $l_q \in \{a_{q\uparrow}, a_{q\downarrow}\}$ indique le sens de déplacement du $q^{\text{ième}}$ ascenseur et $x_i \in \{\perp_1, p_{1\uparrow}, p_{1\downarrow}, p_{1\uparrow\downarrow}\}$ pour les i impairs et $x_i \in \{\perp_2, p_{2\uparrow}, p_{2\downarrow}, p_{2\uparrow\downarrow}\}$ pour les i pairs indique les requêtes (s'ils veulent monter ou descendre) des usagers au $i^{\text{ème}}$ étage. Décrivons maintenant la sémantique des symboles :

- les positions de l_r et l_s dans le mot indiquent les positions des ascenseurs. Si $l_r = a_{1\uparrow}$ (resp. $l_r = a_{1\downarrow}$) alors l'ascenseur 1 est à l'étage $j+1$ (resp. j) et il se déplace vers le haut (resp. vers le bas) ;
- $x_i = p_{j\uparrow\downarrow}, j \in \{1,2\}$ signifie que des personnes à l'étage i, desservi par l'ascenseur j, attendent pour monter et d'autres pour descendre ;
- $x_i = p_{j\uparrow}$(resp. $x_i = p_{j\downarrow}$), avec $j \in \{1,2\}$ le numéro d'ascenseur, signifie que des personnes, à l'étage i, attendent uniquement pour monter (resp. descendre) ;

– $x_i = \perp_j, j \in \{1,2\}$ indique que personne n'a fait de demande à l'étage i.

Par exemple, la configuration suivante

$$\#\perp_1 p_{2\downarrow} p_{1\uparrow\downarrow} a_{1\uparrow} a_{2\downarrow} p_{2\downarrow} \#$$

décrit un système à quatre étages. Au premier, personne n'a fait de demande. Au second et au quatrième étages, des usagers veulent descendre. Au troisième étage, qui est desservi par l'ascenseur 1, on trouve à la fois des personnes voulant monter et des personnes voulant descendre. Dans cette configuration, l'ascenseur 1 se situe au quatrième étage et l'ascenseur 2 au troisième.

L'ensemble des configurations initiales, pour un nombre pair quelconque d'étages, est l'ensemble de mots $L_0 = \#a_{1\uparrow} a_{2\uparrow} (\perp_1 \perp_2)^* \#$ où les ascenseurs sont au premier étage, a_1 devant a_2, et aucun usager n'a fait de demande. Nous pouvons remarquer que l'ensemble L_0 n'est pas reconnaissable par un automate partiellement ordonné et n'appartient pas à PolCom.

Transitions du système

Le comportement dynamique du système, c'est-à-dire ses transitions, est représenté par les ensembles de règles de réécriture décrites ci-après.

Les règles *request* sont des substitutions représentant l'arrivée d'usagers dans le système. La montée (resp. la descente) d'un ascenseur, sans prise en charge d'usager, est modélisée par les relations de semi-commutation *move-up* (resp. *move-down*). Les règles *take-up* (resp. *take-down*) représentent l'action de prise en charge des usagers qui veulent monter (resp. descendre) par un ascenseur qui monte (resp. descend). Les changements de directions des ascenseurs sont modélisés par les règles *up2down* et *down2up*. Finalement, le croisement des ascenseurs est représenté par les relations de semi-commutation *cross*. Plus précisément, nous avons :

request : $i \in \{1,2\}$

$$\begin{array}{rcl} \perp_i & \to & p_{i\downarrow} \\ \perp_i & \to & p_{i\uparrow} \\ p_{i\uparrow} & \to & p_{i\uparrow\downarrow} \\ p_{i\downarrow} & \to & p_{i\uparrow\downarrow} \end{array}$$

move-up : $i, j \in \{1,2\}$

$$\begin{array}{rcll} a_{i\uparrow}\perp_j & \to & \perp_j a_{i\uparrow} & \\ a_{i\uparrow} p_{j\downarrow} & \to & p_{j\downarrow} a_{i\uparrow} & \\ a_{i\uparrow} p_{j\uparrow} & \to & p_{j\uparrow} a_{i\uparrow} & i \neq j \\ a_{i\uparrow} p_{j\uparrow\downarrow} & \to & p_{j\uparrow\downarrow} a_{i\uparrow} & i \neq j \end{array}$$

take-up : $i \in \{1,2\}$

$$\begin{array}{rcl} a_{i\uparrow} p_{i\uparrow} & \to & \perp_i a_{i\uparrow} \\ a_{i\uparrow} p_{i\uparrow\downarrow} & \to & p_{i\downarrow} a_{i\uparrow} \end{array}$$

up2down : $i \in \{1,2\}$

$$a_{i\uparrow}\# \to a_{i\downarrow}\#$$

cross : $i \in \{1,2\}$ et $i \neq j$

$$\begin{array}{rcl} a_{i\downarrow} a_{j\downarrow} & \to & a_{j\downarrow} a_{i\downarrow} \\ a_{i\uparrow} a_{j\uparrow} & \to & a_{j\uparrow} a_{i\uparrow} \\ a_{i\uparrow} a_{j\downarrow} & \to & a_{j\downarrow} a_{i\uparrow} \end{array}$$

move-down : $i, j \in \{1,2\}$

$$
\begin{array}{rcll}
\bot_j a_{i\downarrow} & \to & a_{i\downarrow}\bot_j & \\
p_{j\uparrow} a_{i\downarrow} & \to & a_{i\downarrow} p_{j\uparrow} & \\
p_{j\downarrow} a_{i\downarrow} & \to & a_{i\downarrow} p_{j\downarrow} & i \neq j \\
p_{j\uparrow\downarrow} a_{i\downarrow} & \to & a_{i\downarrow} p_{j\uparrow\downarrow} & i \neq j
\end{array}
$$

$$
\begin{array}{rcl}
p_{i\downarrow} a_{i\downarrow} & \to & a_{i\downarrow}\bot_i \\
p_{i\uparrow\downarrow} a_{i\downarrow} & \to & p_{i\uparrow} a_{i\downarrow}
\end{array}
$$

down2up : $i \in \{1,2\}$

take-down : $i \in \{1,2\}$

$$\#a_{i\downarrow} \to \#a_{i\uparrow}$$

Ensemble des configurations atteignables

Préoccupons nous maintenant du calcul de l'ensemble des configurations atteignables du système. Nous extrayons tout d'abord de l'ensemble des règles R, le sous-ensemble R_{sc} des règles de semi-commutation (c'est-à-dire *cross*, *move-up* et *move-down*). On pose alors $T = R \setminus R_{sc}$. Ensuite, on pose $L_n = T(R_{sc}^*(L_{n-1}))$, pour $n \geq 1$. On calcule les L_i tant que l'on peut (i.e. tant que l'heuristique pour le calcul de la clôture par semi-commutation fonctionne) et on s'arrête lorsque l'on obtient un point fixe $L_n = L_{n+1}$ (ce qui pourrait ne pas arriver).

Nous avons implémenté ces procédures avec le langage fonctionnel OCaml. Nous avons pu, à l'aide de cette implantation, calculer automatiquement l'ensemble des configurations atteignables (point fixe). Cet ensemble d'atteignabilité est représenté par un automate d'une dizaine d'états et d'une cinquantaine de transitions après minimisation.

3.4 Relations de semi-commutation sur les arbres

La notion d'automate partiellement ordonné s'étend facilement aux arbres. Un automate d'arbre est *partiellement ordonné* s'il existe une relation d'ordre $\leq$ sur ses états telle que si $f(q_1,\ldots,q_n) \to q$ est une transition, alors pour chaque i, $q_i \leq q$. On montre facilement que la classe des langages d'arbre reconnaissables par un automate partiellement ordonné est stable par union et intersection.

3.4.1 Automates d'arbre partiellement ordonnés et formules du premier ordre.

Nous invitons le lecteur intéressé par les caractérisations logiques des langages réguliers d'arbre à consulter [CDG+02, Section 3.3] ou [Tho97, Chapitre 7].

Nous montrons dans cette section que la classe des langages d'arbre acceptés par des formules que l'on définira (et dites *formules* Σ_2), est strictement incluse dans la classe des langages acceptés par des automates partiellement ordonnés.

Commençons par définir les formules logiques que nous allons utilisées. Soit $\mathcal{F}$ un ensemble de symboles avec arités et $\mathcal{X}$ un ensemble de variables. Une *formule atomique* sur $\mathcal{F}$ est définie inductivement par :

- $(p_1 = p_2)$, $(p_1 <_i p_2)$ et $R_f(p_1)$, où p_1 et p_2 sont des éléments de $\mathbb{N}^* \cup \mathcal{X}$, $f \in \mathcal{F}$, et $i \in \mathbb{N}$, sont des formules atomiques.
- Si φ_1 et φ_2 sont des formules atomiques, alors $\neg\varphi_1$ et $(\varphi_1 \vee \varphi_2)$ sont aussi des formules atomiques.

Si dans la définition précédente, on se restreint aux p_1, p_2 et p dans $\mathcal{X}$, on obtient la sous classe des formules atomiques appelées *formules atomiques sans constante*. Par exemple $\varphi_{\text{exe}} = (\neg R_f(y) \vee (x = y))$ est une formule atomique sans constante. Pour tout terme t, on définit inductivement la fonction ν_t des formules atomiques dans $\{0,1\}$ par :

- $\nu_t((p_1 = p_2)) = 1$ ssi p_1 et p_2 sont des positions t et si $p_1 = p_2$.
- $\nu_t((p_1 <_i p_2)) = 1$ ssi $p_1 \cdot i$ et p_2 sont des positions de t et si $p_1 \cdot i$ est un préfixe de p_2.
- $\nu_t(R_f(p)) = 1$, ssi p est une position de t et si $t(p) = f$.
- $\nu_t(\neg\varphi) = 1 - \nu_t(\varphi)$.
- $\nu_t((\varphi \vee \varphi_2)) = \max(\nu_t(\varphi_1), \nu_t(\varphi_2))$.

Par exemple, si $t_{\text{exe}} = f(A, h(A))$ alors $\nu_{t_{\text{exe}}}(R_A(2.1)) = 1$.

On dit que t satisfait φ, si $\nu_t(\varphi) = 1$. Une formule de la forme

$$\exists x_1 \ldots \exists x_n \forall y_1 \ldots \forall y_k \ \varphi(x_1, \ldots, x_n, y_1, \ldots, y_k),$$

où φ est une formule atomique sans constante est appelée une *formule* Σ_2. Un terme t satisfait une formule de Σ_2 si et seulement s'il existe des positions $p_1, \ldots, p_n$ de t telles que pour toutes positions $p'_1, \ldots, p'_k$ de t, t satisfait $\varphi(p_1, \ldots, p_n, p'_1, \ldots, p'_k)$. Dans ce cas on dit que t est *accepté* par la formule.

Par exemple, t_{exe} est accepté par la formule $\exists x \, \forall y \, \varphi_{\text{exe}}(x, y)$ (l'ensemble des termes acceptés par cette formule est l'ensemble des termes où le symbole f apparaît exactement une fois).

Proposition 3.7 *La classe des langages acceptés par des formules Σ_2 est strictement incluse dans la classe des langages acceptés par des automates d'arbre partiellement ordonnés.*

L'inclusion se montre de façon constructive. Pour montrer que les deux classes ne sont pas égales, on utilise le témoin suivant : $\mathcal{F} = \{f, a, b, \bot\}$ où f est d'arité 2, a, b sont d'arité 1 et $\bot$ est une constante. L'automate donné par les transitions $\bot \to q_0$, $a(q_0) \to q_0$, $b(q_0) \to q_0$, $a(q_0) \to q_1$, $f(q_1, q_1) \to q_1$ et l'état final est q_1. Cet automate est partiellement ordonné mais on peut montrer que le langage qu'il accepte n'est pas reconnaissable par une formule Σ_2.

3.4.2 Automates d'arbre partiellement ordonnés et semi-commutation

On note $\mathcal{F}_k$ le sous-ensemble de $\mathcal{F}$ des symboles d'arité k. Une relation de semi-commutation S est un sous-ensemble de $\cup_{k\geq 1}\mathcal{F}_k \times \mathcal{F}_k$. Comme pour les mots, on définit $S(t)$ où t est un terme comme l'ensemble des termes s ayant les même positions que t et tel qu'il existe deux positions p et $p\cdot i$ de s telles que $(s(p), s(p\cdot i)) \in S$, $s(p) = t(p\cdot i)$, $s(p\cdot i) = t(p)$ et pour toute autre position p' de s, $s(p') = t(p')$. On définit $S^*(K)$ de façon naturelle.

Proposition 3.8 *Il existe un langage régulier K accepté par une formule formule Σ_2 et tel que $S^*(K)$ n'est pas un langage régulier d'arbre.*

Un langage K possible est l'ensemble des termes de la forme $h^*(f(t_1,t_2))$ où h est d'arité 1, f est d'arité 2, t_1 est un terme quelconque de $\mathcal{T}(\{f,\perp\})$ ($\perp$ est une constante, f est d'arité deux) et t_1 est un terme quelconque de $\mathcal{T}(\{g,\perp\})$ ($\perp$ est une constante, g est d'arité deux). La relation S utilisée est $S = \{(f,g),(g,f)\}$.

Le langage K est accepté par la formule suivante :

$$\begin{aligned}\exists y \forall x_1 \forall x_2 \forall x_3 \quad & R_f(y) \wedge (x_1 <_0 y \Rightarrow R_h(x_1)) \wedge \\ & (y <_0 x_2 \Rightarrow (R_f(x_2) \vee R_\perp(x_2))) \wedge (y <_1 x_3 \Rightarrow (R_g(x_3) \vee R_\perp(x_3)))\end{aligned}$$

Or $S^*(K)$ est l'ensemble des termes $\{h^*(f(t_1,t_2)) \mid |t_1|_g = |t_2|_f\} \cup \{h^*(g(t_1,t_2)) \mid |t_1|_g = |t_2|_f + 1\}$ où ($|t|_g$ et $|t|_f$ désignent respectivement le nombre de symboles g et f dans t), et n'est pas régulier.

3.5 Conclusion et perspectives

Nous avons montré comment calculer le R-mélange de deux langages réguliers définis par des automates. Cela nous a permit de développer un algorithme efficace pour le calcul de clôtures transitives par relations de semi-commutation d'un langage donné par un automate partiellement ordonné. De plus, sur le plan théorique, on a pu exhiber une classe, PolCom, stable par semi-commutation et contenant strictement toutes les classes de langages connues pour être stables par toutes les relations de semi-commutation. Les questions suivantes restent ouvertes :

– On ne sait pas si la classe PolCom est décidable. Par ailleurs, même si l'on sait qu'un langage est dans PolCom, calculer sa clôture par une relation de semi-commutation demande que ce langage soit décomposé sous une bonne forme. Le calcul d'une telle décomposition, s'il est théoriquement possible par énumération, mériterait une approche beaucoup

plus efficace pour fonctionner en pratique. Il faut noter que la même question se pose pour les langages acceptés par un automate partiellement ordonné : la classe est décidable (en espace logarithmique non déterministe), mais on ne sait pas efficacement calculer un automate partiellement ordonné à partir d'un automate déterministe.

– Les exemples qui montrent que l'on sort des réguliers en clôturant par semi-commutation reposent en général sur des arguments de comptage. Il serait intéressant de se pencher sur l'utilisation d'automates à compteurs afin de calculer des clôtures par semi-commutation pour des classes plus larges que les réguliers.
– Enfin, de nombreux résultats théoriques, notamment ceux de [GGP08b] ne proposent pas de procédures simples et/ou efficaces pour calculer les clôtures. Il serait intéressant de développer des algorithmes performants dans ce cadre.

Chapitre 4

Vérification de systèmes composés

Sommaire

4.1 Substitutivité et composition 70
4.2 Substitutivité qualitative et quantitative 71
4.2.1 Automates pondérés par des entiers 71
4.2.2 Substitutivité et substitutivité forte 72
4.2.3 Résultats pour l'inclusion de trace et la simulation 74
4.2.4 Expérimentation, services Web 75
4.3 Composition avec contraintes 75
4.3.1 Exemple introductif 76
4.3.2 Résultats 78
4.4 Idéaux de mélange 79
4.5 Conclusion et perspectives 79

Les publications liées à ce chapitre sont [22, 21, 19, 2] ainsi que les articles soumis [23, 24].

Nous nous intéressons dans ce chapitre à des problèmes de vérification sur des systèmes composés.

Section 4.1. Cette section introduit des problématiques générales de vérification sur les systèmes composés.

Section 4.2. Nous présentons dans cette section nos contributions sur des problèmes de substitutivité prenant en compte des aspects quantitatifs et qualitatifs. Nous donnons des résultats de complexité ainsi que des liens avec la composition. Ces travaux ont donné lieu aux publications [22, 21], ainsi qu'aux soumissions [23, 24].

Section 4.3. Dans cette section, nous nous intéressons à des composants dont le comportement varie selon des paramètres externes. Nous les modélisons à l'aide de classes d'automates paramétrés par des formules booléennes. Nous nous intéressons au problème de compositions de ces composants paramétrés. Ces travaux sont publiés dans [2].

Section 4.4. Cette section présente des travaux théoriques sur les idéaux de mélange, qui sont aussi exactement les langages clos par le haut pour la relation sous-mot et qui forment une classe de langages réguliers intéressante pour l'analyse des systèmes communicants. Ces résultats ont été publiés dans [19].

4.1 Substitutivité et composition

Concevoir, modéliser et développer des systèmes à partir de composants est une approche très répandue, notamment pour les systèmes embarqués, les réseaux de capteurs ou les services Web. Un composant est un système pouvant réaliser certaines tâches et qui dispose d'interfaces pour communiquer avec d'autres composants. En fonction des applications, des plates-formes, des objectifs, etc, de très nombreux problèmes théoriques ou pratiques se posent. Nous allons citer ici que quelques uns des problèmes de vérification pour les systèmes composés.

- **Composition.** Le problème de composition est de savoir si, étant donné un ensemble de composants, il existe un moyen de faire communiquer ces composants entre eux de telle sorte qu'ils effectuent une tâche donnée. Le problème de composition a été largement étudié dans le cadre des systèmes communicants, notamment après les travaux de Milner [Mil80].
- **Synthèse d'un contrôleur.** Le but de la synthèse de contrôleurs est d'imposer à un système une propriété en le synchronisant avec un automate (le contrôleur). La question est alors : à partir du système et de la propriété, comment calculer un contrôleur ?
- **Substitutivité.** Si un composant tombe en panne, il faut pouvoir le remplacer. Étant donnés deux composants, la question est alors de savoir si le second peut faire au moins la même chose que le premier.

4.2 Substitutivité qualitative et quantitative

La comparaison de systèmes communicants (processus, composants, etc.) est un sujet incontestablement important en informatique. Les premières approches théoriques orientées dans ce sens sont généralement attribuées à Milner [Mil80]. De très nombreux travaux ont été fait depuis sur de nombreux modèles, des automates communicants aux algèbres de processus en passant par les HMSC et sans oublier les réseaux de Petri.

Le problème de la substitutivité entre composants est étudiée dans de nombreux travaux, avec des nuances selon le modèle utilisé et la définition que l'on donne à la substitutivité. Dans [BSBM04], les auteurs définissent trois notions de compatibilité entre composants Web qui débouchent sur plusieurs notions de substitutivité avec comme modèle des algèbres de processus. Dans [MPC01], les auteurs modélisent les composants par des automates finis et étudient une substitutivité sous contexte (en comparant les traces), alors que [BCT06] s'intéresse au même problème pour la bisimulation. Dans le cadre très utilisé de la description de composants par automates d'interface [Hen03, CGS01], la substitutivité est étudiée dans [TBFM06]. Citons enfin [PBH07] qui étudie la substitutivité sous contexte (définis par du Mu-calcul), de composants donnés par des systèmes de transition.

Tous les travaux cités ci-dessus ne tiennent compte que de propriétés fonctionnelles. Or les exigences actuelles[1] visent à prendre en compte des propriétés quantitatives, comme les dépenses énergétiques.

4.2.1 Automates pondérés par des entiers

Un automate pondéré par des entiers est un automate dans lequel chaque transition est munie d'un poids dans $\mathbb{Z}$. Formellement, un automate pondéré par des entiers est un quintuplet (Q,A,E,I,F) où Q est un ensemble fini d'états, A est un alphabet fini, $I \subseteq Q$ est l'ensemble des états initiaux, $F \subseteq Q$ est l'ensemble des états finaux et $E \subseteq Q \times A \times \mathbb{Z} \times Q$ est l'ensemble des transitions et vérifie que si (p,a,c_1,q) et (p,a,c_2,q) sont dans E, alors $c_1 = c_2$. La figure 4.1 donne deux exemples d'automates pondérés par des entiers.

Le coût d'un chemin fini π dans un automate pondéré $\mathcal{A}$ par des entiers est la somme des coûts des transitions et il est noté $\mathrm{cost}_{\mathcal{A}}(\pi)$.

Un automate pondéré par des entiers est dit k-ambiguë si pour tout mot il existe au plus k chemins réussis d'étiquette ce mot. Un automate 1-ambiguë est dit *non-ambiguë*. Un automate est dit *finiment ambiguë* s'il existe un k tel qu'il soit k-ambiguë. Par exemple l'automate $\mathcal{A}_{\text{pond2}}$ est non-ambiguë (mais il n'est pas déterministe car il possède deux états initiaux). En revanche l'automate

[1] De très nombreux travaux existent depuis plusieurs années sur la prise en compte des aspects temporisés, mais nous n'aborderons pas du tout cet aspect particulier ici.

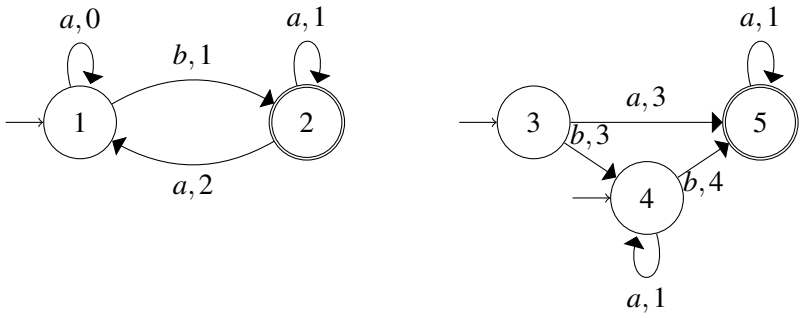


FIG. 4.1 – Automates $\mathcal{A}_{\text{pond1}}$ et $\mathcal{A}_{\text{pond2}}$

$\mathcal{A}_{\text{pond1}}$ n'est pas finiment ambiguë car le mot ba^n est l'étiquette de n chemins réussis.

Les résultats suivants sont connus sur les automates pondérés par des entiers.

Théorème 4.1 *Soient $\mathcal{A}_1$ et $\mathcal{A}_2$ deux automates pondérés par des entiers.*

- *On ne peut pas décider si pour tout chemin réussi π_1 de $\mathcal{A}_1$ il existe un chemin réussi π_2 de $\mathcal{A}_2$ tel que $\text{cost}_{\mathcal{A}_1}(\pi_1) \leq \text{cost}_{\mathcal{A}_2}(\pi_2)$ [Kro94]. Ce problème est décidable en temps polynomial si $\mathcal{A}_1$ et $\mathcal{A}_2$ sont finiment ambiguës [HIJ02, Web94].*
- *On ne peut pas décider si pour tout mot reconnu par $\mathcal{A}_1$, il existe un chemin réussi π_1 de $\mathcal{A}_1$ d'étiquette ce mot et tel que $\text{cost}_{\mathcal{A}_1}(\pi_1) \geq 0$ [Kro94].*

On note **WA** la classe de tous les automates pondérés par des entiers, **FA** la classe des automates pondérés par des entiers finiment ambiguës, **UA** la classe des automates pondérés par des entiers non-ambiguës et **D** la classe des automates finiment ambiguës pondérés par de entiers déterministes.

4.2.2 Substitutivité et substitutivité forte

Notre objectif est de définir des notions de substitutivité qui, tout en prenant en compte l'aspect fonctionnel, considèrent aussi des propriétés quantitatives. Considérons l'exemple introductif représenté par les deux composants modélisés par les automates de la figure 4.2. Ces composants représentent des systèmes qui, s'ils reçoivent une requête (action modélisée par la lettre a), font une action (notée b_1) puis, éventuellement, une autre action notée b_2. Le composant C_1 ne peut faire qu'une fois l'action b_2 alors que le composant C_2 peut la faire un nombre non borné de fois. Ensuite, les composants envoient une réponse (action modélisée par la lettre c). Pour chaque composant et chaque action, les coûts des actions sont notés par des poids sur les transitions. Dans

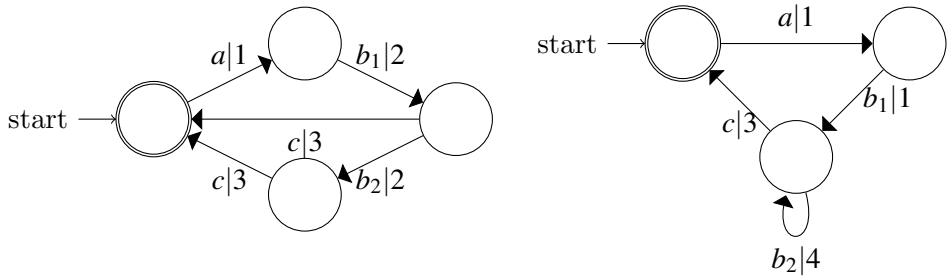


FIG. 4.2 – Composants C_1 et C_2

ce cadre et d'un point de vue fonctionnel (et informellement), tout ce que peut faire C_1 peut être fait par C_2. En revanche le coût pour effectuer la séquence d'actions ab_1b_2c est de 8 pour C_1 alors qu'il est de 9 pour C_2. Le composant C_2 peut donc être plus *coûteux* que le composant C_1.

Nous souhaitons modéliser la prise en compte de coûts pour la substitutivité. Pour cela, on considère que les composants sont modélisés par des automates à poids entiers et on définit les problèmes paramétrés suivants, où $\mathbf{C_1}$ et $\mathbf{C_2}$ sont des classes d'automates pondérés par des entiers et $\preceq$ est une relation entre les chemins réussis d'un automate de $\mathbf{C_1}$ et d'un automate de $\mathbf{C_2}$.

Substitutivité, noté $P^{\preceq}[\mathbf{C_1}, \mathbf{C_2}]$
Entrée : Deux automates pondérés par des entiers $\mathcal{A}_1 \in \mathbf{C_1}$ et $\mathcal{A}_2 \in \mathbf{C_2}$
Question : Pour tout chemin réussi π_1 de $\mathcal{A}_1$, il existe un chemin réussi π_2 de $\mathcal{A}_2$ tel que $\pi_1 \preceq \pi_2$ et $\text{cost}_{\mathcal{A}_2}(\pi_2) \leq \text{cost}_{\mathcal{A}_1}(\pi_1)$.

Informellement, le problème de la substitutivité peut s'énoncer : "*quoique puisse faire le premier composant, le second peut aussi le faire et avec un coût moindre*".

Substitutivité forte, noté $P^{\preceq}_{\text{strong}}[\mathbf{C_1}, \mathbf{C_2}]$
Entrée : Deux automates pondérés par des entiers $\mathcal{A}_1$ et $\mathcal{A}_2$
Question : Pour tout chemin réussi π_1 de $\mathcal{A}_1$, il existe un chemin réussi π_2 de $\mathcal{A}_2$ tel que $\pi_1 \preceq \pi_2$ et $\text{cost}_{\mathcal{A}_2}(\pi_2) \leq \text{cost}_{\mathcal{A}_1}(\pi_1)$ et pour tout chemin réussi π'_2 de $\mathcal{A}_2$ tel que $\pi_1 \preceq \pi'_2$, $\text{cost}_{\mathcal{A}_2}(\pi'_2) \leq \text{cost}_{\mathcal{A}_1}(\pi_1)$.

Informellement, le problème de la substitutivité peut s'énoncer : "*quoique puisse faire le premier composant, le second peut aussi le faire et quelque soit la façon dont il le fait, c'est toujours avec un coût moindre*". Il est clair que la substitutivité forte implique la substitutivité.

4.2.3 Résultats pour l'inclusion de trace et la simulation

Nous avons étudié la complexité des deux problèmes de substitutivité pour deux relations, l'inclusion de trace et la simulation.

Inclusion de trace

L'*inclusion de trace* tr est définie par : $\pi_1 \mathrm{tr}\ \pi_2$ ssi π_1 et π_2 ont le même label. Pour cette relation et pour la substitutivité forte, les résultats sont les suivants.

$P^{\mathrm{tr}}_{\mathrm{strong}}[\mathbf{C_1}, \mathbf{C_2}]$	$\mathbf{C_1 = D}$	$\mathbf{C_1 = UA}$	$\mathbf{C_1 = FA}$	$\mathbf{C_1 = WA}$
$\mathbf{C_2 = D}$	P	P	P	P
$\mathbf{C_2 = UA}$	P	P	P	PSPACE
$\mathbf{C_2 = FA}$	P	P	P	PSPACE
$\mathbf{C_2 = WA}$	PSPACE	PSPACE	PSPACE	PSPACE-complet

Pour la substitutivité faible, les résultats sont les suivants.

$P^{\mathrm{tr}}[\mathbf{C_1}, \mathbf{C_2}]$	$\mathbf{C_1 = D}$	$\mathbf{C_1 = UA}$	$\mathbf{C_1 = FA}$	$\mathbf{C_1 = WA}$
$\mathbf{C_2 = D}$	P	P	P	P
$\mathbf{C_2 = UA}$	P	P	P	PSPACE
$\mathbf{C_2 = FA}$	P	P	P	PSPACE
$\mathbf{C_2 = WA}$	indécidable	indécidable	indécidable	indécidable

Le résultat le plus difficile obtenu est la décidabilité en espace polynomial du problème $P^{\mathrm{tr}}[\mathbf{WA}, \mathbf{FA}]$, qui demande un codage non-trivial.

Simulation

Soient $\mathcal{A}_1 = (Q_1, A, E_1, I_1, F_1)$ et $\mathcal{A}_2 = (Q_2, A, E_2, I_2, F_2)$ deux automates pondérés par des entiers. Un relation $\mathcal{R} \subseteq Q_1 \times Q_2$ est une simulation si $(p_1, p_2) \in \mathcal{R}$ implique que pour tout a dans A et tout c_1 in $\mathbb{Z}$,

i) Pour chaque $q_1 \in Q_1$, si $(p_1, a, c_1, q_1) \in E_1$ alors il existe $q_2 \in Q_2$ et $c_2 \in \mathbb{Z}$ tels que $(p_2, a, c_2, q_2) \in E_2$ et $(q_1, q_2) \in \mathcal{R}$, et

ii) si p_1 est final, alors p_2 est final aussi.

On peut prouver que la plus large relation de simulation entre $\mathcal{A}_1$ et $\mathcal{A}_2$ existe. On la note $\preccurlyeq_{\mathcal{A}_1, \mathcal{A}_2}$ et simplement $\preccurlyeq$ s'il n'y a pas d'ambiguïté sur $\mathcal{A}_1$ et $\mathcal{A}_2$.

Cette relation s'étend aux chemins réussis de façon naturelle. Si π_1 est un chemin de $\mathcal{A}_1$ et π_2 un chemin de $\mathcal{A}_2$, alors $\pi_1 \preccurlyeq \pi_2$ si π_1 et π_2 ont la même longueur et si pour tout i, les i-èmes états de π_1 et π_2 sont en relation par $\preccurlyeq$.

Nous avons obtenu les résultats suivants.

Proposition 4.2 *Le problème* $P^{\preccurlyeq}_{\mathrm{strong}}[\mathbf{WA}, \mathbf{WA}]$ *est P-complet.*
Le problème $P^{\preccurlyeq}[\mathbf{WA}, \mathbf{FA}]$ *est résoluble en temps polynomial.*

La décidabilité de $P^{\preccurlyeq}[\mathbf{FA},\mathbf{FA}]$ est un problème ouvert. Nous avons aussi étudié la compatibilité de la substitutivité et de la substitutivité forte (pour la simulation) avec différentes relations de compositions (séquentielle, parallèle, synchronisée, etc). La substitutivité est compatible avec toutes ces relations ; en revanche la substitutivité forte n'est compatible que lorsque l'on compose avec des contextes (des automates pondérés) ayant un alphabet disjoint.

4.2.4 Expérimentation, services Web

Nous avons expérimenté de façon préliminaire les notions vues ci-dessus dans le cadre des services Web, dans le cas où ces services sont décrits par des fichiers WSDL et BPEL. Tout d'abord, nous avons enrichi le langage de spécification afin de prendre en compte des coûts. En nous appuyant sur une technique développée dans [FUMK07], nous avons alors extrait automatiquement à partir des fichiers de description des automates finis (pondérés) modélisant des services. La substitutivité a été vérifiée sur deux exemples jouets donnés dans les formats de description (pour ces deux exemples, les automates sont déterministes).

4.3 Composition avec contraintes

Nous nous intéressons ici au problème de composition : étant donnés des composants et un service attendu, peut-on composer les composants afin qu'ils satisfassent ce service. Notre travail porte sur le cas particulier de composants modélisés par des automates finis paramétrés par des formules logiques booléennes. Nous considérons un produit synchronisé par rendez-vous produisant des ε-transitions.

Le problème de comparer des systèmes composés n'est pas nouveau, la question naturelle étant de savoir s'il est possible de faire cette comparaison plus efficacement qu'en calculant explicitement les deux compositions, puis en effectuant la comparaison.

Dans [LS00] notamment, les auteurs regardent le problème suivant : étant donnés $n+m$ (avec $n,m \geq 2$) automates finis $\mathcal{A}_1,\ldots,\mathcal{A}_{n+m}$, quelle est la complexité de décider si $\mathcal{A}_1 \times \ldots \times \mathcal{A}_n$ est équivalent à $\mathcal{A}_{n+1} \times \ldots \times \mathcal{A}_{n+m}$. Le produit considéré est un produit partiellement synchronisé mais, contrairement à notre approche, la synchronisation ne produit pas de transitions ε mais des transitions étiquetées. Le résultat principal est de montrer que ce problème est EXPTIME-dur pour toute relation entre la simulation et la bisimulation et EXPSPACE-dur pour toute relation entre l'inclusion de trace et une relation plus complexe.

Un problème plus général est considéré dans [Rab97] où le produit est plus proche du notre. L'auteur prouve alors que le problème de [LS00] est

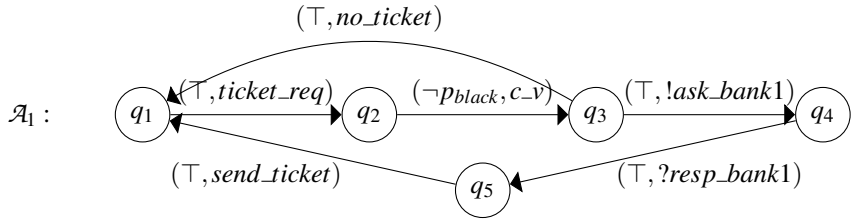


FIG. 4.3 – Automate booléen $\mathcal{A}_1$

PSPACE-dur pour toute relation entre la bisimulation et l'égalité de trace et EXPTIME-complet pour la bisimulation. Il y est aussi conjecturé que le problème est EXPTIME-dur pour toute relation entre la bisimulation et l'égalité de trace. Cette conjecture a été élargie et prouvé dans [Saw03] : le problème est EXPTIME-dur pour toute relation entre la bisimulation et le préordre de trace.

Il faut noter que dans [LS00, Rab97] et [Saw03] on a la condition importante que $n, m \geq 2$, ce qui n'est pas le cas qui nous intéresse (où le service que l'on souhaite réaliser par composition est donné explicitement). En revanche, dans [MW07], il est prouvé que décider si $\mathcal{A}$ est simulé par $\mathcal{A}_1 \times \ldots \times \mathcal{A}_n$ est EXPTIME-dur. Dans ce travail, le produit considéré est asynchrone.

4.3.1 Exemple introductif

Nous donnons ici un exemple introductif pour illustrer les problèmes qui nous intéressent. On considère trois services S_1, S_2 et S_3 utilisés pour organiser des voyages *footbalisitiques* en Espagne. Les services sont modélisés par des automates dont les transitions sont étiquetées par un couple comprenant une formule booléenne (qui sert de garde à la transition), et une lettre dénotant classiquement l'action modélisée par cette transition.

Le **Service** S_1 propose de réserver des places pour les matchs de football. Il est modélisé par l'automate $\mathcal{A}_1$ de la figure 4.3.

Lorsque ce service reçoit une requête pour réserver une place de football (ce que l'on modélise par l'action *ticket_req*), il vérifie s'il y a des places disponibles (action c_v). Cette vérification ne peut être faite que si la requête provient d'un utilisateur qui n'est pas sur liste noire (l'action a donc pour garde $\neg p_{black}$). S'il n'y a pas de place disponible, le Service S_1 informe l'utilisateur (action *no_ticket*). S'il y a de la place, il interroge alors le Service S_3 (la banque) pour le paiement du ticket (action !*ask_bank*1). Si cette dernière est d'accord pour le paiement (message ?*resp_bank*1), le ticket est envoyé à l'utilisateur (action *send_ticket*).

Le **Service** S_2 vend des tickets d'avion. Il est modélisé par l'automate

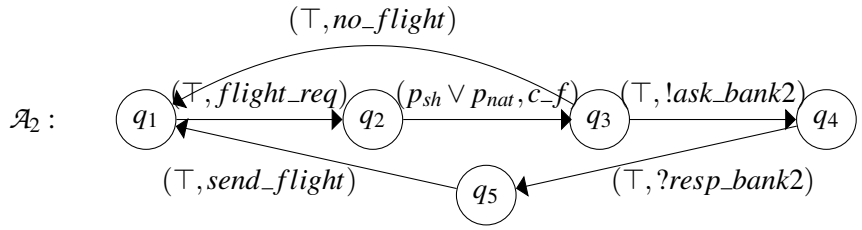


FIG. 4.4 – Automate booléen $\mathcal{A}_2$

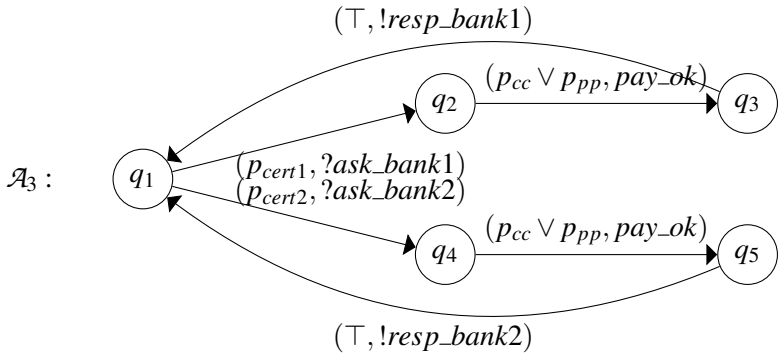


FIG. 4.5 – Automate booléen $\mathcal{A}_3$

$\mathcal{A}_2$ de la figure 4.4. Son fonctionnement est similaire à celui du Service S_1 mais en ne vendant des tickets qu'aux personnes n'ayant pas besoin de visa pour l'Espagne, donc soit des gens d'une certaine nationalité (modélisée par le prédicat p_{nat}), ou dont le point de départ est dans l'espace Shengen (prédicat p_{sh}).

Le service **service** S_3 de la banque est modélisé par l'automate $\mathcal{A}_3$ de la figure 4.5.

Ce service n'accepte que les requêtes de services ayant un certificat de sécurité (le prédicat p_{cert_i} code que le Service i possède un tel certificat). Dans ce cas, si l'acheteur a une carte de crédit (prédicat p_{cc}) ou un compte paypal (prédicat p_{pp}), et si le paiement est validé (ce que l'on code juste, de façon simplifiée, par pay_ok), la banque valide la transaction.

On souhaite composer ces trois services afin d'obtenir le service S modélisé par l'automate de la figure 4.6.
Plus précisément, on peut se demander :

1. Pour une valuation donnée des prédicats, les services S_1, S_2 et S_3 peuvent-ils se composer afin de fournir le service S ?
2. Existe-t-il une valuation des prédicats pour laquelle les services S_1, S_2 et

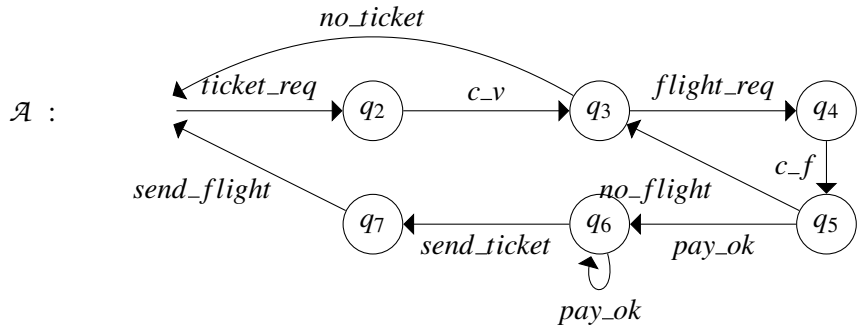


FIG. 4.6 – Automate $\mathcal{A}$

S_3 peuvent se composer afin de fournir le service S ? (appelé problème de *Valuation*)

3. Étant donné une formule booléenne ψ, est-ce que les services S_1, S_2 et S_3 peuvent se composer pour une valuation σ des prédicats afin de fournir le service S ssi $\psi(\sigma)$ est vraie ? (appelé problème de *Décision modulo une formule*)

4. Calculer une formule booléenne ψ telle que les services S_1, S_2 et S_3 peuvent se composer pour une valuation σ des prédicats afin de fournir le service S ssi $\psi(\sigma)$ est vraie ? (appelé problème *Synthèse d'une formule caractéristique*).

4.3.2 Résultats

Nous ne fournissons pas la formalisation précise de ces problèmes qui demanderait beaucoup de place pour des définitions intuitives : la sémantique des automates est donnée par les gardes qui sont des formules booléennes. La composition est un produit synchronisé par rendez-vous : lorsque deux actions de communication se synchronisent, elles produisent une transition ε dans le produit. La synchronisation des gardes se fait par une conjonction des formules booléennes. Les comparaisons modélisant le fait de *fournir un service* sont modélisées soit par l'inclusion de trace, soit par l'égalité des traces, soit par la simulation, soit par la bisimulation. Nous avons obtenus les résultats suivants (les bornes supérieures s'obtiennent facilement ; les bornes inférieures demandent des codages plus complexes).

Relation	valuation	Décision modulo une formule
Égalité des trace	*EXPSPACE*-complet	*EXPSPACE*-complet
Inclusion des traces	*EXPSPACE*-complet	*EXPSPACE*-complet
Simulation	*EXPTIME*-complet	*EXPTIME*-complet
Bisimulation	*PSPACE*-dur *EXPTIME*	*PSPACE*-dur *EXPTIME*

On a aussi montré que le problème de synthèse d'une formule caractéristique est résoluble en espace exponentiel pour les quatre relations de comparaisons étudiées.

4.4 Idéaux de mélange

Les idéaux de mélange forme une classe de langages réguliers intéressants dans le cadre de l'étude des systèmes communicants.

Rappelons la définition du produit de mélange, dont une utilité a été vue au chapitre 3 : étant donnés deux mots u et v sur un alphabet A, le produit de mélange de u par v est le langage des mots de la forme $u_1v_1 \dots u_kv_k$ tels que $u = u_1 \dots u_k$ et $v = v_1 \dots v_k$; c'est exactement le produit de $(A \times A)$-mélange, selon la définition du chapitre 4. Le produit de mélange de u par v est simplement noté $u \sqcup\!\sqcup v$.

Un idéal de mélange est un langage L qui vérifie $L \sqcup\!\sqcup A^* = L$. Ces langages sont étroitement liés à la notion de sous-mot : u est un sous mot de v si $v \in \{u\} \sqcup\!\sqcup A^*$. Un idéal de mélange est donc un langage clos par le haut pour la relation sous-mot. La notion de sous-mot ainsi que les langages clos par le haut (ou par le bas), jouent un rôle singulier dans l'étude des systèmes communicant par canaux à perte (voir par exemple [Fin94, AJ96, CS07]). La relation de sous-mot est une relation d'ordre partielle bien structurée : il n'existe pas de suite strictement croissante infinie de mots [Hig52, Kru72]. Une conséquence directe de cette propriété est que les idéaux de mélange sont finiment engendrés. Notons que les systèmes bien structurés sont particulièrement intéressants dans le cadre de l'analyse d'accessibilité des systèmes infinis [FS01, HMR05, FGL09].

Dans [19], nous avons étudié différentes constructions relatives aux idéaux de mélange. Le résultat le plus notable est un résultat de type *state complexity* : il existe un réel $r > 1$ et une suite $\mathcal{A}_n$ d'automates minimaux tels que pour tout n, $\mathcal{A}_n$ a n états et l'automate minimal l'idéal de mélange engendré par $L(\mathcal{A}_n)$ possède plus de $r^{\sqrt{n}}$ états.

4.5 Conclusion et perspectives

Les systèmes composés ont fait l'objet de nombreux travaux mais restent un sujet d'étude majeur : les méthodes de conception d'applications informatiques s'orientent de plus en plus vers des développements modulaires (les composants) éventuellement déployés sur des plates-formes distantes. Les systèmes embarqués, les réseaux de capteurs, les services Web, les réseaux Ad'hoc, etc, sont des domaines très dynamiques proposant des singularités qu'il va falloir prendre en compte dans les modèles (et dans les techniques de vérification). Nous avons contribué aux travaux de modélisations dans ce cadre en proposant une notion de substitutivité qualitative et quantitative et en étudiant les problèmes de complexité inhérants. Nous nous sommes aussi intéressés au problème de composition pour des services modélisés par des automates paramétrés. Dans ces directions, de nombreux travaux restent à faire sur des modèles plus précis/expressifs que les automates finis : dans ce cadre, les problèmes seront probablement indécidables et une approche semi-algorithmique pourrait être une voie de recherche intéressante.

Plus précisément, plusieurs pistes de travail restent à développer dans la suite de nos travaux, notamment sur la prise en compte de l'aspect quantitatif dans la vérification.

- D'autres façons de considérer les coûts pourraient être intéressantes : par exemple, calculer des coûts moyens au lieu de coût maximum. Il serait aussi possible de s'approcher d'un modèle de coût très utilisé en économie : plus une action aura lieu tardivement, moins son coût aura de l'importance ; le modèle d'automate correspondant est appelé *Weighted automata with discount* [KM08, DSV08, DSV08].
- Quelque soit la notion de coût utilisée, il serait intéressant d'étudier d'autres propriétés que la substitutivité et sur des modèles plus riches que les automates de mot. Dans ce sens de nombreux travaux ont déjà été menés sur les automates temporisés.
- Sur un plan algorithmique il est souvent indécidable de vérifier des propriétés tenant compte d'aspects quantitatifs. Dans ce cadre, le développement d'heuristiques ou de semi-algorithmes dédiés pourrait être intéressant. Une première étude dans ce sens sera présentée au chapitre 6.

Chapitre 5

Test aléatoire

Sommaire

5.1 Tester les systèmes 82
5.1.1 Introduction 82
5.1.2 Le test aléatoire 83
5.2 Utilisation d'une génération aléatoire d'automates pour le test 84
5.2.1 Test sur les entrées pour la correction 84
5.2.2 Utilisation dans un cadre de test à partir de modèles . . . 85
5.3 Génération aléatoire d'automates d'arbre 90
5.3.1 Transducteurs lettre-à-lettre 90
5.3.2 Génération aléatoire de transducteurs 91
5.3.3 Application aux automates d'arbre 92
5.4 Génération aléatoire équiprobable de structures récursives : outil Seed 93
5.4.1 Test à partir de grammaires 93
5.4.2 Génération de structures récursives avec l'outil Seed . . . 94
5.4.3 Exemple : réduction de formules LTL 95
5.5 Conclusion et Perspectives 96

Les publications liées à ce chapitre sont [17, 26] ainsi que l'article en soumission [25].

Nous nous intéressons dans ce chapitre à la validation de systèmes par des techniques aléatoires, éventuellement combinées avec d'autres approches de test.

Section 5.1. Cette section présente brièvement quelques généralités sur le test, notamment sur le test aléatoire.

Section 5.2. Nous présentons dans cette section les travaux de [17] : on montre comment un algorithme de génération aléatoire d'automates finis développé dans [BN07] peut être utilisé dans un cadre de test à partir de modèles.

Section 5.3. Nous montrons dans cette section comment générer uniformément des automates d'arbres (de haut en bas et cheminant), déterministes et accessibles. Nous montrons un exemple d'application pour l'évaluation d'algorithmes. Ces travaux ont fait l'objet de la publication [26].

Section 5.4. Dans cette section, nous abordons brièvement la description d'un outil pour la génération aléatoire uniforme de structures récursives. Nous donnons quelques pistes d'applications dans le cadre du test. Ce travail est actuellement en cours de soumission [25].

5.1 Tester les systèmes

5.1.1 Introduction

Nous avons vu aux chapitres précédents des techniques de vérification. Cependant, ces approches sont limitées soit par des problèmes théoriques (indécidabilité), soit par des problèmes algorithmiques (la complexité est trop importante). Dans ce dernier cas, les techniques ne s'appliquent qu'à des systèmes de taille modérée, comme les protocoles de communication, ou sur des abstractions des systèmes (comme pour les composants). L'approche de validation par le test est complémentaire de l'approche par vérification : si les garanties sont théoriquement moins bonnes, les techniques de test s'appliquent sur les systèmes réels ou sur des modèles beaucoup plus précis.

Il existe de très nombreuses techniques de tests selon l'approche (comment tester ?), selon les garanties que l'on souhaite (quels tests faire ? et pourquoi ?), et ce que l'on veut tester (quoi tester ?). Les approches diffèrent aussi selon ce dont on dispose pour tester : le code est il accessible ? Peut-on le modifier ? Que peut on observer ? Quel est le coût d'un test ? Dans leur conception, les techniques de test s'appuient fréquemment sur la notion de *critère de couverture* qui garantit, modulo ce critère, l'exhaustivité de la campagne de test. Par exemple, lors du test d'un programme, on peut poser comme critère de couverture que chaque instruction doit être exécutée au moins une fois lors de la campagne. Il existe une variété importante de critères de couverture :

certains, propres au test à partir de modèles, seront décrits plus loin.

5.1.2 Le test aléatoire

L'utilisation de l'aléatoire pour le test a été introduite initialement dans [Ham94]. Comme tester comporte une phase de choix, l'approche aléatoire consiste à faire des choix à l'aide de générateurs pseudo-aléatoires. Les avantages du test aléatoire sont les suivants :

1. Il est facile de générer aléatoirement des données numériques : tous les langages de programmation proposent des primitives efficaces de génération pseudo-aléatoire.
2. Le test aléatoire est en pratique très efficace, notamment pour les phases préliminaires de test, lorsque les objectifs ne sont pas trop précis.
3. Le test aléatoire permet de s'affranchir de la subjectivité de l'ingénieur validation, qui peut conduire à ne tester que les parties cruciales du code que les développeurs ont justement pris soin de très bien faire.
4. Le test aléatoire est particulièrement efficace pour faire du test de performances, car il est fréquent de ne pas disposer de benchmark.

Cependant, le test aléatoire présente les inconvénients suivants.

1. Le test aléatoire n'est pas adapté, de façon directe, à des campagnes de tests avec des objectifs précis : les comportements que l'on souhaite tester peuvent apparaître très rarement et seront *manqués* par les séquences aléatoires.
2. Pour garder toutes ces qualités, il est important que le test aléatoire se fasse selon une distribution connue, si possible uniforme ; dans ce cadre la génération d'objets non numériques devient beaucoup plus difficile.
3. Les approches aléatoires se combinent mal avec les approches visant à optimiser le nombre de tests, ce qui est important pour les systèmes pour lesquels les tests sont très coûteux.

Pour le premier point, il est possible de combiner le test aléatoire avec d'autres techniques, comme dans [HC83] ou [GDG+08]. Mais cela peut déboucher sur des problèmes similaires au second point : générer aléatoirement et uniformément des objets respectant certaines contraintes.

Il faut noter que la génération aléatoire a fait l'objet de nombreux travaux et que des théories puissantes sont à notre disposition. Nous allons montrer comment les utiliser dans le cadre du test. A notre connaissance, les seuls travaux dans ce sens actuellement sont [GDG+08, DGG+06] et [HC83] qui s'appuie sur [McK97].

Les travaux présentés ici visent à améliorer les points 1. et 2. des inconvénients en montrant comment utiliser la génération aléatoire pour le test soit en combinant avec d'autres techniques, soit en orientant les tests.

5.2 Utilisation d'une génération aléatoire d'automates pour le test

Nous nous appuyons dans cette section sur les travaux [BN07], implanter dans [BDN07]. Ces travaux proposent une méthode et un outil pour générer aléatoirement des automates finis (de mot) déterministes complets et accessibles, en temps $O(n^{3/2})$ où n est le nombre d'états de l'automate généré.

5.2.1 Test sur les entrées pour la correction

Nous nous sommes tout d'abord intéressés au générateur [BN07] afin de tester, de façon simple, une implantation d'un algorithme de résolution du problème du postier chinois. Il s'agit là d'un version très simple du test : on choisit aléatoirement et sans aucune orientation des données d'entrée.

Postier chinois
Entrée : Un graphe orienté étiqueté fini G
Question : Trouver un chemin de taille minimale passant au moins une fois par chaque arête de G.

Ce problème, dual du voyageur de commerce, est polynomial [EJ73, LZ88]. L'implantation testée est celle de [Thi03] librement téléchargeable[1]. Pour cela, nous avons généré aléatoirement des automates finis de tailles variées, puis nous avons lancé l'implantation.

En générant 8 automates avec 30 états sur un alphabet à 20 lettres, on a découvert un automate provoquant un arrêt défectueux du programme. En recherchant dans le code, nous avons trouvé une erreur de dépassement tableau. Après correction de cette erreur, on a effectué plusieurs tests récapitulés dans le tableau 5.1 :

- La colonne *États* donne le nombre d'états de l'automate généré,
- La colonne *alphabet* la taille de l'alphabet,
- La colonne *test* le nombre d'automates générés,
- La colonne *(1) erreur* le nombre de tests pour lesquels le programme plante,
- La colonne *(2) erreur* le nombre de tests pour lesquels la nouvelle implantation plante,
- La colonne *(2) est mieux* le nombre de tests pour lesquels la nouvelle implantation donne un chemin plus court que l'implantation originale,
- La colonne *(1) est mieux* le nombre de test pour lesquels l'ancienne implantation donne un chemin plus court que la nouvelle implantation.

Comme on peut l'observer, aucun test ne provoque d'arrêt défectueux de la nouvelle implantation et ses résultats sont meilleurs que l'ancienne.

[1] `http://www.uclic.ucl.ac.uk/harold/cpp/`

états	alphabet	tests	(1) erreur	(2) erreur	(2) est mieux	(1) est mieux
17	3	10	0	0	1	0
30	5	8	0	0	3	0
30	20	8	1	0	1	0
30	5	10	1	0	2	0
100	5	10	0	0	1	0

TAB. 5.1 – Résultats de Tests pour deux implantations résolvant le problème du postier chinois.

5.2.2 Utilisation dans un cadre de test à partir de modèles

Le test à partir d'automates finis

Le test à partir de modèles vise principalement à confronter une implantation à son modèle théorique : on parle de test de conformité. L'objectif étant répondre à la question : "est-ce que l'implantation correspond bien au modèle ?". C'est une démarche complémentaire de la *vérification* qui prouve que les modèles sont corrects (vis-à-vis des spécifications). Il existe de nombreuses techniques de test à partir de modèles qui dépendent notamment de la définition de *conformité* ainsi que du modèle choisi ; le lecteur intéressé pourra se référer à [MBTS04, LU07].

Nous nous restreignons dans ce document aux systèmes modélisés par des automates finis. Dans ce cadre, le principe du test à partir de modèles consiste à choisir des chemins de l'automate, à les exécuter, puis à comparer ces exécutions sur le système avec leur comportement théorique sur le modèle. Nous nous ne nous intéresserons pas à ces deux dernières phases, appelées *concrétisation* et *observation* qui soulèvent de nombreuses problématiques.

Le défi du test à partir du modèles est le suivant : plus on choisit de tests plus, a priori, la campagne de test sera de qualité. Plus elle sera coûteuse aussi. Tout est donc une question de compromis et il n'y a pas qu'une bonne réponse : selon le contexte, l'importance de la qualité varie, de même que le coût d'un test.

Formellement, un *critère de couverture* est une application π qui à un automate fini et à tout ensemble de chemins de cet automate associe 0 ou 1. Étant donné un automate fini, un ensemble de chemins réussis de cet automate et un critère de couverture, on dira que cet ensemble de chemins réussis satisfait le critère de couverture si son image par ce critère vaut 1.
Classiquement, on définit les critères de couvertures suivants.

- Le critère φ_{etats} défini par : un ensemble de chemins réussis Π d'un automate fini satisfait le critère φ_{etats} si tout état de l'automate est visité au moins une fois par un chemin de Π. Ce critère est couramment appelé *Tous les états*.

- Le critère $\varphi_{\text{transitions}}$ défini par : un ensemble de chemins réussis Π d'un automate fini satisfait le critère $\varphi_{\text{transitions}}$ si toute transition de l'automate est visité au moins une fois par un chemin de Π. Ce critère est couramment appelé *Toutes les transitions*.
- Le critère $\varphi_{2-\text{trans}}$ défini par : un ensemble de chemins réussis Π d'un automate fini satisfait le critère $\varphi_{2-\text{trans}}$ si tout couple de transitions du type $(p,a,q),(q,b,r)$ (transitions consécutives) de l'automate est visité au moins une fois, de façon consécutive, par un chemin de Π. Ce critère est couramment appelé *Toutes les transitions consécutives*.
- Le critère φ_{boucles} définit par : un ensemble de chemins réussis Π d'un automate fini satisfait le critère φ_{boucles} si toute boucle simple est visitée au moins une fois par un chemin de Π. Ce critère est couramment appelé *Toutes les boucles simples*.
- Il existe toute une zoologie de critères que nous ne préciserons pas ici.

Les critères *Tous les états*, *Toutes les transitions* et *Toutes les transitions consécutives* peuvent être générés en temps polynomial (pas nécessairement de façon optimale). Les spécialistes du test considèrent que le critère *Tous les états* est trop faible pour le test de conformité car il ne couvre pas assez le modèle et que le critère *Toutes les transitions* est le minimum à faire.

Notre objectif est de combiner ces critères de couverture avec des aspects aléatoires. On s'appuie pour cela sur la proposition suivante.

Proposition 5.1 *Soit $\mathcal{A}$ un automate dans lequel tous les états sont accessibles et co-accessibles. Soit $x \in \{etats, transitions, 2-trans\}$. Si $\mathcal{B}$ est un automate complet et accessible dont tous les états sont finaux et si $\varphi_x(\mathcal{A}\times\mathcal{B},\Pi) = 1$, alors $\varphi_x(\mathcal{A},\Pi') = 1$. Si $\mathcal{B}$ est complet et fortement connexe et si $\varphi_{boucles}(\mathcal{A}\times\mathcal{B},\Pi) = 1$, alors $\varphi_{boucles}(\mathcal{A},\Pi') = 1$. Dans ces deux implications, Π' représente le projeté de Π dans l'ensemble des chemins de $\mathcal{A}$.*

Approche aléatoire

Notre objectif est de définir une procédure de test comportant une phase aléatoire satisfaisant un critère de couverture donné et pour laquelle le nombre de tests est contrôlé.

Nous la décrivons pour le critère *toutes les transitions*, mais elle s'adapte facilement aux autres critères de la proposition 5.1 ; la technique de génération de tests est la suivante :

1. On suppose que l'on a au départ un automate $\mathcal{A}$ contenant m états finaux codant le système que l'on souhaite tester. On suppose aussi que l'on veut faire de l'ordre de k tests.
2. On génère un automate déterministe complet accessible $\mathcal{B}$ à k/m états (et dont tous les états sont finaux) en utilisant [BN07].

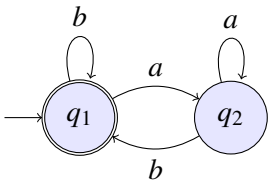


FIG. 5.1 – Exemple de système à tester

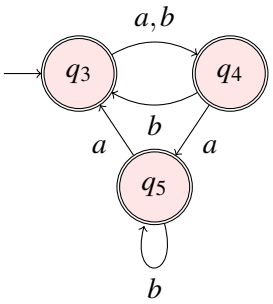


FIG. 5.2 – Exemple d'automate généré

3. On calcule un ensemble de chemins Π de $\mathcal{A} \times \mathcal{B}$ satisfaisant le critère *toutes les transitions* en utilisant un algorithme de postier chinois.

4. La séquence de tests est la projection de Π dans les chemins de $\mathcal{A}$.

Illustrons l'approche sur l'exemple de la figure 5.1. Si l'on exécute directement l'algorithme du postier chinois sur cet exemple, on obtient un seul chemin, par exemple le chemin d'étiquette *baab*. Supposons que l'on souhaite faire 3 tests, couvrant toutes les transitions. On génère alors un automate $\mathcal{B}$ à 3 états, par exemple celui de la figure 5.2.

On calcule alors l'automate $\mathcal{A} \times \mathcal{B}$ (représenté figure 5.3). Sur ce produit, on applique l'algorithme du postier chinois. On obtient, par exemple, les chemins étiquetés par *aabbaab* et *baabb*. Les chemins correspondants dans $\mathcal{A}$ sont les suites de tests de la campagne.

On peut remarquer que l'algorithme proposé est polynomial, la génération est donc rapide. On peut se demander pourquoi générer un automate de taille k/m. La réponse est purement expérimentale : en générant de nombreux automates, on s'aperçoit que le nombre de tests (en utilisant le critère *toutes les transitions*), est à peu près égal au nombre d'états finaux de l'automate. Comme $\mathcal{A} \times \mathcal{B}$ possède $(k/m) * m$ états finaux, le nombre de tests sera de l'ordre de k.

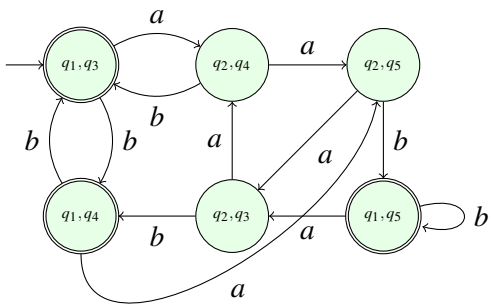


FIG. 5.3 – Produit des automates

Résultats expérimentaux

Demoney est une spécification d'un porte-monnaie électronique développé par Trusted Logics [MM01] dans le but d'expérimenter des techniques de test ou de vérification. Pour ce porte-monnaie, on dispose d'une spécification en B d'une partie de la spécification globale et d'une implantation en Java de cette partie.

Grâce à des logiciels [BLP04] développés au Laboratoire de l'Université de Franche-Comté, nous avons pu évaluer[2] notre approche sur ce cas d'étude. A partir du modèle B, nous avons pu abstraire (à la main), un automate fini modélisant le porte-monnaie. Nous avons fait différentes campagnes de tests, en faisant varier le nombre d'états finaux du modèle du porte-monnaie et le nombre de tests souhaités. Afin d'évaluer la performance de l'approche, nous avons introduit dans l'implantation des *mutants*, c'est-à-dire des erreurs volontaires, et nous avons compté combien de ces mutants étaient tués (i.e. combien de ces erreurs étaient détectées).

Les résultats sont résumés dans la figure 5.4. Dans ce tableau, on indique
- Dans la colonne *Suite de Tests* le nom de la suite de tests générée,
- Dans la colonne *# tests* le nombre de tests effectivement générés,
- Dans la colonne *long. moy.* la longueur moyenne des tests,
- Dans la colonne *long. max.* la longueur maximale des tests
- Dans la colonne *temps* le temps de génération et,
- Dans la colonne *mutants tués* le nombre de mutants tués.

Pour chaque bloc, la première ligne représente le nombre de tests générés par simple application du postier chinois. Il faut noter que les différences proviennent du fait que, pour chaque état final, on ajoute une transition *reset* de cet état vers l'état initial. Les automates ne sont donc plus les mêmes.

[2]Ces outils permettent la concrétisation des tests et leur observation.

Le bloc *ChineseAug2F_10L* (avec $L \in \{a, \ldots, e\}$) montre 5 résultats de l'approche avec pour objectif d'atteindre 10 tests, et avec, dans la spécification initiale, 2 états finaux.

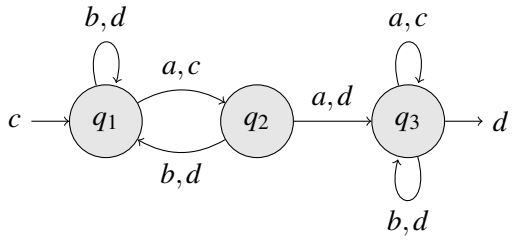
Suite de Tests	# tests	long. moy.	long. max.	temps	mutants tués
ChinesePostman2F	3	91	175	3.8s	24/31 (77%)
ChineseAug2F_10a	9	126	618	25.23s	28/31 (90%)
ChineseAug2F_10b	9	109	547	25.32s	27/31 (87%)
ChineseAug2F_10c	9	127	632	32.94s	27/31 (87%)
ChineseAug2F_10d	9	121	668	24.88s	28/31 (90%)
ChineseAug2F_10e	9	117	618	26.26s	28/31 (90%)
ChineseAug2F_12	13	135	929	2min18s	31/31 (100%)
ChineseAug2F_15	15	130	973	5min32s	31/31 (100%)
ChineseAug2F_18	19	133	1388	18min38s	31/31 (100%)
ChineseAug2F_20	21	125	1264	23min28s	31/31 (100%)
ChineseAug2F_25	25	132	1675	1h1min28s	31/31 (100%)
ChinesePostman4F	5	61	180	3.8s	24/31 (77%)
ChineseAug4F_10	9	74	390	8.8s	25/31 (80%)
ChineseAug4F_12	13	73	517	15.5s	31/31 (100%)
ChineseAug4F_15	13	68	495	13.4s	31/31 (100%)
ChineseAug4F_18	17	71	573	31.8s	31/31 (100%)
ChineseAug4F_20	21	72	771	1min24s	31/31 (100%)
ChineseAug4F_25	25	74	869	2min12s	31/31 (100%)
ChinesePostman5F	6	52	121	4.1s	24/31 (77%)
ChineseAug5F_10	11	48	124	6.5s	25/31 (80%)
ChineseAug5F_12	11	48	124	6.6s	25/31 (80%)
ChineseAug5F_15	16	55	206	12.9s	31/31 (100%)
ChineseAug5F_18	16	53	200	11.6s	31/31 (100%)
ChineseAug5F_20	21	55	220	31.7s	31/31 (100%)
ChineseAug5F_25	26	58	403	1min10s	31/31 (100%)
LTG	59	2.66	8	2min 34s	19/31 (61%)

FIG. 5.4 – Résultat de l'approche utilisant de la génération aléatoire d'automates

Plus généralement, les lignes *ChineseAugNF_S* représentent l'application de notre approche pour obtenir S tests et où l'automate initial à N états finaux.

La dernière ligne représente les résultats obtenus avec l'outil industriel *LTG* de l'entreprise Smartesting[3]. Cet outil ne s'appuie que sur le modèle B et peut donc être utilisé sans la difficile étape (faite ici à la main) d'abstraire le modèle B en automate fini.

[3] `http://www.smartesting.com`



$$
\begin{array}{llllll}
\delta(q_1,a) & = q_2 & \delta(q_2,a) & = q_3 & \delta(q_3,a) & = q_3 \\
\delta(q_1,b) & = q_1 & \delta(q_2,b) & = q_1 & \delta(q_3,b) & = q_3 \\
\gamma(q_1,a) & = c & \gamma(q_2,a) & = d & \gamma(q_3,a) & = c \\
\gamma(q_1,b) & = d & \gamma(q_2,b) & = d & \gamma(q_3,b) & = d \\
 & & & & \rho(q_3) & = d \\
a_{\text{init}} & = c & & & &
\end{array}
$$

FIG. 5.5 – Un transducteur séquentiel lettre-à-lettre.

Les résultats montrent que la technique permet de tuer tous les mutants dès que l'on demande 12 tests. On peut aussi remarquer que le nombre de tests générés est à chaque fois proche du nombre de tests demandés.

5.3 Génération aléatoire d'automates d'arbre

Nous avons vu dans la partie précédente une utilisation possible de la génération d'automates finis pour le test. Nous montrons ici comment générer aléatoirement et uniformément des automates d'arbre de haut en bas (*top-down*) ainsi que des automates d'arbre cheminant (*tree walking automata*).

L'approche générale est la suivante : on montre comment, à partir de [BN07], générer des transducteurs lettre-à-lettre déterministes. Ensuite, on établit des bijections entre ces transducteurs et les modèles d'automates déterministes de haut en bas et cheminant.

5.3.1 Transducteurs lettre-à-lettre

Un *transducteur séquentiel lettre-à-lettre* d'un alphabet Σ_1 vers un alphabet Σ_2 est un tuple $\mathcal{T} = (\Sigma_1, \Sigma_2, Q, q_{\text{init}}, \delta, \gamma, \rho, a_{\text{init}})$ où Q est un ensemble fini d'*états*, $q_{\text{init}} \in Q$ est l'*état initial*, δ est une fonction de $Q \times \Sigma_1$ dans Q appelée *fonction de transition*, γ est une fonction de $Q \times \Sigma_1$ dans Σ_2 telle que δ et γ aient le même domaine (γ est appelée *fonction de sortie*, ρ est une fonction partielle de Q dans Σ_2 appelée *fonction de fin*, et $a_{\text{init}} \in \Sigma_2$ la *sortie initiale*. Un transducteur séquentiel lettre-à-lettre est *complet* si le domaine de δ est $Q \times \Sigma_1$. Les états *accessibles* sont classiquement définis par : q_{init} est accessible et si q est accessible, pour chaque $a \in \Sigma_1$, $\delta(q,a)$ est accessible. Un transducteur séquentiel lettre-à-lettre est *accessible* si tous ses états sont accessibles. Un exemple est donné dans la figure 5.5.

Un isomorphisme de transducteur séquentiel lettre-à-lettre est une bijection entres les ensembles des états qui préserve les structures (états initiaux, transitions, etc.). Nous allons voir maintenant comment générer aléatoirement, à isomorphisme près, des transducteurs séquentiels lettre-à-lettre accessibles.

5.3.2 Génération aléatoire de transducteurs

Familles de transducteurs

Considérons les familles $\mathcal{D}_n(\Sigma_1,\Sigma_2,r,r_i,r_F)$ des transducteurs séquentiels lettre-à-lettre accessibles, où Σ_1 est l'alphabet d'entrée, Σ_2 est l'alphabet de sortie, $r:\Sigma_1\to 2^{\Sigma_2}$ est la fonction de restriction de l'alphabet, $r_i\in 2^{\Sigma_2}$ est la restriction sur l'initialisation et $r_F\in 2^{\Sigma_2}$ est la restriction sur les finalisations. Un transducteur séquentiel lettre-à-lettre accessible $(\Sigma_1,\Sigma_2,Q,i,\delta,\gamma,\rho,a_i)$ appartient à $\mathcal{D}_n(\Sigma_1,\Sigma_2,r,r_i,r_F)$ si les conditions suivantes sont satisfaites :

- (i) $a_i\in r_i$,
- (ii) $\rho(Q)\subseteq r_F$,
- (iii) Q est de cardinal n et,
- (iv) pour tout $a\in\Sigma_1$, $\gamma(Q,a)\subseteq r(a)$.

On note par $\mathcal{C}_n(\Sigma_1,\Sigma_2,r,r_i,r_F)$ le sous-ensemble $\mathcal{D}_n(\Sigma_1,\Sigma_2,r,r_i,r_F)$ qui contient tous les transducteurs complets. Afin de générer aléatoirement un élément de $\mathcal{D}_n(\Sigma_1,\Sigma_2,r,r_i,r_F)$ ou de $\mathcal{C}_n(\Sigma_1,\Sigma_2,r,r_i,r_F)$, on décompose le problème en trois : générer le graphe sous-jacent étiqueté uniquement avec les symboles d'entrée, générer les sorties sur les transitions, puis générer les états finaux. Pour les transducteurs complets, on peut montrer que ces trois parties peuvent êtres faites indépendamment tout en préservant l'uniformité.

Algorithmes de génération

Pour générer un automate de $\mathcal{C}_n(\Sigma_1,\Sigma_2,r,r_i,r_F)$, on procède comme suit.

1. On génère un automate déterministe complet accessible sur Σ_1 en utilisant [BN07].
2. Pour chaque $q\in Q$ et chaque $a\in\Sigma_1$, on choisit uniformément et aléatoirement $\gamma(q,a)$ dans $r(a)$.
3. Pour chaque $q\in Q$, on choisit aléatoirement un élément x de $r_F\uplus\{\#\}$, où $\#$ est un nouveau symbole indiquant que l'état n'est pas final. On définit alors $\rho(q)=x$ si $x\neq\#$ et sinon $\rho(q)$ est non-défini.

Notons que l'on peut aussi paramétrer le nombre d'états finaux par f ; dans ce cas la troisième étape est remplacée par :

3.bis Choisir aléatoirement un sous-ensemble F de Q de cardinal f et, pour chaque q dans F, choisir aléatoirement $\rho(q)$ dans r_F.

Pour générer des transducteurs de $\mathcal{D}_n(\Sigma_1,\Sigma_2,r,r_i,r_F)$, on ordonne arbitrairement Σ_2 et on modifie la procédure comme suit (la modification est indiquée en italique).

1. On génère un automate déterministe (pas nécessairement complet) accessible sur Σ_1 en utilisant [BDN09].

2. Pour chaque $q \in Q$ et chaque $a \in \Sigma_1$, on choisit uniformément et aléatoirement $\gamma(q,a)$ dans $r(a)$. *Si $\delta(q,a)$ n'est pas défini et si $\gamma(q,a)$ n'est pas minimal, recommencer à l'étape 1.*
3. Pour chaque $q \in Q$, on choisit aléatoirement un élément x de $r_F \uplus \{\#\}$, où $\#$ est un nouveau symbole indiquant que l'état n'est pas final. On définit alors $\rho(q) = x$ si $x \neq \#$ et sinon $\rho(q)$ est non-défini.

Il faut noter que l'étape 2 est une étape utilisant un rejet. Par cette méthode, on garantit l'uniformité de la génération en sortie. En revanche, la procédure peut boucler indéfiniment. Mais l'on peut montrer que cela n'arrive qu'avec une probabilité nulle et que le nombre moyen de rejets est constant.

En s'appuyant sur des résultats théoriques de [BDN09], on peut prouver la proposition suivante.

Proposition 5.2 *Soit E_n un sous-ensemble de $\mathcal{D}_n(\Sigma_1, \Sigma_2, r, r_i, r_F)$ tel que*

$$\mathcal{C}_n(\Sigma_1, \Sigma_2, r, r_i, r_F) \subseteq E_n.$$

L'algorithme par rejet consistant à générer un élément de $\mathcal{D}_n(\Sigma_1, \Sigma_2, r, r_i, r_F)$ jusqu'à ce qu'il soit dans E_n demande un nombre moyen de rejets constant.

5.3.3 Application aux automates d'arbre

Grâce à la proposition 5.2, nous avons pu mettre au point des algorithmes pour générer aléatoirement des automates d'arbre déterministes accessibles de haut en bas [CDG+02], ainsi que des automates d'arbre cheminant (*tree walking automata*) [Boj08] déterministes et accessibles. Nous ne détaillerons pas les aspects techniques ; l'approche consiste à établir des bijections entre ces automates et des classes restreintes d'automates de transducteurs séquentiels lettre-à-lettre.

Les automates d'arbre cheminant ont des liens importants avec des formalismes logiques incluant certains fragments XPATH [EH99, tCS08]. Dans ce cadre, la satisfiabilité d'une formule se réduit au test du vide d'un langage donné par un automate cheminant. Ce problème est EXPTIME-complet et les algorithmes connus consistent à traduire l'automate d'arbre cheminant en un automate d'arbre de haut en bas ayant potentiellement 2^{n^2} états, où n est le nombre d'états de l'automate cheminant initial. Nous avons implémenté l'algorithme de traduction d'un automate d'arbre cheminant vers un automate d'arbre de haut en bas, puis nous avons, par génération aléatoire, estimé le nombre moyen d'états de l'automate obtenu.

L'algorithme a été lancé, pour chaque n, sur 100 automates générés aléatoirement. La figure 5.6 donne, pour chaque n, la moyenne du nombre d'états pour les 10 valeurs les plus hautes, les 10 valeurs les plus faibles, et la moyenne pour les 80 autres. On observe sur cette figure une croissance en $\mathcal{O}(2^n)$ en

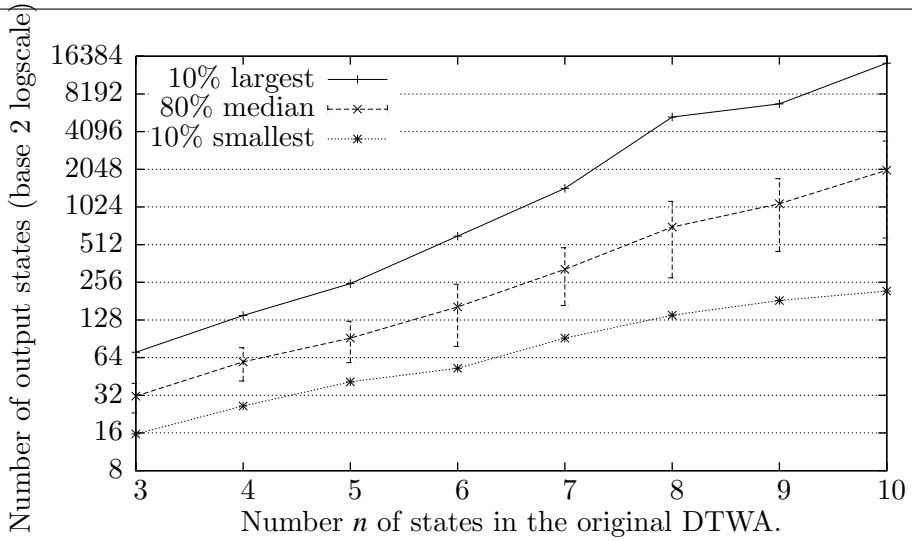


FIG. 5.6 – Nombre moyen d'états pour la transformation d'un automate d'arbre cheminant à n états en un automate d'arbre de haut en bas équivalent

moyenne du nombre des états, ce qui est significativement plus faible que la borne théorique de $O(2^{n^2})$.

5.4 Génération aléatoire équiprobable de structures récursives : outil SEED

Nous abordons dans cette section la génération aléatoire de structures récursives. Sur le plan théorique, les techniques utilisées sont celles de [NW78, FZC94]. La contribution porte sur la mise-en-œuvre de ces techniques pour le test au travers d'un outil facile d'utilisation appelé SEED[4]. Notre objectif est avant tout de réaliser un outil facile d'utilisation afin de mettre les techniques de génération aléatoire avancées au service du test.

5.4.1 Test à partir de grammaires

De nombreux systèmes (compilateurs, calculateurs, etc.) ont comme paramètre d'entrée des objets non-numériques (arbres, listes, formules arithmétiques ou logiques, etc.). Comme toute donnée, numérique ou non, est codée en binaire en machine, on pourrait être tenté de générer alétoirement des structures non numériques en générant le codage associé. En pratique, cela pose plusieurs problèmes : tout d'abord les codages ne sont pas injectifs, il sera alors difficile de produire une génération uniforme. Ensuite de nombreuses

[4] `http://monge.univ-mlv.fr/~nicaud/seed`

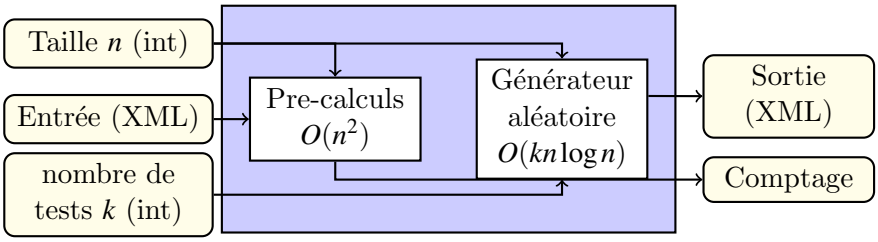


FIG. 5.7 – Architecture de l'outil SEED

suites de bits ne correpondent pas à une entrée valide (les codages ne sont pas surjectifs), la génération produira donc de nombreux déchets. Ce problème est encore accentué si l'on souhaite générer des données avec une strucutre satisfaisant des contraintes.

L'approche présentée ici consite à générér uniformement des structures arborescentes décrites par une grammaire.

L'approche [McK97, HC83] est assez similaire à la nôtre. Cependant, elle gère moins d'opérateurs, est moins efficace sur le plan algorithmique et n'est pas proposée dans un outil.

Dans [Mau92], l'auteur propose une méthodologie pour la génération aléatoire de tests à partir de spécifications sous forme de grammaires. Cependant, la technique est à la fois coûteuse et non uniforme. Une approche non aléatoire de test à partir de grammaires est proposée dans [CL05] par génération systématique d'entrées. A cause de l'explosion que cela induit, [LS06, MX07, GKL08] combinent l'approche à base de grammaire avec des approches symboliques. D'un point de vue plus applicatif, les travaux [Pur72] et [DDGM07] s'interessent respectivement aux tests de *parsers* définis par des grammaires et aux tests de *refactoring engines*, des programmes qui réécrivent d'autres programmes avec, là aussi, des spécifications sous forme de grammaires.

5.4.2 Génération de structures récursives avec l'outil SEED

En utilisant la métode récursive de génération aléatoire [NW78, FZC94], nous proposons un outil facile d'emploi permettant de générer des objets définis par des grammaires. La structure de l'outil est donnée dans la figure 5.7.

L'outil prend en entrée une spécification sous forme de grammaire pouvant contenir des opérateurs unaires, binaires, binaires symétriques, binaires idempotents, binaires symétriques et idempotents ou associatifs. Il est aussi possible de définir des variables afin de générer des formules du premier ordre. On définit aussi la taille n des objets générés et le nombre k d'objets que l'on souhaite générer. L'outil génère ensuite dans un fichier XML k structures arborescentes de taille n, uniformement et modulo les propriétés algébriques des

opérateurs. L'outil propose aussi une fonction qui compte le nombre d'objets de taille n. SEED est disponible[5] sous la forme d'une archive Java.

5.4.3 Exemple : réduction de formules LTL

Nous illustrons les possibilités de l'outil sur un problème de réduction de formules LTL [Pnu77].

De nombreux outils et travaux sont dédiés au model-checking de formules LTL comme, par exemple, [GO01, SB00, Hol97]. On considère ici, comme définition d'une formule LTL sur $\{a,b\}$ la grammaire suivante[6] :

$$L := a \mid b \mid \neg a \mid \neg b \mid L \vee L \mid L \wedge L \mid \circ L \mid \diamond L \mid \Box L \mid LUL \mid LRL.$$

Les formules générées par cette grammaire sont qualifiées de formules de *type 1*. On considère aussi les formules de *type 2* générées par la grammaire (le symbole initial est L) :

$$\begin{aligned} L &:= a \mid b \mid \neg a \mid \neg b \mid L \vee L \mid L \wedge L \mid \circ L \mid \diamond D \mid \Box S \mid LUL \mid LRL \\ D &:= a \mid b \mid \neg a \mid \neg b \mid L \vee L \mid L \wedge L \mid \circ L \mid \Box S \mid LUL \mid LRL \\ S &:= a \mid b \mid \neg a \mid \neg b \mid L \vee L \mid L \wedge L \mid \circ L \mid \diamond D \mid LUL \mid LRL \end{aligned}$$

Cette spécification interdit d'avoir deux symboles $\Box$ ou deux symboles $\circ$ consécutifs. De plus, on va spécifier dans l'outil que les symboles $\vee, \wedge$ sont symétriques et idempotents, c'est-à-dire que

- tout noeud de la formule (vu comme un arbre) étiqueté par un $\vee$ ou un $\wedge$ doit avoir deux sous arbres différents (idempotence) et,
- les formules $\varphi_1 \vee \varphi_2$ (resp. $\varphi_1 \wedge \varphi_2$) et $\varphi_2 \vee \varphi_1$ (resp. $\varphi_2 \wedge \varphi_1$) sont considérées comme les mêmes formules (symetrie).

De plus R et U sont considérés comme idempotents, c'est-à-dire que tout noeud de la formule étiqueté par un U ou un R doit avoir deux fils distincts. L'outil SEED permet une génération uniforme sous ces contraintes.

Pour chacune de ces formules, on a généré 1000 formules de taille 200. On a ensuite appliqué les règles de réductions classiques décrites dans la figure 5.8, en mesurant la taille des formules obtenues.

– Pour la spécification de type 1, les formules obtenues ont une taille moyenne de 161 (la taille la plus probable est de 190). Même avec ces chiffres, la proportion de formules irréductibles générées est très faible : on ne peut pas s'appuyer sur un algorithme de rejet pour obtenir des formules irréductibles.

[5] L'outil a été implémenté par C. Nicaud.

[6] Il s'agit là de la syntaxe courrament utilisée par les model-checkers.

$E \vee E \to E$
$E \vee \top \to \top$
$\Box\Box E \to \Box E$
$EU\top \to \top$
$ER\top \to \top$
$ER\top \to \top$
$(\circ E)U(\circ F) \to \circ(EUF)$
$\Box\diamond E \vee \Box\diamond F \to \Box\diamond(F \vee E)$
$\diamond\circ E \to \circ\diamond E$
$\diamond(E \wedge \Box\diamond F) \to (\diamond E) \wedge (\Box\diamond F)$
$\circ(E \wedge \Box\diamond F) \to (\circ E) \wedge (\Box\diamond F)$

$E \wedge E \to E$
$E \vee \bot \to E$
$\diamond\diamond E \to \diamond E$
$EU\bot \to \bot$
$ER\bot \to \bot$
$ER\bot \to \bot$
$\circ\top \to \top$

$E \vee \neg E \to \bot$
$E \wedge \bot \to \bot$
$\diamond\top \to \top$
$\top UE \to \diamond E$
$\top RE \to E$
$\top RE \to E$
$(\circ E) \wedge (\circ F) \to \circ E \wedge F$
$EU\Box\diamond F \to \Box\diamond F$
$\circ\Box\diamond E \to \Box\diamond E$
$\Box(E \vee \Box\diamond F) \to (\Box E) \vee (\Box\diamond F)$
$\circ(E \vee \Box\diamond F) \to (\circ E) \vee (\Box\diamond F)$

$E \wedge \neg E \to \top$
$E \wedge \top \to E$
$\diamond\bot \to \bot$
$\bot UE \to E$
$\bot RE \to \Box E$
$\bot RE \to \Box E$

FIG. 5.8 – Règles de simplifications de formules LTL

– Pour les formules de type 2, la taille moyenne est de 175. Cependant environ 8% des formules sont irréductibles : il est possible de générer des formules irréductibles par rejet.

Illustrons ce cas des formules LTL sur l'exemple d'application suivant. Fréquemment, les outils transformant les formules LTL en automates finis (pour le model-checking), se décomposent en deux phases. Tout d'abord la formule LTL est simplifiée en utilisant des règles de réécritures. Ensuite la formule est traduite en automate fini ; c'est la partie la plus lourde sur le plan algorithmique. Supposons, dans notre exemple, que l'on souhaite tester l'efficacité d'un model-checker (en boîte noire), à effectuer la seconde phase : traduire la formule LTL en automate fini. Dans ce cadre, générer des formules non irréductibles peut engendrer du *bruit* : en générant des formules de type 1, les résultats seront fortement biaisés par la qualité des réductions et l'on ne testera que l'ensemble des deux phases. En revanche, avec les formules de type 2, on peut en générer des formules irréductibles (par rejet). Même si l'on ne gère pas exactement les mêmes réductions que l'outil, le test de performance pour la seconde étape sera beaucoup plus pertinent.

5.5 Conclusion et Perspectives

Les travaux présentés dans ce chapitre s'articulent autour de la génération aléatoire pour le test. Nous nous sommes intéressés à la génération aléatoire d'automates d'arbre ainsi qu'à l'utilisation de la génération aléatoire d'automates finis dans un cadre de test à partir de modèle. L'activité de test suppose inéluctablement que l'on fasse des choix. Certains estiment que faire ces choix aléatoirement est la méthode du pauvre : comme on ne sait pas comment choi-

sir, on choisit au hasard. On peut cependant voir le choix aléatoire d'une autre façon : le choix aléatoire est parfois presque aussi bon qu'un choix calculé tout en étant extrêmement moins coûteux. La motivation des travaux présentés est d'obtenir, dans ce cadre, des générateurs performants et de qualité. Bien sûr, des validations expérimentales sont encore nécessaires afin de confirmer la pertinence de l'approche.

Plus précisément, les perspectives suivantes sont envisagées.

- Dans le cadre de la génération aléatoire d'automates finis, une bonne approche pour une génération d'automates non déterministes reste à développer.
- Les applications possibles de Seed pour le test aléatoire sont prometteuses : on pourrait s'intéresser à son utilisation pour du test à partir de modèles logico-algébriques [AAGL07, AAB+05, GG08].
- Enfin, des travaux récents combinant test aléatoire et test à partir de modèles ont été effectués dans [GDG+08] : les critères de couvertures sont probabilistes et les tests choisis aléatoirement. Dans ces travaux, les modèles de systèmes sont des automates finis. Comme il s'agit d'abstractions fortes, il serait intéressant d'étendre l'approche à des modèles plus riches, comme les automates à compteurs ou les automates temporisés, afin d'avoir plus de tests concrétisables.

Chapitre 6

Autres travaux sur l'analyse d'accessibilité

Sommaire

6.1 Calculs de noyaux abéliens 100
6.1.1 Introduction 100
6.1.2 Calcul du noyau abélien et accessibilité 101
6.1.3 Expérimentations et discutions 102
6.2 Comparaison d'automates max-plus 102
6.2.1 Graphe d'accessibilité d'un automate max-plus 103
6.2.2 Exploration du graphe de déterminisation 104
6.2.3 Encodage par réécriture 106
6.2.4 Expérimentations 107
6.2.5 Perspectives et discutions 107

Les publications liées à ce chapitre sont [18, 11].

Nous nous intéressons dans ce chapitre à deux problèmes d'accessibilité.

Section 6.1. Cette section est dédiée au problème du calcul du noyau abélien d'un monoïde fini. Nous présentons un algorithme polynomial pour ce problème [18], alors que le meilleur algorithme connu était exponentiel.

Section 6.2. Nous abordons ici d'un point de vue pratique le problème indécidable [Kro94] de la comparaison d'automates max-plus. Nous proposons une combinaison d'approches semi-algorithmiques efficaces en pratique pour répondre à ce problème [11].

6.1 Calculs de noyaux abéliens

6.1.1 Introduction

Le calcul de noyaux est un problème mathématique de théorie des monoïdes finis. Nous n'entrerons pas ici dans les détails formels utiles à la définition précise de ces noyaux, ni sur leur intérêt. Il est important de noter que l'étude de la classification des monoïdes finis se place dans le cadre général de la théorie mathématique des variétés. Une variété de monoïdes finis est une classe de monoïdes finis ayant de bonnes propriétés de clôture. Par exemple, la classe de tous les monoïdes finis commutatifs est une variété, de même que la classe des monoïdes finis apériodiques.

La théorie des monoïdes finis, et particulièrement des variétés de monoïdes finis, est intéressante lorsque l'on étudie aux langages réguliers. En effet, il existe une bijection [Eil74] entre les variétés de monoïdes finis et les variétés de langages réguliers (qui sont aussi des classes de langages ayant de bonnes propriétés de clôture). Par exemple, la variété de monoïdes finis apériodiques correspond à la classe des langages sans étoile [Sch65]. Cette correspondance entre classes de monoïdes finis et classes de langages réguliers, approfondie notamment dans [PW97, GGP08a], permet d'obtenir des algorithmes efficaces (en général), de test d'appartenance à une classe de langages réguliers [PW97]. Par exemple, de cette théorie découle l'algorithme permettant de tester si un langage est reconnaissable par un automate partiellement ordonné (voir chapitre 3), en temps polynomial [PW97].

Le calcul des noyaux de monoïdes finis entre dans la problématique de décision de l'appartenance d'un monoïde fini à une variété de monoïdes : dans certains cas, décider si un monoïde fini appartient à une variété se réduit à savoir si un de ses noyaux appartient à une autre variété. Si cette dernière est décidable, le problème se réduit donc au calcul de ce noyau. Nous ne détaillerons pas plus la motivation qui demanderait des explications longues pour un sujet qui sort du cadre de ce document. Le lecteur intéressé pourra se référer à [HMPR91].

6.1.2 Calcul du noyau abélien et accessibilité

Nous allons expliquer ici comment (mais pas pourquoi), le calcul du noyau abélien d'un monoïde fini se réduit à un problème d'accessibilité contraint dans son graphe de Cayley.

On se donne un monoïde fini M engendré par A. On construit le A-graphe de Cayley $G_A(M)$ de M, c'est-à-dire le graphe dont les sommets sont les éléments de M et dans lequel il y a une transition entre deux éléments α_1 et α_2 étiquetée par $a \in A$ si $\alpha_1 a = \alpha_2$ (produit dans M).

Par exemple, on considère le monoïde donné par la table de multiplication suivante :

	1	s_1	s_2	s_3	s_4	0
1	1	s_1	s_2	s_3	s_4	0
s_1	s_1	0	s_3	0	s_1	0
s_2	s_2	s_4	0	s_4	0	0
s_3	s_3	s_3	0	s_3	0	0
s_4	s_4	0	s_4	0	s_4	0
0	0	0	0	0	0	0

Ce monoïde est engendré par $\{s_1, s_2\}$. On peut donc construire son $\{s_1, s_2\}$-graphe de Cayley, décrit dans la figure 6.1.

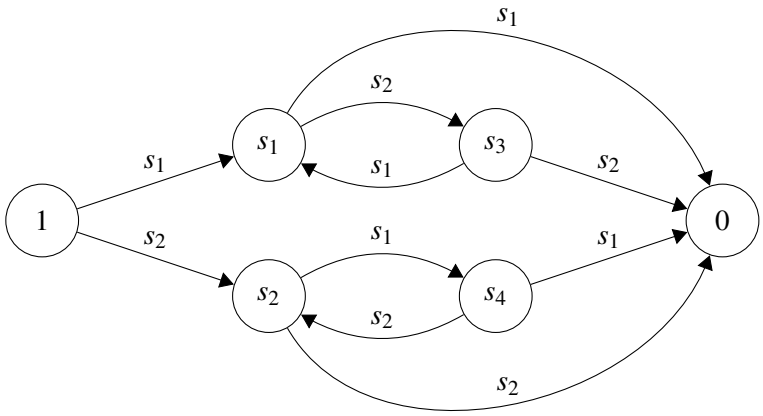


FIG. 6.1 – Exemple de graphe de Cayley

On pose $\overline{A}$ une copie disjointe de A. A partir du A-graphe de Cayley de M on construit un graphe G' ainsi : pour chaque transition (p, a, q) du graphe de

Cayley, si p et q sont dans la même composante fortement connexe, alors on ajoute une transition $(q, \overline{a}, p)$. Le résultat principal de [Del01] est le suivant.

Proposition 6.1 *Soit $G_A(M)$ le A-graphe de Cayley d'un monoïde fini M. Le noyau abélien de M est l'ensemble des éléments de M accessibles dans $G'_A(M)$ à partir de 1 par un chemin qui utilise autant de a que de $\overline{a}$, pour tout $a \in A$.*

Il est montré dans [Del01] comment utiliser cette propriété pour calculer le noyau abélien de M en temps exponentiel.

En nous appuyant sur la proposition 6.1, nous avons montré que si un élément de M est dans le noyau abélien, alors il est accessible dans $G'_A(M)$ à partir de 1 par un chemin de taille polynomial qui utilise autant de a que de $\overline{a}$, pour tout $a \in A$. Ce qui aboutit au résultat suivant.

Proposition 6.2 *Le noyau abélien d'un monoïde A-engendré peut se calculer en temps polynomial, si la taille de A est considérée comme une constante.*

6.1.3 Expérimentations et discutions

Le résultat de la proposition 6.2 est avant tout théorique. En effet, le degré du polynôme bornant la taille minimale d'un chemin de la forme souhaitée (autant de a que de $\overline{a}$), est très grand (16 fois $|A|^2$). Ce qui fait, qu'en pratique, l'approche est moins efficace que celle de [Del01].

Le tableau 6.1 illustre quelques résultats : chaque ligne représente un monoïde fini associé à un ensemble de générateurs. La première colonne est le nom du monoïde, la seconde sa taille, la troisième la taille du noyau abélien (calculée avec [Del01]). L'avant dernière colonne montre le logarithme de la borne que l'on obtient à partir de la proposition 6.2. Enfin, la dernière colonne indique le maximum des tailles effectives minimales d'un chemin contenant autant de a que de $\overline{a}$ et permettant d'atteindre un sommet du noyau (i.e. la borne optimale à laquelle on peut arrêter la recherche). On observe aisément que cette borne est très nettement inférieure à la borne k_M. On peut donc penser que si la borne obtenue dans la proposition 6.2 est trop grande, l'approche consistant à borner la taille des chemins est pertinente. Il resterait à travailler pour préciser les constantes, mais cela fait appel à des résultats complexes de calcul symbolique sur les matrices.

6.2 Comparaison d'automates max-plus

Nous nous intéressons ici à un tout autre problème : la comparaison d'automates max-plus.

Un automate max-plus est un automate pondéré par des entiers pour lequel on définit une fonction de coût sur les mots. Si un mot n'est pas reconnu par

Monoïde M	$\lvert M\rvert$	$\lvert A\rvert$	$\lvert\mathrm{Ker}_{\mathbf{Ab}}(M)\rvert$	$[\log_{10} k_M]$	α_M
$op3$	24	2	22	44	6
$popi3n$	31	2	14	47	6
$op4$	128	2	125	60	12
$popi4n$	141	2	94	61	8
$popi5n$	631	2	532	76	16
$poi3$	20	3	14	64	4
$T3$	27	3	24	68	4
$T4$	256	3	244	98	6
$podi3$	30	4	16	93	2
$poi4$	70	4	58	107	4
$popi3$	31	4	14	94	2
$popi4$	141	5	94	148	4
$pori3$	34	5	22	120	2
$podi4$	123	5	58	146	2
$pori4$	193	6	115	186	2
$podi5$	478	6	236	207	2

TAB. 6.1 – Calculs du noyau abélien

un automate son coût est $-\infty$. Sinon son coût est le maximum des coûts des chemins réussis le reconnaissant. Pour un automate max-plus $\mathcal{A}$ et un mot u, on note par $\mathcal{A}(u)$ le coût de u dans $\mathcal{A}$.

Le problème de positivité des automates max-plus, lié aux problématiques du chapitre 4, est le suivant.

AutomatePositif
Entrée : $\mathcal{A}$, un automate max-plus sur les entiers relatifs
Question : Pour tout mot w reconnu par $\mathcal{A}$, est-ce que $\mathcal{A}(w) \geq 0$?

Ce problème est indécidable [Kro94], mais il est décidable si $\mathcal{A}$ est finiment ambiguë [Web94, HIJ02]. Il est aussi décidable pour une sous-classe des automates polynomialement ambiguës [KL09].

6.2.1 Graphe d'accessibilité d'un automate max-plus

Commençons par traduire le problème AutomatePositif en un problème d'accessibilité sur un graphe (potentiellement) infini. Il s'agit d'un codage naturel et classique, même s'il est plus souvent écrit sous forme matricielle.

Soit $\mathcal{A} = (Q, \Sigma, E, I, F)$ un automate max-plus. Le *graphe de déterminisation* $\mathcal{G}(\mathcal{A}) = (V, \delta, s_0, K)$ de $\mathcal{A}$ est défini par

– $V = (\mathbb{Z} \cup -\infty)^Q$, $s_0 \in V$ et $s_0(p) = 0$ si $p \in I$, et $s_0(p) = -\infty$ sinon ;

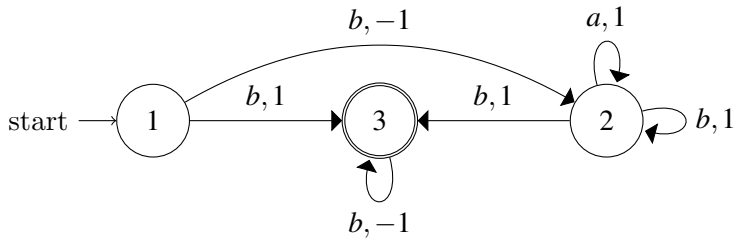


FIG. 6.2 – Exemple d'automate max-plus

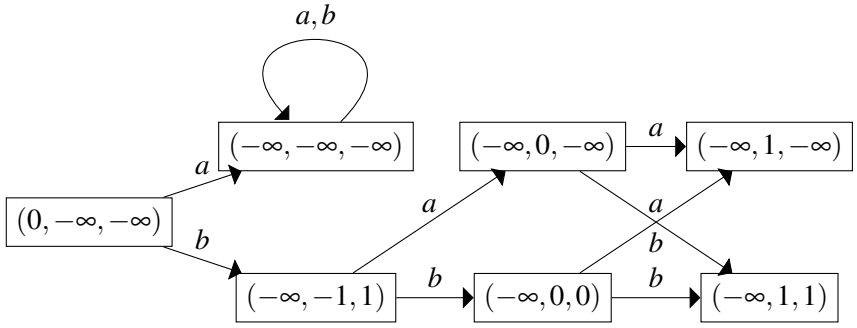


FIG. 6.3 – Une partie de $\mathcal{G}(\mathcal{A}_0)$

- $\delta \subseteq (V \times \Sigma) \times V$ est l'application définie par $\delta(s,a) = s'$ ssi $s'(p) = \max\{s(q) + c \mid (q,a,c,p) \in E\}$, avec la convention $\max \emptyset = -\infty$;
- $K = \{s \in V \mid \exists q \in F,\ s(q) \neq -\infty \text{ and } \forall p \in F,\ s(p) < 0\} \subseteq V$.

Considérons par exemple l'automate de la figure 6.2. En notant un élément s de $(\mathbb{Z} \cup -\infty)^{\{1,2,3\}}$ par (x,y,z) si $s(1) = x$, $s(2) = y$ et $s(3) = z$, le graphe d'accessibilité de l'automate de la figure 6.2 est le graphe

$$\mathcal{G}_0 = ((\mathbb{Z} \cup -\infty)^{\{1,2,3\}}, \delta_0, (0,-\infty,-\infty), K_0)$$

où $K_0 = \{(x,y,z) \mid z < 0 \text{ et } z \neq -\infty\}$. Une partie de δ_0 est décrite dans la figure 6.3.

Un automate max-plus $\mathcal{A}$ est dit *non-positif* si dans $\mathcal{G}(\mathcal{A})$ il existe un chemin de s_0 jusqu'à un élément de K. La proposition suivante lie cette notion avec le problème qui nous intéresse.

Proposition 6.3 *Soit $\mathcal{A} = (Q, \Sigma, E, I, F)$ un automate max-plus. Il existe $w \in L(\mathcal{A})$ tel que $\mathcal{A}(w) < 0$ si et seulement si $\mathcal{A}$ est non-positif.*

6.2.2 Exploration du graphe de déterminisation

Afin d'attaquer le problème de la positivité d'un automate max-plus, nous allons nous appuyer sur la proposition 6.3 en explorant le graphe de détermi-

nisation. On va de plus utiliser une technique de *rejet* de sommets. Pour cela, on introduit la relation $\preceq$ sur les sommets du graphe $\mathcal{G}(\mathcal{A})$ d'un automate max-plus $\mathcal{A}$. Cette relation est définie par : $s_1 \preceq s_2$ ssi pour chaque état p de $\mathcal{A}$, $s_1(p) = -\infty$ ssi $s_2(p) = -\infty$ et $s_1(p) \preceq s_2(p)$ sinon.

On a le résultat suivant.

Proposition 6.4 *Soit $\mathcal{A} = (Q, \Sigma, E, I, F)$ un automate max-plus et $\mathcal{G}(A) = (V, \delta, s_0, K)$ son graphe de déterminisation. Soient $s_1, s_2 \in (\mathbb{Z} \cup -\infty)^Q$ tels que $s_1 \preceq s_2$. Si un sommet s_2' de K est accessible dans $\mathcal{G}(\mathcal{A})$ depuis s_2, alors il existe s_1' dans K accessible depuis s_1.*

On peut alors utiliser l'algorithme suivant afin de semi-décider le problème de la positivité d'un automate max-plus. Dans cet algorithme, k désigne le nombre de sommets que l'on va explorer dans le graphe.

Algorithme 6.5

Nom : Exploration
Entrées : $\mathcal{A}$, k (entier positif)
Variables : L, C ensembles finis
Début
  $C := \emptyset$
  $L := \{s_0\}$
  TantQue $(k \geq 0)$
    Si $(C \cap K = \emptyset$ et $L = \emptyset)$
      Retourner 1
    FinSi
    Prendre $s \in L$
    Si $C \cap K \neq \emptyset$
      Retourner -1
    FinSi
    Si il n'existe pas $s' \in C$ telle que $s' \preceq s$
      $C := C \cup \{s\}$
      $L := L \cup \{\delta(s, a) \mid a \in \Sigma\}$
    FinSi
    $k := k - 1$
  FinTantQue
  Retourner 0
Fin

Dans cet algorithme , δ et K sont relatifs au graphe de déterminisation de $\mathcal{A}$. L'ensemble C est l'ensemble des sommets accessibles calculés. L'ensemble

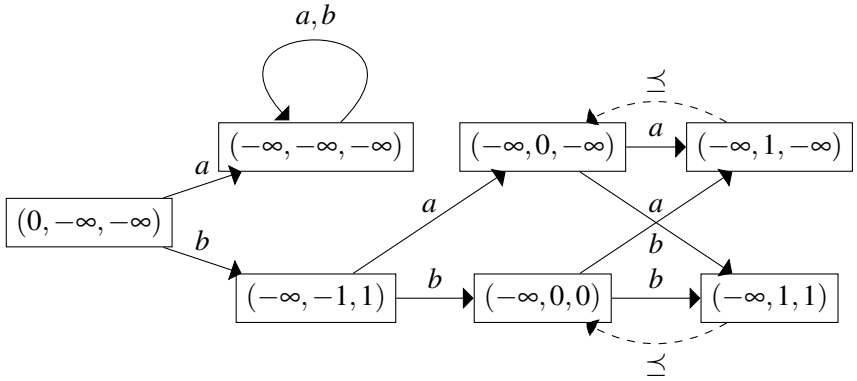


FIG. 6.4 – Exemple d'application de l'algorithme d'exploration

L code les sommets qui restent à explorer. La fonction **Prendre** retourne un élément de L qui lui est enlevé. Selon les structures de données choisies et l'implantation de cette fonction, on obtient différents types de parcours (en largeur, en profondeur, etc.). L'algorithme s'arrête en retournant 1 s'il n'y a plus de sommet à visiter : aucun sommet de K n'est accessible et, par conséquent, pour tout $u \in \Sigma^+$, $\mathcal{A}(u) \geq 0$. L'algorithme s'arrête en retournant -1 s'il calcule un état de K accessible. L'algorithme retourne 0 s'il ne peut pas conclure.

Par exemple, sur l'automate de la figure 6.2, l'algorithme va explorer le graphe de la figure 6.4, où les flèches pointillées représentent la relation $\preceq$. Sur cet exemple, l'algorithme s'arrêtera après quelques étapes en retournant 1.

6.2.3 Encodage par réécriture

La deuxième approche utilisée pour attaquer le problème est d'utiliser l'approche par complétion et automates d'arbre étudiée au chapitre 2.

Pour un automate max-plus donné, les sommets de son graphe de configuration sont codés par un terme. Par exemple le sommet s_0 du graphe de l'automate de la figure 6.2 sera codé par le terme $\mathsf{Conf}(0, \bot, \bot)$, où $\bot$ code $-\infty$. Les entiers sont codés avec l'arithmétique de Peano : 4 sera codé $s(s(s(s(0))))$, le symbole unaire s représentant le successeur. De même, $p(0)$ code l'entier -1, le symbole p représentant le prédécesseur.

Ensuite, on code dans un système de réécriture à la fois la fonction de transition du graphe et les propriétés arithmétiques. L'ensemble K étant régulier, on se ramène ainsi à l'analyse régulière d'accessibilité vue au chapitre 2 : on peut alors (essayer de) prouver, par sur-approximation, qu'aucun élément de K n'est accessible, et donc que l'automate est positif.

n	2	3	4	5	6	7	8	9	10	12	14	16	18	20
pos.	0.34	0.23	0.15	0.1	0.1	0.12	0.13	0.16	0.21	0.27	0.33	0.40	0.42	0.6
neg.	0.65	0.74	0.82	0.86	0.87	0.85	0.83	0.81	0.77	0.71	0.66	0.59	0.57	0.54
prof.	2.45	3.83	4.66	5.92	6.68	6.88	7.14	7.10	7.40	7.35	7.46	7.64	7.47	7.37
??	4	21	25	39	27	28	27	22	21	23	8	8	5	4
TRS	36	43	58	79	125	296	554	1068	2094	8242	10^5	10^6	10^6	10^7
inc.	0	4	6	10	6	T	T	T	T	T	T	T	T	T

TAB. 6.2 – Résultats expérimentaux

6.2.4 Expérimentations

Afin d'évaluer notre approche sur de nombreux exemples, nous avons choisi une évaluation aléatoire. On utilise pour cela la méthode de génération d'automates non déterministes suivante : étant donné un ensemble d'état $\{1,\dots,n\}$, pour chaque lettre a et chaque i,j, il y a une probabilité $p_{\text{transition}}$ d'avoir une transition de la forme (i,a,c,j). Si une telle transition existe, son poids est tirée aléatoirement entre $-c_{\max}$ and $c_{\max}$. De plus, dans ce modèle de génération, 1 est l'unique état initial et n est toujours final. De plus, pour chaque état autre que n, il y a une probabilité p_{final} que cet état soit final. Si un automate généré reconnaît le langage vide, il est rejeté.

Le tableau 6.2 décrit les résultats obtenus pour $c_{\max} = 3$, $p_{\text{transition}} = 0.3$ et $p_{\text{final}} = 0.1$. Pour chaque valeur de n entre 2 et 20, on a généré aléatoirement 1000 automates à n états. Pour chaque automate, on utilise d'abord l'algorithme 6.5 avec $k = 10n$. La ligne *pos.* (resp. *neg.*) montre la proportion d'entrées pour lesquelles l'algorithme retourne 1 (resp. -1). La ligne *??* le nombre d'automates (sur les 1000 générés pour chaque n) pour lesquels l'algorithme 6.5 retourne 0 (ne peut pas conclure). La ligne *prof.* donne le nombre moyen de sommets qu'il a fallu calculer pour obtenir une réponse (moyenne calculée sur les réponses 1 ou -1).

Lorsque l'algorithme 6.5 ne peut pas conclure, on applique alors la technique par réécriture, en utilisant la version 3 de TIMBUK [GR09]. La ligne *TRS* indique la taille approximative (en nombre de règles) du système de réécriture généré. La ligne *inc.* montre le nombre de cas qui restent indéfinis. Le résultat *T* montre que l'algorithme n'a pas pu terminer à cause d'un dépassement mémoire dans l'outil (par réécriture).

6.2.5 Perspectives et discutions

La première remarque est que le générateur aléatoire utilisé n'est pas très bon. En effet avec cette approche il n'y a absolument pas équiprobabilité dans la génération. S'il n'y a aucune méthode connue de *bonne* génération aléatoire d'automates non déterministes, on pourrait en revanche chercher d'autres méthodes de génération aléatoire pour évaluer notre approche (ce que nous avons

fait pour une méthode, inspirée de [TV05], qui produit des automates moins denses et pour laquelle les résultats de nos semi-algorithmes sont similaires). Cependant, les résultats obtenus sont encourageants : il semble fort possible qu'en pratique on puisse répondre, par semi-decision, au problème de positivité d'un automate max-plus.

Par ailleurs, des techniques d'accélération peuvent être envisagées afin de calculer, en une étape, une infinité de sommets accessibles dans le graphe de déterminisation. Dans ce cadre, [Lom01] montre comment *accélérer* une lettre. D'un point de vue plus général, les configurations étant des vecteurs d'entiers, les nombreux travaux sur l'accélération de systèmes à compteurs [Ler08, BFLP08, BW02, ABS01] pourraient être des moyens puissants de résoudre le problème en pratique sur les cas difficiles.

Chapitre 7

Bilan personnel

J'ai soutenu ma thèse au LIAFA (Université Paris 7, CNRS) en théorie des automates finis en 2001. Mon intégration au LIFC et dans le projet CASSIS sur des thématiques de vérification a demandé une légère réorientation thématique. Si les outils théoriques utilisés en model-checking sont proches des automates finis, les questions et le contexte diffèrent. Une des difficultés principales fut probablement de comprendre suffisamment l'état de l'art pour me poser de bonnes questions. En contrepartie, l'éclairage nouveau que cela m'apporta sur les automates fut très enrichissante.

Les premiers travaux sur les relations de semicommutation ont été une transition naturelle de la théorie des automates au model-checking régulier. Mais il est clair que les travaux sur les approximations régulières de langages d'arbre et la vérification de protocoles m'ont permis d'entrer plus franchement dans les problématiques du model-checking et de ses applications. L'encadrement de la thèse de Yohan Boichut et la participation au projet européen AVISPA ont été des moteurs essentiels pour mes travaux dans cette direction.

Mon intérêt pour les problématiques de vérification autour des composants est récente et provient de deux facteurs. D'une part, une partie de l'équipe VESONTIO travaille sur la modélisation des composants ainsi que leur vérification par raffinement. D'autre part, la politique générale de recherche en Franche-Comté nous oriente vers les réseaux de capteurs, qui sont des systèmes communicants simples (donc propices à une vérification efficace), et ouvrant des problématiques nouvelles (gestion fine de l'énergie, communications unidirectionnelles, etc.).

La problématique du test est plus récente et l'intérêt provient du contexte géographique : une partie importante de l'équipe VESONTIO du LIFC travaille sur la génération automatique de tests. Les séminaires, les soutenances de thèses et les discussions autour d'un café ont aiguisé ma curiosité. Si les motivations sont proches de celle de la vérification, la communauté et la discipline est différente. Le test est encore une discipline très expérimentale dans

laquelle il y a peu de travaux théoriques. On peut penser que la multiplication des expérimentations et l'importance industrielle du sujet devraient aboutir à des généralisations, des conceptualisations précises, des abstractions et à une théorie riche.

Sur un plan moins scientifique, la dizaine d'années que j'ai passées dans le monde de la recherche m'ont permis de connaître différents contextes de travail : thèse au LIAFA, poste au LIFC (enseignements en IUT, puis en UFR Sciences), équipe-projet INRIA CASSIS et une année de délégation au LSV (ENS Cachan, CNRS et INRIA). Ces différents changements ont été très enrichissants humainement et scientifiquement.

Table des figures

1.1 Jeu de morpion (vue partielle) 8
1.2 Analyse d'accessibilité 11
1.3 Approche par sur-approximation 14

2.1 Protocole `NSPK` 23
2.2 Faille du protocole NSPK 25
2.3 Exemple de complétion 28
2.4 Exemple après une étape de complétion 28
2.5 Exemple après deux étapes de complétion 28
2.6 Exemple après deux (autres) étapes de complétion 29
2.7 Autre exemple après deux (autres) étapes de complétion 29
2.8 Exemple d'approximation automatique 32
2.9 Exemple d'approximation automatique après une étape 33
2.10 Exemple d'approximation automatique après deux étapes . . . 33
2.11 Calculs à la volée pour les règles quadratiques 38
2.12 Un *run* de $\mathcal{A}^{(2)}$ sur $f(A,A)$ 39
2.13 Raffinement d'approximations. Les automates $\mathcal{A}_k$ sont les automates obtenus par complétion. Les automates $\mathcal{A}_k^e$ sont les automates obtenus par une complétion exacte à partir de $\mathcal{A}_{k-1}$. Les automates $\mathcal{A}_{\text{prop}}^k$ sont obtenus à partir de $\mathcal{A}_{\text{prop}}$ en effectuant k étapes de complétion exactes en utilisant $\mathcal{R}^{-1}$. 43
2.14 Modèle de $\Box(\mathcal{R}_1 \Rightarrow \circ\mathcal{R}_2)$ 46
2.15 Modèle de $\neg\mathcal{R}_2 \wedge \Box(\circ\mathcal{R}_2 \Rightarrow \mathcal{R}_1)$ 47
2.16 Modèle de $\Box(\mathcal{R}_1 \Rightarrow \Box\neg\mathcal{R}_2)$ 47

3.1 Automate partiellement ordonné. 56
3.2 Transitions pour le R-mélange 58
3.3 Automates $\mathcal{A}_1$ et $\mathcal{A}_2$ 59
3.4 R-mélange des automates $\mathcal{A}_1$ et $\mathcal{A}_2$. 59
3.5 Automates $\mathcal{A}_1$ et $\mathcal{A}_2$ 60
3.6 Heuristique de décomposition 63

4.1 Automates $\mathcal{A}_{\text{pond1}}$ et $\mathcal{A}_{\text{pond2}}$ 72
4.2 Composants C_1 et C_2 73
4.3 Automate booléen $\mathcal{A}_1$ 76
4.4 Automate booléen $\mathcal{A}_2$ 77
4.5 Automate booléen $\mathcal{A}_3$ 77
4.6 Automate $\mathcal{A}$ 78

5.1 Exemple de système à tester 87
5.2 Exemple d'automate généré 87
5.3 Produit des automates 88
5.4 Résultat de l'approche utilisant de la génération aléatoire d'automates 89
5.5 Un transducteur séquentiel lettre-à-lettre. 90
5.6 Nombre moyen d'états pour la transformation d'un automate d'arbre cheminant à n états en un automate d'arbre de haut en bas équivalent 93
5.7 Architecture de l'outil SEED 94
5.8 Règles de simplifications de formules LTL 96

6.1 Exemple de graphe de Cayley 101
6.2 Exemple d'automate max-plus 104
6.3 Une partie de $\mathcal{G}(\mathcal{A}_0)$ 104
6.4 Exemple d'application de l'algorithme d'exploration 106

Références personnelles (postérieures à la thèse)

[1] Alessandro Armando, David A. Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, Paul Hankes Drielsma, Pierre-Cyrille Héam, Olga Kouchnarenko, Jacopo Mantovani, Sebastian Mödersheim, David von Oheimb, Michaël Rusinowitch, Judson Santiago, Mathieu Turuani, Luca Viganò, and Laurent Vigneron. The avispa tool for the automated validation of internet security protocols and applications. In Kousha Etessami and Sriram K. Rajamani, editors, *CAV 2005*, volume 3576 of *Lecture Notes in Computer Science*, pages 281–285. Springer, 2005.

[2] Philippe Balbiani, Cheick Fahima, Pierre-Cyrille Héam, and Olga Kouchnarenko. Composition of services with constraints. In *FACS'09*, 2009.

[3] Yohan Boichut, Roméo Courbis, Pierre-Cyrille Héam, and Olga Kouchnarenko. Finer is better : Abstraction refinement for rewriting approximations. In Andrei Voronkov, editor, *RTA 2008*, volume 5117 of *Lecture Notes in Computer Science*, pages 48–62. Springer, 2008.

[4] Yohan Boichut, Roméo Courbis, Pierre-Cyrille Héam, and Olga Kouchnarenko. Handling left-quadratic rules when completing tree automata. *Electronic Notes in Theoretical Computer Sciences*, 223 :61–70, 2008. Actes de RP 2008.

[5] Yohan Boichut, Roméo Courbis, Pierre-Cyrille Héam, and Olga Kouchnarenko. Handling non left-linear rules when completing tree automata. *International Journal of Foundations of Computer Science*, 2009. A paraître.

[6] Yohan Boichut and Pierre-Cyrille Héam. A theoretical limit for safety verification techniques with regular fix-point computations. *Information Processing Letters*, 108(1) :1–2, 2008.

[7] Yohan Boichut, Pierre-Cyrille Héam, and Olga Kouchnarenko. Automatic abstraction generation : Hox to make an expert verification technique for

security protocols available to noon-expert users. Technical Report 6039, INRIA, 2006.

[8] Yohan Boichut, Pierre-Cyrille Héam, and Olga Kouchnarenko. Handling algebraic properties in automatic analysis of security protocols. In Kamel Barkaoui, Ana Cavalcanti, and Antonio Cerone, editors, *ICTAC 2006*, volume 4281 of *Lecture Notes in Computer Science*, pages 153–167. Springer, 2006.

[9] Yohan Boichut, Pierre-Cyrille Héam, and Olga Kouchnarenko. Tree automata for detecting attacks on protocols with algebraic crytpographic primitives. In *INFINITY'07*, pages 44–53, 2007. A paraître dans Electronic Notes in Theoretical Computer Sciences.

[10] Yohan Boichut, Pierre-Cyrille Héam, and Olga Kouchnarenko. Approximation based tree regular model-checking. *Nordic Journal of Computing*, 2009. A paraître.

[11] Yohan Boichut, Pierre-Cyrille Héam, and Olga Kouchnarenko. How to tackle integer weighted automata positivity. In *RP'2009*, Lecture Notes in Computer Science, 2009. A paraître.

[12] Yohan Boichut, Pierre-Cyrille Héam, Olga Kouchnarenko, and Frédéric Ohel. Improvements on the genet and klay technique to automatically verify security protocols. In *AVIS'2004*, pages 1–11, 2004.

[13] Gérard Cécé, Pierre-Cyrille Héam, and Yann Mainier. Clôtures transitives de semi-commutations et model-checking régulier. In *AFADL'04*, 2004.

[14] Gérard Cécé, Pierre-Cyrille Héam, and Yann Mainier. Clôtures transitives de semi-commutations et model-checking régulier. *Technique et Science Informatiques*, 27(1-2) :7–28, 2008.

[15] Gérard Cécé, Pierre-Cyrille Héam, and Yann Mainier. Efficiency of automata for semicommutation verification techniques. *Theoretical Informatics and Applications*, 42(2) :197–215, 2008.

[16] Roméo Courbis, Pierre-Cyrille Héam, and Olga Kouchnarenko. Taged approximations for temporal properties model-checking. In *CIAA 2009*, Lecture Notes in Computer Science, 2009. A paraître.

[17] Frédéric Dadeau, Pierre-Cyrille Héam, and Jocelyn Levrey. On the use od uniform random generation of automata for testing. In *MBT'09*, 2009. A paraître dans Electronic Notes in Theoretical Computer Sciences.

[18] Manuel Delagado and Pierre-Cyrille Héam. A polynomial time algorithm to compute the abelian kernel of a finite monoid. *Semigroup Forum*, 67 :97–110, 2003.

[19] Pierre-Cyrille Héam. On shuffle ideals. *Theoretical Informatics and Applications*, 36(4) :359–384, 2002.

[20] Pierre-Cyrille Héam. A note on partially ordered tree automata. *Information Processing Letters*, 108(4) :242–246, 2008.

[21] Pierre-Cyrille Héam, Olga Kouchnarenko, and Jérôme Voinot. Component simulation-based substitutivity managing qos aspects. In *FACS'08*. A paraître dans Electronic Notes in Theoretical Computer Sciences.

[22] Pierre-Cyrille Héam, Olga Kouchnarenko, and Jérôme Voinot. How to handle qos aspects in web services substitutivity verification. In *WETICE*, pages 333–338. IEEE Computer Society, 2007.

[23] Pierre-Cyrille Héam, Olga Kouchnarenko, and Jérôme Voinot. Towards formalizing qos of web services with weighted automata. Technical Report 6218, INRIA, 2007. Soumis á IJFCS, en cours de révision.

[24] Pierre-Cyrille Héam, Olga Kouchnarenko, and Jérôme Voinot. Component simulation-based substitutivity managing QoS features. 2009.

[25] Pierre-Cyrille Héam and Cyril Nicaud. SEED : an easy-to-use tool for the random generation of recursive data structures for testing. Soumis, 2009.

[26] Pierre-Cyrille Héam, Cyril Nicaud, and Sylvain Schmitz. Random generation of deterministic tree (walking) automata. In *CIAA 2009*, Lecture Notes in Computer Science, 2009. A paraître.

Bibliographie

[AAB99] Parosh Aziz Abdulla, Aurore Annichini, and Ahmed Bouajjani. Symbolic verification of lossy channel systems : Application to the bounded retransmission protocol. In Rance Cleaveland, editor, *TACAS*, volume 1579 of *Lecture Notes in Computer Science*, pages 208–222. Springer, 1999.

[AAB+05] Marc Aiguier, Agnès Arnould, Clément Boin, Pascale Le Gall, and Bruno Marre. Testing from algebraic specifications : Test data set selection by unfolding axioms. In Wolfgang Grieskamp and Carsten Weise, editors, *FATES*, volume 3997 of *Lecture Notes in Computer Science*, pages 203–217. Springer, 2005.

[AAGL07] Marc Aiguier, Agnès Arnould, Pascale Le Gall, and Delphine Longuet. Test selection criteria for quantifier-free first-order specifications. In Farhad Arbab and Marjan Sirjani, editors, *FSEN*, volume 4767 of *Lecture Notes in Computer Science*, pages 144–159. Springer, 2007.

[ABJ98] Parosh Aziz Abdulla, Ahmed Bouajjani, and Bengt Jonsson. On-the-fly analysis of systems with unbounded, lossy fifo channels. In Hu and Vardi [HV98], pages 305–318.

[ABS01] Aurore Annichini, Ahmed Bouajjani, and Mihaela Sighireanu. Trex : A tool for reachability analysis of complex systems. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *CAV*, volume 2102 of *Lecture Notes in Computer Science*, pages 368–372. Springer, 2001.

[ADG+08] Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors. *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II - Track B : Logic, Semantics, and Theory of Programming & Track C : Security and Cryptography Foundations*, volume 5126 of *Lecture Notes in Computer Science*. Springer, 2008.

[AJ96] Parosh Aziz Abdulla and Bengt Jonsson. Verifying programs with unreliable channels. *Inf. Comput.*, 127(2) :91–101, 1996.

[ALdR06] Parosh Aziz Abdulla, Axel Legay, Julien d'Orso, and Ahmed Rezine. Tree regular model checking : A simulation-based approach. *J. Log. Algebr. Program.*, 69(1-2) :93–121, 2006.

[Arf87] Mustapha Arfi. Polynomial operations on rational languages. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *STACS*, volume 247 of *Lecture Notes in Computer Science*, pages 198–206. Springer, 1987.

[Arf91] Mustapha Arfi. Opérations polynomiales et hiérarchies de concaténation. *Theor. Comput. Sci.*, 91(1) :71–84, 1991.

[Bac00] Leo Bachmair, editor. *Rewriting Techniques and Applications, 11th International Conference, RTA 2000, Norwich, UK, July 10-12, 2000, Proceedings*, volume 1833 of *Lecture Notes in Computer Science*. Springer, 2000.

[BBGM08] Emilie Balland, Yohan Boichut, Thomas Genet, and Pierre-Etienne Moreau. Towards an efficient implementation of tree automata completion. In José Meseguer and Grigore Rosu, editors, *AMAST*, volume 5140 of *Lecture Notes in Computer Science*, pages 67–82. Springer, 2008.

[BCT06] Boualem Benatallah, Fabio Casati, and Farouk Toumani. Representing, analysing and managing web service protocols. *Data Knowl. Eng.*, 58(3) :327–357, 2006.

[BDN07] Frédérique Bassino, Julien David, and Cyril Nicaud. : A library to randomly and exhaustively generate automata. In Jan Holub and Jan Zdárek, editors, *CIAA*, volume 4783 of *Lecture Notes in Computer Science*, pages 303–305. Springer, 2007.

[BDN09] Frédérique Bassino, Julien David, and Cyril Nicaud. Enumeration and random generation of possibly incomplete deterministic automata. *Pure Mathematics and Applications*, 2009.

[BFLP08] Sébastien Bardin, Alain Finkel, Jérôme Leroux, and Laure Petrucci. Fast : acceleration from theory to practice. *STTT*, 10(5) :401–424, 2008.

[BG96] Bernard Boigelot and Patrice Godefroid. Symbolic verification of communication protocols with infinite state spaces using qdds (extended abstract). In Rajeev Alur and Thomas A. Henzinger, editors, *CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 1996.

[BG06] Yohan Boichut and Thomas Genet. Feasible trace reconstruction for rewriting approximations. In Frank Pfenning, editor, *RTA*,

volume 4098 of *Lecture Notes in Computer Science*, pages 123–135. Springer, 2006.

[BGJ08] Benoît Boyer, Thomas Genet, and Thomas P. Jensen. Certifying a tree automata completion checker. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *IJCAR*, volume 5195 of *Lecture Notes in Computer Science*, pages 523–538. Springer, 2008.

[BGJR07] Yohan Boichut, Thomas Genet, Thomas P. Jensen, and Luka Le Roux. Rewriting approximations for fast prototyping of static analyzers. In Franz Baader, editor, *RTA*, volume 4533 of *Lecture Notes in Computer Science*, pages 48–62. Springer, 2007.

[BHRV06] Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz, and Tomás Vojnar. Abstract regular tree model checking. *Electr. Notes Theor. Comput. Sci.*, 149(1) :37–48, 2006.

[BHV04] Ahmed Bouajjani, Peter Habermehl, and Tomás Vojnar. Abstract regular model checking. In Rajeev Alur and Doron Peled, editors, *CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 372–386. Springer, 2004.

[BJNT00] Ahmed Bouajjani, Bengt Jonsson, Marcus Nilsson, and Tayssir Touili. Regular model checking. In Emerson and Sistla [ES00], pages 403–418.

[Bla01] Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *CSFW*, pages 82–96. IEEE Computer Society, 2001.

[BLP03] L. Bozga, Y. Lakhnech, and M. Perin. Pattern-based abstraction for verifying secrecy in protocols. In *9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems,TACAS 2003, Proceedings*, volume 2619 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[BLP04] Fabrice Bouquet, Bruno Legeard, and Fabien Peureux. Clps-b - a constraint solver to animate a b specification. *STTT*, 6(2) :143–157, 2004.

[BMT07] Ahmed Bouajjani, Anca Muscholl, and Tayssir Touili. Permutation rewriting and algorithmic verification. *Inf. Comput.*, 205(2) :199–224, 2007.

[BN07] Frédérique Bassino and Cyril Nicaud. Enumeration and random generation of accessible automata. *Theor. Comput. Sci.*, 381(1-3) :86–104, 2007.

[Boj08] Mikolaj Bojanczyk. Tree-walking automata. In Carlos Martín-Vide, Friedrich Otto, and Henning Fernau, editors, *LATA*, volume

5196 of *Lecture Notes in Computer Science*, pages 1–2. Springer, 2008.

[BSBM04] Lucas Bordeaux, Gwen Salaün, Daniela Berardi, and Massimo Mecella. When are two web services compatible ? In Ming-Chien Shan, Umeshwar Dayal, and Meichun Hsu, editors, *TES*, volume 3324 of *Lecture Notes in Computer Science*, pages 15–28. Springer, 2004.

[BT02] Ahmed Bouajjani and Tayssir Touili. Extrapolating tree transformations. In Ed Brinksma and Kim Guldstrand Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 539–554. Springer, 2002.

[BW02] Bernard Boigelot and Pierre Wolper. Representing arithmetic constraints with finite automata : An overview. In Peter J. Stuckey, editor, *ICLP*, volume 2401 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2002.

[CDG+02] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. 2002. Available at *http ://www.grappa.univ-lille3.fr/tata/*.

[CDL06] Véronique Cortier, Stéphanie Delaune, and Pascal Lafourcade. A survey of algebraic properties used in cryptographic protocols. *Journal of Computer Security*, 14(1) :1–43, 2006.

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In Emerson and Sistla [ES00], pages 154–169.

[CGS01] Fabio Casati, Dimitrios Georgakopoulos, and Ming-Chien Shan, editors. *Technologies for E-Services, Second International Workshop, TES 2001, Rome, Italy, September 14-15, 2001, Proceedings*, volume 2193 of *Lecture Notes in Computer Science*. Springer, 2001.

[CL05] David Coppit and Jiexin Lian. yagg : an easy-to-use generator for structured test inputs. In David F. Redmiles, Thomas Ellman, and Andrea Zisman, editors, *ASE*, pages 356–359. ACM, 2005.

[CLC04] Hubert Comon-Lundh and Véronique Cortier. Security properties : two agents are sufficient. *Sci. Comput. Program.*, 50(1-3) :51–71, 2004.

[CS07] Pierre Chambart and Ph. Schnoebelen. Post embedding problem is not primitive recursive, with applications to channel systems. In Vikraman Arvind and Sanjiva Prasad, editors, *FSTTCS*, volume 4855 of *Lecture Notes in Computer Science*, pages 265–276. Springer, 2007.

[CW05] Véronique Cortier and Bogdan Warinschi. Computationally sound, automated proofs for security protocols. In Shmuel Sagiv, editor, *ESOP*, volume 3444 of *Lecture Notes in Computer Science*, pages 157–171. Springer, 2005.

[DDGM07] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated testing of refactoring engines. In *ESEC/FSE 2007 : Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, New York, NY, USA, September 2007. ACM Press.

[Del01] Manuel Delgado. Commutative images of rational languages and the abelian kernel of a monoid. *ITA*, 35(5) :419–435, 2001.

[DGG+06] Alain Denise, Marie-Claude Gaudel, Sandrine-Dominique Gouraud, Richard Lassaigne, and Sylvain Peyronnet. Uniform random sampling of traces in very large models. In Johannes Mayer and Robert G. Merkel, editors, *Random Testing*, pages 10–19. ACM, 2006.

[DKR09] Stéphanie Delaune, Steve Kremer, and Mark Ryan. Verifying privacy-type properties of electronic voting protocols. *Journal of Computer Security*, 17(4) :435–487, 2009.

[DLMS99] N. Durgin, P. D. Lincoln, J. C. Mitchell, and A. Scedrov. Undecidability of Bounded Security Protocols. In *Formal Methods and Security Protocols, FMSP'99*, 1999. Available at *http ://www.cs.bell-labs.com/who/nch/fmsp99/program.html*.

[DM97] Volker Diekert and Yves Métivier. Partial commutation and traces. In G. Rozenberg and A. Salomaa, editors, *Handbook on Formal Languages*, volume III. Springer, Berlin-Heidelberg-New York, 1997.

[DSV08] Manfred Droste, Jacques Sakarovitch, and Heiko Vogler. Weighted automata with discounting. *Inf. Process. Lett.*, 108(1) :23–28, 2008.

[DT90] Max Dauchet and Sophie Tison. The theory of ground rewrite systems is decidable. In *LICS*, pages 242–248. IEEE Computer Society, 1990.

[dT06] Julien d'Orso and Tayssir Touili. Regular hedge model checking. In Gonzalo Navarro, Leopoldo E. Bertossi, and Yoshiharu Kohayakawa, editors, *IFIP TCS*, volume 209 of *IFIP*, pages 213–230. Springer, 2006.

[EH99] Joost Engelfriet and Hendrik Jan Hoogeboom. Tree-walking pebble automata. In Juhani Karhumäki, Hermann A. Maurer, Gheorghe Paun, and Grzegorz Rozenberg, editors, *Jewels are Forever*, pages 72–83. Springer, 1999.

[Eil74] Samuel Eilenberg. *Automata, languages and machines*, volume volume A and B. Academic Press, New York, 1974.

[EJ73] Jack Edmonds and Ellis L. Johnson. Matching, euler tours and the chinese postman. *Mathematical Programming*, 5(1) :88–124, 1973.

[EM07] S. Escobar and J. Meseguer. Symbolic model checking of infinite-state systems using narrowing. In *Rewriting Techniques and Applications, RTA'07*, pages 153–168, 2007.

[ES00] E. Allen Emerson and A. Prasad Sistla, editors. *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*. Springer, 2000.

[FGL09] Alain Finkel and Jean Goubault-Larrecq. Forward analysis for wsts, part i : Completions. In Susanne Albers and Jean-Yves Marion, editors, *STACS*, volume 09001 of *Dagstuhl Seminar Proceedings*, pages 433–444. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2009.

[FGT04] Guillaume Feuillade, Thomas Genet, and Valérie Viet Triem Tong. Reachability analysis over term rewriting systems. *J. Autom. Reasoning*, 33(3-4) :341–383, 2004.

[Fin94] Alain Finkel. Decidability of the termination problem for completely specified protocols. *Distributed Computing*, 7(3) :129–135, 1994.

[FS01] Alain Finkel and Ph. Schnoebelen. Well-structured transition systems everywhere ! *Theor. Comput. Sci.*, 256(1-2) :63–92, 2001.

[FTT08] E. Filiot, J.-M. Talbot, and S. Tison. Tree automata with global constraints. In *Developments in Language Theory*, pages 314–326, 2008.

[FUMK07] Howard Foster, Sebastián Uchitel, Jeff Magee, and Jeff Kramer. Ws-engineer : A model-based approach to engineering web service compositions and choreography. In Luciano Baresi and Elisabetta Di Nitto, editors, *Test and Analysis of Web Services*, pages 87–119. Springer, 2007.

[FZC94] Philippe Flajolet, Paul Zimmermann, and Bernard Van Cutsem. A calculus for the random generation of labelled combinatorial structures. *TCS*, 132(2) :1–35, 1994.

[GDG+08] Marie-Claude Gaudel, Alain Denise, Sandrine-Dominique Gouraud, Richard Lassaigne, Johan Oudinet, and Sylvain Peyron-

net. Coverage-biased random exploration of models. *Electr. Notes Theor. Comput. Sci.*, 220(1) :3–14, 2008.

[Gen98] Thomas Genet. Decidable approximations of sets of descendants and sets of normal forms. In Tobias Nipkow, editor, *RTA*, volume 1379 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 1998.

[GG08] Marie-Claude Gaudel and Pascale Le Gall. Testing data types implementations from algebraic specifications. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 209–239. Springer, 2008.

[GGP08a] Mai Gehrke, Serge Grigorieff, and Jean-Eric Pin. Duality and equational theory of regular languages. In Aceto et al. [ADG+08], pages 246–257.

[GGP08b] Antonio Cano Gómez, Giovanna Guaiana, and Jean-Eric Pin. When does partial commutative closure preserve regularity ? In Aceto et al. [ADG+08], pages 209–220.

[GK00] Thomas Genet and Francis Klay. Rewriting for cryptographic protocol verification. In David A. McAllester, editor, *CADE*, volume 1831 of *Lecture Notes in Computer Science*, pages 271–290. Springer, 2000.

[GKL08] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In Rajiv Gupta and Saman P. Amarasinghe, editors, *PLDI*, pages 206–215. ACM, 2008.

[GMSZ08] Blaise Genest, Anca Muscholl, Olivier Serre, and Marc Zeitoun. Tree pattern rewriting systems. In Sung Deok Cha, Jin-Young Choi, Moonzoo Kim, Insup Lee, and Mahesh Viswanathan, editors, *ATVA*, volume 5311 of *Lecture Notes in Computer Science*, pages 332–346. Springer, 2008.

[GO01] Paul Gastin and Denis Oddoux. Fast LTL to Büchi automata translation. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65, Paris, France, July 2001. Springer.

[GR08] John P. Gallagher and Mads Rosendahl. Approximating term rewriting systems : A horn clause specification and its implementation. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *LPAR*, volume 5330 of *Lecture Notes in Computer Science*, pages 682–696. Springer, 2008.

[GR09] Thomas Genet and Vlad Rusu. Equational approximation for tree automata completion. submitted paper, 2009.

[GRS04] Giovanna Guaiana, Antonio Restivo, and Sergio Salemi. On the trace product and some families of languages closed under partial commutations. *Journal of Automata, Languages and Combinatorics*, 9(1) :61–79, 2004.

[GT95] Rémi Gilleron and Sophie Tison. Regular tree languages and rewrite systems. *Fundam. Inform.*, 24(1/2) :157–174, 1995.

[GTTT03] T. Genet, Y.-M. Tang-Talpin, and V. Viet Triem Tong. Verification of copy-protection cryptographic protocol using approximations of term rewriting systems. In *WITS'03, Workshop on Issues in the Theory of Security*, 2003.

[Ham94] Richard Hamlet. Random testing. In *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

[HC83] Timothy J. Hickey and Jacques Cohen. Uniform random generation of strings in a context-free language. *SIAM J. Comput.*, 12(4) :645–655, 1983.

[Héa03] Pierre-Cyrille Héam. Some complexity results for polynomial rational expressions. *Theor. Comput. Sci.*, 1-3(299) :735–741, 2003.

[Hen03] Thomas A. Henzinger. Automata for specifying component interfaces. In Oscar H. Ibarra and Zhe Dang, editors, *CIAA*, volume 2759 of *Lecture Notes in Computer Science*, pages 1–2. Springer, 2003.

[Hig52] G. Higman. Ordering by divisibility in abstract algebra. *Proc. of the London Mathematical Society*, 2(7) :326–336, 1952.

[HIJ02] K. Hashiguchi, K. Ishiguro, and S. Jimbo. Decidability of the Equivalence Problem for Finitely Ambiguous Finance Automata. *IJAC*, 12(3), 2002.

[HMG06] Nicolas Halbwachs, David Merchat, and Laure Gonnord. Some ways to reduce the space dimension in polyhedra computations. *Formal Methods in System Design*, 29(1) :79–95, 2006.

[HMPR91] Karsten Henckell, Stuart W. Margolis, Jean-Eric Pin, and John Rohdes. Ash's type ii theorem, profinite topology and malcev products. *International Journal of Algebra and Computation*, 1 :411–436, 1991.

[HMR05] Thomas A. Henzinger, Rupak Majumdar, and Jean-François Raskin. A classification of symbolic transition systems. *ACM Trans. Comput. Log.*, 6(1) :1–32, 2005.

[Hol97] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Software Eng.*, 23(5) :279–295, 1997.

[HV98] Alan J. Hu and Moshe Y. Vardi, editors. *Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings*, volume 1427 of *Lecture Notes in Computer Science*. Springer, 1998.

[Jac96] Florent Jacquemard. Decidable approximations of term rewriting systems. In Harald Ganzinger, editor, *RTA*, volume 1103 of *Lecture Notes in Computer Science*, pages 362–376. Springer, 1996.

[JR08] Florent Jacquemard and Michaël Rusinowitch. Closure of hedge-automata languages by hedge rewriting. In Andrei Voronkov, editor, *RTA*, volume 5117 of *Lecture Notes in Computer Science*, pages 157–171. Springer, 2008.

[KL09] D. Kirsten and S. Lombardy. Deciding unambiguity and sequentiality of polynomially ambiguous min-plus automata. In *STACS*, pages 589–600, 2009.

[KM08] Michael Kaminski and Simone Martini, editors. *Computer Science Logic, 22nd International Workshop, CSL 2008, 17th Annual Conference of the EACSL, Bertinoro, Italy, September 16-19, 2008. Proceedings*, volume 5213 of *Lecture Notes in Computer Science*. Springer, 2008.

[Kro94] D. Krob. The Equality Problem for Rational Series with Multiplicities in the Tropical Semiring is Undecidable. *IJAC*, 4(3), 1994.

[Kru72] Joseph B. Kruskal. The theory of well-quasi-ordering : A frequently discovered concept. *J. Comb. Theory, Ser. A*, 13(3) :297–305, 1972.

[Ler08] Jérôme Leroux. Structural presburger digit vector automata. *Theor. Comput. Sci.*, 409(3) :549–556, 2008.

[Lom01] S. Lombardy. Sequentialization and unambiguity of (max,+) rational series over one letter. In S. Gaubert and J.-J. Loiseau, editors, *Workshop on max-plus Algebras and Their Applications to Discrete-event Systems, TCS, and Optimization*, Prague, 2001. IFAC, Elsevier Sciences.

[Low95] Gavin Lowe. An attack on the needham-schroeder public-key authentication protocol. *Inf. Process. Lett.*, 56(3) :131–133, 1995.

[LS00] François Laroussinie and Ph. Schnoebelen. The state explosion problem from trace to bisimulation equivalence. In Jerzy Tiuryn, editor, *FoSSaCS*, volume 1784 of *Lecture Notes in Computer Science*, pages 192–207. Springer, 2000.

[LS06] Ralf Lämmel and Wolfram Schulte. Controllable combinatorial coverage in grammar-based testing. In M. Ümit Uyar, Ali Y.

Duale, and Mariusz A. Fecko, editors, *TestCom*, volume 3964 of *Lecture Notes in Computer Science*, pages 19–38. Springer, 2006.

[LS07] Christof Löding and Alex Spelten. Transition graphs of rewriting systems over unranked trees. In Ludek Kucera and Antonín Kucera, editors, *MFCS*, volume 4708 of *Lecture Notes in Computer Science*, pages 67–77. Springer, 2007.

[LU07] Bruno Legeard and Marc Utting. *Practical Model-based Testing : A Tools Approach*. Morgan Kaufmann Publishers, 2007.

[LZ88] Yaxiong Lin and Yongchang Zhao. A new algorithm for the directed chinese postman problem. *Computers & OR*, 15(6) :577–584, 1988.

[Mau92] Peter M. Maurer. The design and implementation of a grammar-based data generator. *Softw., Pract. Exper.*, 22(3) :223–244, 1992.

[MBTS04] Glenford.J. Myers, Tom Badget, Tood Thomas, and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, second edition, 2004.

[McK97] Bruce McKenzie. Generating string at random from a context-free grammar. Technical Report TR-COSC 10/97, Univ ersity of Canterbury, 1997.

[Mes92] J. Meseguer. Conditioned rewriting logic as a united model of concurrency. *Theoretical Computer Science*, 96(1) :73–155, 1992.

[Mes07] J. Meseguer. The temporal logic of rewriting. Technical Report UIDCS-R-2007-2815, Dept of Computer Science, University of Illinois at Urbana-Champaign, September 2007.

[Mes08] J. Meseguer. The temporal logic of rewriting : A gentle introduction. *Concurrency, Graphs and Models : Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*, pages 354–382, 2008.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[MM01] R. Marlet and D. Le Metayer. Security properties and java card specificities to be studied in the secsafe project, 2001.

[Mor00] Pierre-Etienne Moreau. Rem (reduce elan machine) : Core of the new elan compiler. In Bachmair [Bac00], pages 265–269.

[MP96] Anca Muscholl and Holger Petersen. A note on the commutative closure of star-free languages. *Inf. Process. Lett.*, 57(2) :71–74, 1996.

[MPC01] Massimo Mecella, Barbara Pernici, and Paolo Craca. Compatibility of e -services in a cooperative multi-platform environment. In Casati et al. [CGS01], pages 44–57.

[MW07] Anca Muscholl and Igor Walukiewicz. A lower bound on web services composition. In Helmut Seidl, editor, *FoSSaCS*, volume 4423 of *Lecture Notes in Computer Science*, pages 274–286. Springer, 2007.

[MX07] Rupak Majumdar and Ru-Gang Xu. Directed test generation using symbolic grammars. In *ASE*, pages 134–143, 2007.

[NW78] Albert Nijenhuis and Herbert S. Wilf. *Combinatorial Algorithms*. Academic Press, 1978.

[OCKS02] Frédéric Oehl, Gérard Cécé, Olga Kouchnarenko, and David Sinclair. Automatic approximation for the verification of cryptographic protocols. In Ali E. Abdallah, Peter Ryan, and Steve Schneider, editors, *FASec*, volume 2629 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2002.

[OT05] Hitoshi Ohsaki and Toshinori Takai. Actas : A system design for associative and commutative tree automata theory. *Electr. Notes Theor. Comput. Sci.*, 124(1) :97–111, 2005.

[PBH07] Jyotishman Pathak, Samik Basu, and Vasant Honavar. On context-specific substitutability of web services. In *ICWS*, pages 192–199. IEEE Computer Society, 2007.

[Pnu77] A. Pnueli. The temporal logic of programs. In *FOCS'77*, pages 46–57, 1977.

[Pur72] P. Purdom. A sentence generator for testing parsers. *BIT*, 12(3) :366–375, 1972.

[PW97] Jean-Eric Pin and Pascal Weil. Ponynominal closure and unambiguous product. *Theory Comput. Syst.*, 30(4) :383–422, 1997.

[Rab97] Alexander Moshe Rabinovich. Complexity of equivalence problems for concurrent systems of finite agents. *Inf. Comput.*, 139(2) :111–129, 1997.

[Rét99] Pierre Réty. Regular sets of descendants for constructor-based rewrite systems. In Harald Ganzinger, David A. McAllester, and Andrei Voronkov, editors, *LPAR*, volume 1705 of *Lecture Notes in Computer Science*, pages 148–160. Springer, 1999.

[RT03] Michaël Rusinowitch and Mathieu Turuani. Protocol insecurity with a finite number of sessions, composed keys is np-complete. *Theor. Comput. Sci.*, 1-3(299) :451–475, 2003.

[Sak92] Jacques Sakarovitch. The "last" decision problem for rational trace languages. In Imre Simon, editor, *LATIN*, volume 583 of *Lecture Notes in Computer Science*, pages 460–473. Springer, 1992.

[Sak03] Jacques Sakarovitch. *Elément de théorie des automates*. Les classiques de l'informatique. Vuibert Informatique, 2003.

[Sal88] Kai Salomaa. Deterministic tree pushdown automata and monadic tree rewriting systems. *J. Comput. Syst. Sci.*, 37(3) :367–394, 1988.

[Saw03] Zdenek Sawa. Equivalence checking of non-flat systems is exptime-hard. In Roberto M. Amadio and Denis Lugiez, editors, *CONCUR*, volume 2761 of *Lecture Notes in Computer Science*, pages 233–248. Springer, 2003.

[SB00] Fabio Somenzi and Roderick Bloem. Efficient büchi automata from ltl formulae. In Emerson and Sistla [ES00], pages 248–263.

[SBB+99] Philippe Schnoebelen, Béatrice Bérard, Michel Bidoit, François Laroussinie, and Antoine Petit. *Vérification de logiciels : techniques et outils du model-checking*. Vuibert, April 1999.

[Sch65] Marcel Paul Schützenberger. On finite monoids having only trivial subgroups. *Information and Control*, 8(2) :190–194, 1965.

[STFK02] Hiroyuki Seki, Toshinori Takai, Youhei Fujinaka, and Yuichi Kaji. Layered transducing term rewriting system and its recognizability preserving property. In Sophie Tison, editor, *RTA*, volume 2378 of *Lecture Notes in Computer Science*, pages 98–113. Springer, 2002.

[STV01] Thomas Schwentick, Denis Thérien, and Heribert Vollmer. Partially-ordered two-way automata : A new characterization of da. In Werner Kuich, Grzegorz Rozenberg, and Arto Salomaa, editors, *Developments in Language Theory*, volume 2295 of *Lecture Notes in Computer Science*, pages 239–250. Springer, 2001.

[Tak04] Toshinori Takai. A verification technique using term rewriting systems and abstract interpretation. In Vincent van Oostrom, editor, *RTA*, volume 3091 of *Lecture Notes in Computer Science*, pages 119–133. Springer, 2004.

[TBFM06] Yehia Taher, Djamal Benslimane, Marie-Christine Fauvet, and Zakaria Maamar. Towards an approach forweb services substitution. In *IDEAS*, pages 166–173. IEEE Computer Society, 2006.

[tCS08] Balder ten Cate and Luc Segoufin. XPath, transitive closure logic, and nested tree walking automata. In Maurizio Lenzerini and Domenico Lembo, editors, *PODS'08*, pages 251–260, 2008.

[Thi03] Harold W. Thimbleby. The directed chinese postman problem. *Softw., Pract. Exper.*, 33(11) :1081–1096, 2003.

[Tho82] Wolfgang Thomas. Classifying regular events in symbolic logic. *J. Comput. Syst. Sci.*, 25(3) :360–376, 1982.

[Tho97] *Handbook of Formal Languages*, volume 3, chapter Beyond Words. Springer, Berlin, 1997.

[TKS00] Toshinori Takai, Yuichi Kaji, and Hiroyuki Seki. Right-linear finite path overlapping term rewriting systems effectively preserve recognizability. In Bachmair [Bac00], pages 246–260.

[TT02] P. Tesson and D. Thérien. Diamonds are forever : the variety da. In *International Conference on Semigroups, Algorithms, Automata and Languages*, 2002.

[TV05] D. Tabakov and M.Y. Vardi. Experimental evaluation of classical automata constructions. In G. Sutcliffe and A. Voronkov, editors, *LPAR*, volume 3835 of *LNCS*, pages 396–411. Springer, 2005.

[WB98] Pierre Wolper and Bernard Boigelot. Verifying systems with infinite but regular state spaces. In Hu and Vardi [HV98], pages 88–97.

[Web94] A. Weber. Finite-valued Distance Automata. *Theoretical Computer Science*, 134, 1994.